PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Programs are Predicates [and Discussion]

C. A. R. Hoare and F. K. Hanna

Phil. Trans. R. Soc. Lond. A 1984 **312**, 475-489
doi: 10.1098/rsta.1984.0071

Phil. Trans. R. Soc. Lond. A **312**, 475–489 (1984) [475]
Printed in Great Britain

Programs are predicates

By C. A. R. Hoare, F.R.S.
Programming Research Group, Oxford University Computing Laboratory,
8–11 *Keble Road, Oxford OX*1 3*QD, U.K.*

A computer program is identified with the strongest predicate describing every relevant observation that can be made of the behaviour of a computer executing that program. A programming language is a subset of logical and mathematical notations, which is so restricted that products described in the language can be automatically implemented on a computer. The notations enjoy a number of elegant algebraic properties, which can be used for optimizing program efficiency.

A specification is a predicate describing all permitted observations of a program, and it may be expressed with greatest clarity by taking advantage of the whole language of logic and mathematics. A program P meets its specification S iff

$$\models P \Rightarrow S.$$

The proof of this implication may use all the classical methods of mathematics and logic.

These points are illustrated by design of a small language that includes assignments, conditionals, non-determinism, recursion, input, output, and concurrency.

1. Introduction

It is the aim of the natural scientist to discover mathematical theories, formally expressed as predicates describing the relevant observations that can be made of some physical system. A physical system is fully defined by the strongest predicate that describes it. Such predicates contain free variables, standing for values determined by observation, for example 'a' for acceleration, 'v' for velocity, 't' for time, etc.

The aim of an engineer is complementary to that of the scientist. He starts with a specification, formally expressible as a predicate describing the desired observable behaviour of a system or product not yet in existence. Then, with a limited set of tools and materials, within a limited timescale and budget, he must design and construct a product that meets that specification. The product is fully defined by the strongest specification that it meets.

For example, an electronic amplifier may be required to amplify its input voltage by a factor of ten. However, a condition of its correct working is that the input voltage must be held in the range 0 to 1 V. Furthermore, a margin of error of up to one volt is allowed on the output. This informal specification may be formalized as a predicate, with free variables:

V_i, standing for the *ith* observed input voltage;
$\tilde{V}_i$, standing for the *ith* observed output voltage.

Then the specification is

$$\forall i \,.\, (i \leqslant j \Rightarrow 0 \leqslant V_i \leqslant 1) \Rightarrow |\tilde{V}_j - 10 \times V_j| \leqslant 1.$$

Table 1 (*a*) and (*b*) show the first six observations made of two different amplifiers. The first observation of each amplifier shows it working with perfect accuracy at the midpoint of its range. The second observation is only just within the margin of tolerance. On the third observation the amplifier reveals its 'non-determinism': it does not always give the same output voltage for the same input voltage. On the fourth observation something goes wrong. For 4 (*a*) it is the amplifier that has gone wrong, because the five volt output is outside the permitted margin of error. Even if every subsequent observation is satisfactory, this product has not met its specification, and should be returned to its maker. For 4 (*b*), it is the observer who is at fault in supplying an excessive input of 1.3 V. As a result, the amplifier breaks, and its subsequent behaviour is entirely unconstrained: no matter what it does, it continues to meet its original specification. So on the sixth observation, it is the observer who returns to his Maker.

TABLE 1. OBSERVATIONS MADE OF TWO DIFFERENT AMPLIFIERS

observation number	(*a*)		(*b*)	
	V	$\tilde{V}$	V	$\tilde{V}$
1	0.5	5	0.5	5
2	0.4	5	0.4	5
3	0.5	4	0.5	4
4	0.3	5	1.3	13
5	0.6	6	0.6	6
6	0.7	7	0.7	997

The serious point of this example is to illustrate the usefulness of material implication in a specification. The consequent of the implication describes the desired relation between the inputs and the outputs of the system. The antecedent describes the assumptions that must be satisfied by the inputs of the system for it to continue working. If the assumptions are falsified, the product may break, and its subsequent (but not its previous) behaviour may be wholly arbitrary. Even if it seems to work for a while, it is completely worthless, unreliable, and even dangerous.

A computer programmer is an engineer whose main materials are the notations and structures of his programming language. A program is a detailed specification of the behaviour of a computer executing that program. Consequently, a program can be identified abstractly with a predicate describing all relevant observations that may be made of this behaviour. This identification assigns a meaning to the program (Floyd 1967), and a semantics to the programming language in which it is expressed.

These philosophical remarks lead to the main thesis of this paper, namely that programs are predicates. However, the converse claim would be incorrect, because any predicate that is wholly unsatisfiable (for example the predicate false) cannot correspond to a program. If it did, the behaviour of a computer executing that program would be wholly unobservable! Consequently, every observation of that behaviour would satisfy every specification! A product that satisfies every need is known as a *miracle*. Since such a product is also in principle unobservable, philosophical considerations lead us to suppose that it does not exist. Certainly any notation in which such a miracle could be expressed would not be an implementable programming language. There are also obvious practical reasons for ensuring that all predicates expressible as programs are in some sense computable, and can be computed at a cost that is controllable by the programmer and acceptable to his client.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

The design of a programming notation requires a preliminary selection of what are the relevant observable phenomena, and a choice of free variables to denote them. A meaning must then be given to the primitive components of the language, and to the operators that compose programs from smaller subprograms. Ideally, these operators should have pleasant algebraic properties, which permit proof of the identity of two programs whenever they are indistinguishable by observation. The achievement of these ideals is far from easy: so the language introduced in the next section for illustrative purposes has been kept very simple. It includes non-determinism, output, input, recursion, concurrency, assignment, and conditional.

2. A simple programming language

The first and simplest predicate that is expressible in our simple programming language is the predicate 'true'. This predicate is satisfied by *all* observations. If this is the *strongest* specification of a product, then there is no constraint whatever on the behaviour or misbehaviour of the product. The only customer who is certain to be satisfied with this product is one who would be satisfied by *anything*. Thus the program 'true' is the most useless of all products, just as a tautology is the most useless of scientific theories.

Now the most useless of computer programs is one that immediately goes into an infinite loop or recursion. Such a program is clearly broken or unstable, and can satisfy only the most undemanding customer. Thus we identify the infinitely looping program with the predicate 'true'. This may be a controversial decision; but in practice the ascription of a meaning to a divergent program is arbitrary, because no programmer will ever deliberately want to write a program that runs any risk of looping forever.

Non-determinism

The first and simplest operator of our programming language is disjunction. If P and Q are programs, the program $(P \vee Q)$ behaves either like P or like Q. There is no way of controlling or predicting the choice between P and Q; the choice is arbitrary or non-deterministic. All that is known is that each observation of $(P \vee Q)$ must be an observation of the behaviour of P or of Q or of both.

The algebraic properties of disjunction are very familiar: it is idempotent, symmetric, associative, etc. Furthermore, it is *distributive* (through disjunction) and *strict* in the sense that

$$p \vee \text{true} = \text{true} \vee P = \text{true}.$$

This means that if either P or Q may break then so may $(P \vee Q)$. To an engineer, a product that *may* break is as bad as one that *does*, because you can never rely on it.

Processes

Now we must be more specific about the nature of the objects described by programs in our simple language. These objects are called *processes*; a process should be regarded as a 'black box' connected to its environment by two wires. One of the wires is used for input of discrete messages, and the other for output (figure 1). A process engages in an unbounded sequence of communications, each of which is either an input from the input wire or an output to the output wire (but not both). If the environment is not ready for the communication, the process waits for it to become so, and vice versa. There is no 'buffer' in the wire; the act of communication requires simultaneous synchronized participation of both the sender and the receiver.

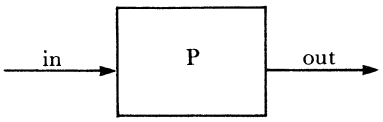

FIGURE 1. A process.

We postulate that the passing of a message on either wire is an observable aspect of the behaviour of the process. Imagine that there is a tape recorder attached to each of the wires, recording each message as it passes, but not recording the length of the gaps between the messages. At any moment, we can observe the current content of each of the two tapes. We introduce the free variable 'in' to stand for the current content of the tape recorder on the input wire, and 'out' to stand for the sequence of messages recorded from the output wire.

We also postulate that the internal state of a process cannot be directly observed: the black box has no openable lid. However, we assume that we can observe, by a green light perhaps, whether the process is working properly. This will be indicated by the value of a free Boolean variable 'stable', which takes as value either true or false. If ever 'stable' goes false, the machine is broken, and anything may happen (beware!).

Specifications

To formulate specifications, we need some notations to describe sequences of messages:

$\langle\ \rangle$ is the empty sequence, containing no messages. This is the value of both variables 'in' and 'out' if they are observed at the very start of the process;

$|s|$ is the length of s.

If s is a sequence other than $\langle\ \rangle$,

s_0 is the first message of s,

s' is the result of removing the first message of s,

$s^\dagger$ is the result of removing the last message of s (truncation).

If s and t are sequences,

$s\hat{}t$ is the result of concatenating s and t in this order

$s \leqslant t \triangleq \exists u.\ s\hat{}u = t$, i.e. s is an initial segment of t.

This is clearly a partial order with bottom $\langle\ \rangle$.

Using these notations, we can describe the behaviour of certain simple processes. For example, a process that just copies messages from its input to its output is always stable, and every observation of it shows the output sequence either exactly equal to the input sequence or one shorter:

$$\text{COPY} \triangleq \text{stable} \wedge (\text{out} = \text{in} \vee \text{out} = \text{in}^\dagger).$$

This copying process must always output each message immediately after inputting it. A more general buffering process relaxes this constraint. For example, a double buffer may be specified:

$$\text{BUFF2} \triangleq \text{stable} \wedge \text{out} \in \{\text{in}, \text{in}^\dagger, \text{in}^{\dagger\dagger}\}.$$

An unbounded buffer ensures that the output is always a copy of some initial segment of the input sequence

$$\text{BUFF} \triangleq \text{stable} \wedge \text{out} \leqslant \text{in}.$$

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Thus we see how predicates with free variables 'in', 'out' and 'stable' (together with conventional mathematical notations) can effectively describe and specify the behaviour of processes. But *none* of these notations can feature in our simple programming language; nor can they be included in any other programming language, since they can be used to express unsatisfiable predicates, such as

$$\text{in} \leqslant \text{out} \wedge \text{out} \leqslant \text{in} \wedge \text{in} \neq \text{out}$$

or unimplementable predicates like

$$\text{stable} \wedge |\text{in}| \geqslant 3,$$

which requires a process to input three messages before it starts! Our programming language must therefore be restricted to notations defined in the remaining paragraphs of this section. The restrictions will also ensure that no process can ever stop, so it will be impossible to implement the specification: $\text{stable} \wedge |\text{in}| + |\text{out}| \leqslant k$.

Output

Let P be a predicate exactly describing the behaviour of a process, and let e be a term composed of (say) constants, variables, and a fixed selection of primitive recursive functions. We introduce the notation

$$!e \rightarrow P$$

to describe the process that first outputs the value of e on its output wire, and then behaves as described by P.

The very first observation of the behaviour of $(!e \rightarrow P)$ is that it is stable and that the sequences of input and output messages are both empty. In every subsequent observation, the output sequence is nonempty, and its first message has value e. Furthermore, on removing the first message from the output sequence, the resulting observation will be an observation of the behaviour of P. These remarks explain the definition:

$$\begin{aligned} !e \rightarrow P(\text{in}, \text{out}, \text{stable}) \triangleq\ & \text{out} = \text{in} = \langle\ \rangle \wedge \text{stable} \\ & \vee\ \text{out} \neq \langle\ \rangle \leqslant \text{out}_0 = e \wedge P(\text{in}, \text{out}', \text{stable}). \end{aligned}$$

This operator is distributive but not strict:

$$!e \rightarrow (P \vee Q) = (!e \rightarrow P) \vee (!e \rightarrow Q).$$

As an example, we give

$$\begin{aligned} (!x \rightarrow \text{COPY}) = (&(\text{in} = \text{out} = \langle\ \rangle \wedge \text{stable}) \\ &\vee (\text{out} \neq \langle\ \rangle \wedge \text{out}_0 = x \\ &\wedge \text{stable} \wedge \text{out}' \in \{\text{in}, \text{in}^\dagger\})). \end{aligned}$$

Notice how the output has introduced the free variable x into the formula.

Input

Let P(x) be a predicate (possibly containing the variable x among its free variables) that describes exactly the behaviour of a process as a function of the initial value of x. Then we introduce the notation

$$?x \rightarrow P(x)$$

to describe the process that first inputs a value on its input wire, and then behaves like P(v), where v is the value it has just input.

The initial observation of the behaviour of $(?x \rightarrow P(x))$ is exactly the same as that of a process that starts with an output. In every subsequent observation, the input sequence is nonempty. Furthermore, on removing the first message from the input sequence, the resulting observation will be an observation of $P(in_0)$, i.e. the process that results from setting the initial value of x to in_0. These remarks explain the definition:

$$?x \rightarrow P(x, in, out, stable) \triangleq ((out = in = \langle\ \rangle \wedge stable) \vee (in \neq \langle\ \rangle \wedge P(in_0, in', out, stable))).$$

This operator is distributive, and binds the variable specified:

$$(?x \rightarrow (P(x) \vee Q(x))) = ((?x \rightarrow P(x)) \vee (?x \rightarrow Q(x)))$$
$$(?x \rightarrow P(x)) = (?y \rightarrow P(y)) \text{ when x is not free in } P(y) \text{ and y is not free in } P(x).$$

As an example we give

$$\begin{aligned}(?x \rightarrow (!x \rightarrow COPY)) = ((&in = out = \langle\ \rangle \wedge stable)\\ &\vee in \neq \langle\ \rangle \wedge (in' = out = \langle\ \rangle \wedge stable\\ &\qquad \vee (out \neq \langle\ \rangle \wedge out_0 = in_0\\ &\qquad\quad \wedge stable \wedge out' \in \{in', in'^{\uparrow}\})))\\ &= stable \wedge (in = out = \langle\ \rangle\\ &\qquad \vee |in| = 1 \wedge out = \langle\ \rangle\\ &\qquad \vee out_0 = in_0 \wedge out' \in \{in', in'^{\uparrow}\})\\ &= COPY.\end{aligned}$$

Note how the input has eliminated the free variable x from the formula. Note also that COPY is the solution for ξ in the equation

$$\xi = (?x \rightarrow (!x \rightarrow \xi)).$$

Recursion

Let ξ be a variable standing for an unknown process. Let $P(\xi)$ be a formula containing ξ, but otherwise containing only the notations of our simple programming language: disjunction, output, input, and the constant 'true'. Consider now the equation

$$\xi = P(\xi).$$

This may be taken as a recursive definition of a process with name ξ and body $P(\xi)$. Every

time ξ is encountered in the body, it stands for another copy of the whole body $P(\xi)$. The predicate that is the weakest solution to this equation will be denoted

$$\mu\xi \,.\, P(\xi).$$

But does such a solution exist? D. S. Scott has shown how to answer this question. Consider the sequence of predicates

$$\text{true}, P(\text{true}), P(P(\text{true})), \ldots, P^n(\text{true}), \ldots$$

and define

$$\mu\xi \,.\, P(\xi) \triangleq \forall n \geqslant 0 \,.\, P^n(\text{true}).$$

The fact that this is the weakest solution to the equation given above depends on the fact that $P(\xi)$ is a *continuous* function of ξ, in the sense that it distributes over the universal quantification of descending chains of predicate, i.e.

$$P(\forall n \geqslant 0 \,.\, Q_n) = \forall n \geqslant 0 \,.\, P(Q_n) \text{ whenever } \models Q_{n+1} \Rightarrow Q_n \text{ for all } n.$$

The continuity of all programs expressed in our simple language is assured by the fact that each operator of the language is continuous, and the composition of continuous operators is also continuous. We therefore have good reason to insist that all future operators introduced into the language must also be continuous.

The simplest example of recursion is the infinite loop

$$\mu\xi \,.\, \xi = \forall n \geqslant 0 \,.\, \text{true} = \text{true}.$$

A more interesting example is the program that copies messages from its input to its output

$$\mu\xi \,.\, (?x \rightarrow !x \rightarrow \xi) = \forall n \geqslant 0 \,.\, P_n,$$

where $P_0 = \text{true}$

and $P_{n+1} = (?x \rightarrow !x \rightarrow P_n)$.

The first few terms of the series are

$$\begin{aligned} P_1 = (&\text{in} = \text{out} = \langle\,\rangle \wedge \text{stable} \\ &\vee \text{in} \neq \langle\,\rangle \wedge (\text{in}' = \text{out} = \langle\,\rangle \wedge \text{stable} \\ &\qquad \vee \text{out} \neq \langle\,\rangle \wedge \text{out}_0 = \text{in}_0 \wedge \text{true})), \end{aligned}$$

$$\begin{aligned} P_2 = (&\text{in} = \text{out} = \langle\,\rangle \wedge \text{stable} \\ &\vee \text{in}' = \text{out} = \langle\,\rangle \wedge \text{stable} \\ &\vee \text{in}' = \text{out}' = \langle\,\rangle \wedge \text{in}_0 = \text{out}_0 \wedge \text{stable} \\ &\vee \text{in}'' = \text{out}' = \langle\,\rangle \wedge \text{in}_0 = \text{out}_0 \wedge \text{stable} \\ &\vee \text{in}'' = \text{out}'' = \langle\,\rangle \wedge \text{in}_0 = \text{out}_0 \wedge (\text{in}')_0 = (\text{out}')_0). \end{aligned}$$

In general, P_n describes the first $2n$ communications of a process that correctly copies the first n messages from the input to the output and then breaks. We therefore guess the general form

$$\begin{aligned} P_n = (&|\text{in}| + |\text{out}| < 2n \Rightarrow \\ &\text{stable} \wedge (\text{out} = \text{in} \vee \text{out} = \text{in}^\dagger)) \\ &\wedge (|\text{in}| + |\text{out}| = 2n \Rightarrow \text{out} = \text{in}). \end{aligned}$$

Finally, we draw the conclusion (which was obvious all along) that

$$\begin{aligned}\mu\xi(?x \to !x \to \xi) &= \forall n \geqslant 0 \,.\, P_n \\ &= \text{stable} \wedge (\text{out} = \text{in} \vee \text{out} = \text{in}^\dagger) \\ &= \text{COPY}.\end{aligned}$$

A simpler way to prove this identity is to show that the predicate COPY is a solution to the defining equation of the recursion, i.e.

$$\text{COPY} = (?x \to !x \to \text{COPY}).$$

The fact that this is the *weakest* solution is a consequence of the fact that it is the *only* solution. A program $P(\xi)$ is said to be *guarded* for ξ if every possible occurrence of ξ is preceded by an input or output operation. Thus

$(!x \to \xi) \vee (?y \to !x \to \xi)$ is guarded,
but $(!x \to \xi) \vee \xi$ is not guarded.

If $P(\xi)$ is guarded for ξ, then the equation

$$\xi = P(\xi)$$

has an unique solution $\mu\xi \,.\, P(\xi)$.

Chain

If P and Q are processes, we define $(P \gg Q)$ as the result of connecting the output wire of P to the input wire of Q (see figure 2). Communications along this connecting wire cannot

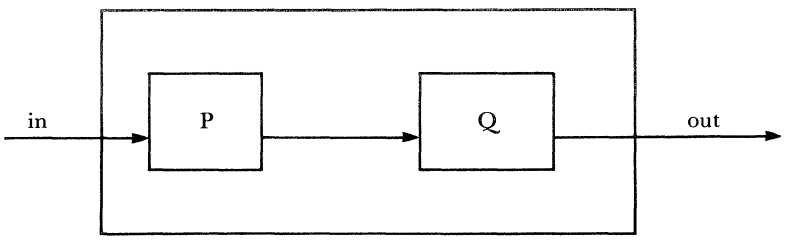

FIGURE 2. A chain.

be observed from the environment; they occur automatically whenever P is ready to output and Q is ready to input. All communication on the input wire of $(P \gg Q)$ is performed by P and all output to the environment is performed by Q. $(P \gg Q)$ is itself a process, and may be chained to other processes:

$$(P \gg Q) \gg R.$$

A simple example of a chain is formed by two instances of the COPY process connected to each other to make a double buffer:

$$\begin{aligned}(\text{COPY} \gg \text{COPY}) &= \text{stable} \wedge \exists b \,.\, b \in \{\text{in}, \text{in}^\dagger\} \wedge \text{out} \in \{b, b^\dagger\} \\ &= \text{stable} \wedge \text{out} \in \{\text{in}, \text{in}^\dagger, \text{in}^{\dagger\dagger}\} \\ &\triangleq \text{BUFF2}.\end{aligned}$$

A more spectacular example is a program that implements an unbounded buffer by chaining a recursive instance of itself:

$$\begin{aligned}&\mu\xi\,.\,(?x\rightarrow(\xi \gg (!x\rightarrow \mathrm{COPY})))\\ &\quad = \mathrm{stable}\wedge \mathrm{out}\leqslant \mathrm{in}\\ &\quad \triangleq \mathrm{BUFF}.\end{aligned}$$

A chain that does nothing but internal communication is just as broken as one that is engaged in an infinite recursion:

$$(\mu\xi\,.\,!0\rightarrow\xi) \gg (\mu\xi\,.\,?x\rightarrow\xi) = \mathrm{true}.$$

Instead of giving an explicit definition of the chaining operator as a predicate, let us list the algebraic properties we would like it to have. Clearly it should be continuous and distributive; it should also be strict, so that it breaks whenever either of its operands is broken; finally, it should obey the following four laws, which describe the expected behaviour of input and output. First, when the left operand outputs and the right operand inputs, both these actions take place simultaneously; the communication cannot be observed, but its effect is to copy the value of the expression from the outputting to the inputting process. These remarks are formalized in the law

$$(!e\rightarrow P) \gg (?x\rightarrow Q(x)) = P \gg Q(e). \tag{1}$$

If either operand of $\gg$ starts with a communication with the other, but the other starts with an external communication, then the external communication takes place first, and the other process must wait:

$$(!e\rightarrow P) \gg (!f\rightarrow Q) = !f\rightarrow((!e\rightarrow P) \gg Q), \tag{2}$$

$$(?x\rightarrow P(x)) \gg (?y\rightarrow Q(y)) = ?z\rightarrow(P(z) \gg (?y\rightarrow Q(y))), \tag{3}$$

where z is chosen not to occur in Q(y). The last of the four laws states that when *both* operands start with an external communication, then either communication may occur first, the choice being non-determinate:

$$\begin{aligned}(?x\rightarrow P(x)) \gg (!f\rightarrow Q) = {}&(?z\rightarrow(P(z) \gg (!f\rightarrow Q)))\\ &\vee (!f\rightarrow((?x\rightarrow P(x)) \gg Q))).\end{aligned} \tag{4}$$

If P and Q are finite in the sense that they contain no recursions, then the collection of laws given are complete, in the sense that $(P \gg Q)$ can be reduced to 'normal' form that does not contain $\gg$. Thus for finite processes, the meaning of the chaining operator (if it has one) is uniquely defined by these laws. The continuity condition for $\gg$ ensures that chaining is uniquely defined for processes containing recursion as well. The proof of this depends on the fact that every process can be expressed as an universal quantification of a descending chain of finite processes. This fact also permits proof of other desirable properties of chaining, for example that it is associative.

The discovery of an explicit definition of the chaining operator is not simple. A first attempt at a definition can be based on the fact that if at any time there exists some sequence b of messages that could have passed on the internal channel, then the current trace of the external channels is a possible observation of the chain. So we make a preliminary definition:

$$\begin{aligned}&P(\mathrm{in},\mathrm{out},\mathrm{stable}) \gg_0 Q(\mathrm{in},\mathrm{out},\mathrm{stable})\\ &\quad \triangleq \exists b\,.\,P(\mathrm{in},b,\mathrm{stable})\wedge Q(b,\mathrm{out},\mathrm{stable}).\end{aligned}$$

But $\gg_0$ is neither strict nor continuous, and so cannot be the right definition of $\gg$.

To ensure continuity, we need to describe the conditions under which the chain may break as a result of engaging in an infinite sequence of internal communications, a phenomenon known as 'infinite chatter':

$$\text{CHATTER} \triangleq \forall n \geqslant 0 \,.\, \exists b \,.\, |b| > n \wedge P(in, b, true) \wedge Q(b, out, true).$$

To ensure strictness, we need to identify those cases when the chain diverges as a result of divergence of just one of its operands. These cases are characterized by the fact that 'stable' is false (in fact this was the main reason why the variable 'stable' was introduced into the formal system).

$$\begin{aligned}\text{UNSTAB1} \triangleq \exists b \,.\, & P(in, b, false) \wedge Q(b, out, true) \\ & \vee\ P(in,\ b,\ true) \wedge Q(b, out, false)).\end{aligned}$$

Finally, we need to ensure that once the chain breaks it remains broken forever, i.e. it degenerates to the bottom process 'true'. To do this we introduce a modal operator ($\Diamond R$) to mean 'there was a time when R was true':

$$\begin{aligned}&\Diamond R(in, out, stable) \triangleq \\ &\quad \exists a \leqslant in \,.\, \exists b \leqslant out \,.\, R(a, b, stable).\end{aligned}$$

At last we can formulate the definition of the chaining operator

$$P \gg Q \triangleq (P \gg_0 Q \vee \Diamond \text{CHATTER} \vee \Diamond \text{UNSTAB1}).$$

That this definition has all the required algebraic properties is only a conjecture: the proof would depend on the fact that the operands of $\gg$ are not arbitrary predicates but are restricted to the notations of our simple programming language.

Assignment

Let x be a list of distinct variables, and let e be a list of the same number of expressions, and let P(x) be a program describing the behaviour of a process as a function of the initial values of x. We then define

$$(e \succ\!\!- x \rightarrow P(x)) \triangleq P(e),$$

i.e. the result of simultaneously substituting each variable in the list x by the corresponding expression in the list e, making sure that free variables of e remain free after the substitution. We assume for simplicity that all expressions of e are defined for all values of the variables they contain, so that if y is a list of distinct fresh variables

$$e \succ\!\!- x \rightarrow P(x) = (\exists y \,.\, y = e \wedge P(y)) = (\forall y \,.\, y = e \Rightarrow P(y)).$$

The predicate $e \succ\!\!- x \rightarrow P(x)$ describes the behaviour of a process that first simultaneously assigns the values of e to the variables of x and then behaves like P(x). The initial assignment is an internal action, and is therefore wholly unobservable. In more conventional programming notation this would be written

$$x := e; P(x).$$

A simple example of a program that uses assignment is one that implements a double buffer

$$\begin{aligned} &?x \to \mu\xi\,.\,((!x \to ?x \to \xi) \vee (?y \to !x \to (y \succ\!\!- x \to \xi))) \\ &\quad = \text{stable} \wedge \text{out} \in \{\text{in}, \text{in}^{\dagger}, \text{in}^{\dagger\dagger}\} \\ &\quad = \text{BUFF2}. \end{aligned}$$

Conditional

Let b be a propositional formula, i.e. a single expression that for all values of its free variables yields a result that is either true or false. Let P and Q be programs. Define

$$P \lessdot b \gtrdot Q \triangleq (b \wedge P \vee \bar{b} \wedge Q).$$

This is a process that behaves like P if b is initially true and like Q if b is initially false. The conventional programming notation for a conditional is

if b **then** P **else** Q.

The reason for the infix notation is that this permits elegant expression of algebraic properties such as idempotence, associativity and distributivity, for example

$$\begin{aligned} P \lessdot b \gtrdot (Q \lessdot b \gtrdot R) &= (P \lessdot b \gtrdot Q) \lessdot b \gtrdot R \\ &= P \lessdot b \gtrdot R. \end{aligned}$$

A complete set of algebraic laws for $\lessdot b \gtrdot$ will be given by Hoare (1985).

$$\begin{aligned} &\mu\xi\,.\,(?x \to ?y \to \\ &\quad 0, x \succ\!\!- q, r \to \\ &\quad \mu\psi\,.\,((q+1, r-y \succ\!\!- q, r \to \psi) \\ &\qquad \lessdot r \geqslant y \gtrdot (!q \to !r \to \xi))) \end{aligned}$$

FIGURE 3. Long division.

A simple example of the use of a conditional is to construct a program (see figure 3) that repeatedly inputs a pair of natural numbers and outputs the quotient and remainder of division of the first by the second. If the divisor is zero, the program breaks. The program uses the simple but slow method of successive subtraction. To emphasize the familiarity of these ideas, figure 4 gives a translation into the notations of a more conventional programming language.

Sequential composition

If P and Q are processes, their sequential composition (P;Q) is a process that behaves like P until P successfully terminates, and then it behaves like Q. If P never terminates successfully, neither does (P;Q). The process that does nothing but terminate successfully will be called 'skip'.

Let us give the algebraic laws that we would expect to govern the behaviour of sequential composition. First it must be continuous and distributive and strict in its first argument. Clearly

```
begin
ξ: input x; input y;
   q: = 0; r: = x;
ψ: if r ⩾ y then begin q: = q+1;
                       r: = r−y;
                       go to ψ
                  end
              else begin output q;
                       output r;
                       go to ξ
                  end
end
```

FIGURE 4. Conventional notation.

it should be associative and have 'skip' as its unit. Finally $(;Q)$, considered as a unary postfix operator, should distribute backward through all other operators of our language (except $\gg$):

$$(!e \to P);Q = !e \to (P;Q),$$
$$(?x \to P(x));Q = ?z \to (P(z);Q),$$
$$(e \rightarrowtail x \to P(x));Q = e \rightarrowtail z \to (P(z);Q), \quad \text{(for z not free in Q)}$$
$$(P \lhd b \rhd R);Q = (P;Q) \lhd b \rhd (R;Q).$$

As for $\gg$, we have sufficient laws to eliminate sequential composition from every finite program. The continuity property ensures that the operator is uniquely defined for all programs, provided that it exists. It is quite difficult to formulate the definition in a satisfactory fashion; for further discussion see Hehner (1984). Certainly, successful termination must be an observable event, and the final values of all variables must also be observable.

3. CONCLUSION

This paper has made the claim that a computer program can be identified with the strongest predicate describing all relevant observations that can be made of a computer executing the program. The claim is illustrated by the formal definition of the notations of a very simple programming language. The claim is justified by purely philosophical arguments. A stronger justification would be its promised practical benefits for the specification and development of reliable programs.

Before writing a program, the programmer is recommended to formulate a specification S of what his program is intended to accomplish. S is a description of the observations that are admissible for his program when it is constructed. The major problem in formulating S is to ensure the utmost simplicity and clarity, so that there can remain no doubt that it describes accurately just what is wanted; for if it does not, there is nothing that the mathematician or

the programmer can do to remedy the consequences, which may be disastrous. For this reason, there should be no restriction on the range of concepts and notations used to express the specification: the full set of logical and mathematical notations should be available for use in the overriding interests of clarity. If suitable concepts are not yet known, new branches of mathematics must be developed to meet the need.

Once the specification is formulated, the task of the programmer remains to find a predicate P, expressed in the restricted notations of his programming language, such that P logically implies the specification S, i.e.

$$\models P \Rightarrow S.$$

Because of the notational restrictions, and in the pursuit of efficiency, P will in general get much longer and more complicated than S. But in proving the correctness of P, the programmer may use all the familiar techniques and methods of classical mathematics. Consequently, he does not need the cumbersome specialised proof rules that have often been associated with proof-oriented programming language definitions (Hoare 1969). Finally, if the specification is not tautologous, the total correctness of the program will be established.

I certainly do not recommend that a large program be proved correct by expanding all the definitions and translating it explicitly into one gigantic predicate. A far more effective technique is to perform the proofs as necessary during the design and construction of the program. This is known as 'top-down programming', and is now described in five steps.

(1) Suppose the original specification is S. The programmer needs the insight to see that the achievement of S will involve completion of (say) two subtasks. He formulates the specification of these subtasks as predicates T and U.

(2) Using only the notations of his programming language he then constructs a framework $P(\xi,\psi)$, containing the names ξ and ψ to stand for the subtask programs that have not yet been written.

(3) He then slots the specifications T and U in place of these two subprograms, and proves that this satisfies the original specification S, i.e.

$$\models P(T,U) \Rightarrow S.$$

Note that P(T,U) is a predicate expressed in a mixture of conventional and programming notations.

(4) He can now safely delegate to others the subtasks of writing programs Q and R, which satisfy the specifications T and U, i.e.

$$\models Q \Rightarrow T$$

and $$\models R \Rightarrow U.$$

(5) When this is done, he can slot programs Q and R into the original framework P, and he may be sure that the result will meet the original specification S,

$$\models P(Q,R) \Rightarrow S.$$

This assurance is gained not by laborious integration testing after delivery of the components, but by a proof that has been made even before the task of writing the subprograms has started. Since the subprograms have been constructed by use of similar reliable methods, the risk of error should be quite small. And the validity of this method of programming by parts depends

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

only on the fact that all operators of our programming language are monotonic in the sense that they respect implication ordering.

If $S \Rightarrow T$
then $P(S) \Rightarrow P(T)$.

Another effective method of programming is to write first an inefficient program P that meets the specification S. This can be useful as a demonstration or training prototype of the eventual product. Then the algebraic laws can be used to transform P into a more efficient program Q, such that

$$\models Q \Rightarrow P.$$

Clearly Q will meet any specification that P meets. If P is a non-deterministic program, the transformation may use implications as well as equivalences in the pursuit of greater efficiency.

Thus the approach advocated in this paper includes that of the other contributors to this collection, in that it gives a mathematical model for the notations of a simple executable programming language and uses algebraic laws for optimization. It differs from the other contributions in making three recommendations:

(1) Specifications should *not* be confined to the notations of an executable programming language.

(2) Implication, rather than just equivalence, should be used to prove correctness of programs, and to transform them in the interests of efficiency.

(3) These methods need not be confined to applicative programming languages. They should be extended to conventional procedural languages, which can be efficiently executed on computers of the present day.

I am grateful to: A. J. R. G. Milner (1980) for his pioneering work in the mathematical theory of communicating systems; E. C. R. Hehner (1983) for pointing out that programs are predicates; D. S. Scott (1981) for the domain theory that underlies a proper theory of recursion; S. D. Brookes and A. W. Roscoe (1984) and E.-R. Olderog (1984) for construction of the model on which this exposition is based; E. W. Dijkstra (1976) for his realization of the value of non-determinacy, and his insistence on total correctness.

References

Brookes, S. D., Hoare, C. A. R., Roscoe, A. W. 1984 A theory of communicating sequential processes. *J. Ass. comput. Mach.* (In the press.)

Dijkstra, E. W. 1976 *A discipline of programming*, p. 217. Englewood Cliffs, N.J.: Prentice-Hall.

Floyd, R. W. 1967 Assigning meanings to programs. *Proc. Am. math. Soc. Symp. Appl. Math.*, vol. 19, pp. 19–31.

Hehner, E. C. R. 1984 Predicative Programming, part I. *Comm. Ass. comput. Mach.* (In the press.)

Hehner, E. C. R. & Hoare, C. A. R. 1983 A more complete model of communicating processes. *Theor. computer Sci.* **26**, 105–120.

Hoare, C. A. R. 1969 An axiomatic basis for computer programming. *Commun. Ass. comput. Mach.* **12**, 576–580, 583.

Hoare, C. A. R. 1985 A couple of novelties in the propositional calculus. *Z. math. Logik* (In the press.)

Milner, A. J. R. G. 1980 A calculus of communicating systems. *Springer LNCS* vol. 92. Berlin: Springer-Verlag.

Olderog, E.-R. & Hoare, C. A. R. 1984 *Specification-oriented semantics for communicating processes, P.R.G.-37*. Oxford University Computing Laboratory.

Scott, D. S. 1981 *Lecture notes on a mathematical theory of computation, P.R.G.-19*, p. 148. Oxford University Computing Laboratory.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Discussion

F. K. Hanna (*Electronics Laboratories, University of Kent at Canterbury, U.K.*). Professor Hoare made a valuable point when he noted that a physical system can be characterized by a predicate that describes all possible observations that may be made at the ports of the system. In fact, not only can this be done, but it can very usefully be done.

At Kent University, we have been working for some years on characterizing digital systems in just this way. One may imagine waveforms in digital systems as being described as partial functions, from time to (typically) a two-element set (conventionally called T and F). One can then write down a predicate (on 3-tuples of waveforms) that characterizes (but not overspecifies) the notion of, for instance, 'behaving like an AND gate'. An AND gate is then, by definition, any device, the waveforms at whose ports satisfy this predicate: likewise with other primitive elements.

Complex digital systems are realized by an interconnected network of primitive elements. The predicate satisfied by the complex system may be related to the predicates satisfied by its component parts by the use of combinators. For instance, the behaviour of proposed implementation of a serial binary adder may be expressed in terms of the behaviour of the primitive gates from which it is constructed, and the combinator representing the constraints on the behaviour of these gates imposed by the 'circuit diagram'. Working within a suitable theory, one may then seek to prove, as a theorem, that this behaviour does in fact correspond to doing binary addition.

The ease with which behavioural predicates may be used to characterize, as weakly or as strongly as desired, the behaviour of systems, irrespective of whether they are implemented in either software or hardware, is today an especially valuable one.

C. A. R. Hoare. Thank you for your supportive comment. The advantages of predicate-oriented specifications and proof of digital systems are especially marked. For small components (for example a single chip) the actions of the components are lock-step synchronized, so there is no problem in hiding infinite sequences of internal events; consequently parallel composition can be simply defined as logical conjunction. This approach is currently being pursued by Mike Gordon at Cambridge.

Unfortunately, there remains the problem of a miracle. A device that simply shortcircuits power to ground is represented by the predicate

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

and by propositional calculus, this implies every specification. So one must take care that a device that shortcircuits will degenerate to 'true' instead of 'false'.

Alternatively, one could admit that we are proving only partial correctness (i.e. conditional upon absence of shortcircuit). Yet another alternative is to accept the obligation to prove a circuit *equivalent* to its specification. Some practical experience is needed to judge between these alternatives.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES