PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

PROGRAMS ARE PREDICATES

C.A.R. Hoare

Programming Research Group,
Oxford University Computing Laboratory,
11 Keble Road, Oxford, OX1 3QD, England.

Abstract

Requirements to be met by a new engineering product can be captured most directly by a logical predicate describing all its desired and permitted behaviours. The behaviour of a complex product can be described as the logical composition of predicates describing the behaviour of its simpler components. If the composition logically implies the original requirement, then the design will meet its specification. This implication can be mathematically proved before starting the implementation of the components. The same method can be repeated on the design of the components, until they are small enough to be directly implementable.

A programming language can be defined as a restricted subset of predicate notation, ensuring that the described behaviour may be efficiently realised by a combination of computer software and hardware. The restrictive notations give rise to a specialised mathematical theory, which is expressed as a collection of algebraic laws useful in the transformation and optimisation of designs. Non-determinism contributes both to reusability of design and to efficiency of implementation.

This philosophy is illustrated by application to hardware design, to procedural programs and to PROLOG. It is shown that the procedural reading of logic programs as predicates is different from the declarative reading, but just as logical.

1 Inspiration

It is a great honour for me to address this conference which celebrates the completion of the Fifth Generation Computer Systems project in Tokyo. I add my own congratulations to those of your many admirers and followers for the great advances and many achievements made by those who worked on the project. The project started with ambitious and noble goals, aiming not only at radical advances in Computer Technology, but also at the direction of that technology to the wider use and benefit of mankind. Many challenges remain; but the goal is one that inspires the best work of scientists and engineers throughout the ages.

For my part, I have been most inspired by the philosophy with which this project approaches the daunting task of writing programs for the new generation of computers and their users. I have long shared the view that the programming task should always begin with a clear and simple statement of requirements and objectives, which can be formalised as a specification of the purposes which the program is required to meet. Such specifications are predicates, with variables standing for values of direct or indirect observations that can be made of the behaviour of the program, including both questions and answers, input and output, stimulus and response. A predicate describes, in a neutral symmetric fashion, all permitted values which those variables may take when the program is executed. The over-riding requirement on a specification is clarity, achieved by a notation of the highest possible modularity and expressive power. If a specification does not *obviously* describe what is wanted, there is a grave danger that it describes what is *not* wanted; it can be difficult, expensive, and anyway mathematically impossible to check against this risk.

A minimum requirement on a specification language is that it should include in full generality the elementary connectives of Boolean Algebra: conjunction, disjunction, and negation — simple *and*, *or*, and *not*. Conjunction is needed to connect requirements, both of which must be met, for example,

- it must control pressure *and* temperature.

Disjunction is needed to allow tolerances in implementation

- it may deviate from optimum by one *or* two degrees.

And negation is needed for even more important reasons

- it must *not* explode!

As a consequence, it is possible to write a specification like

$$P \vee \neg P$$

which is always true, and so describes every possible observation of every possible product. Such a tolerant

specification is easy to satisfy, even by a program that gets into an infinite loop. In fact, such infinite failure will be treated as so serious that the tautologously true specification is the only one that it satisfies.

Another inspiring insight which I share with the Fifth Generation project is that programs too are predicates. When given an appropriate reading, a program describes all possible observations of its behaviour under execution, all possible answers that it can give to any possible question. This insight is one of the most convincing justifications for the selection of logic programming as the basic paradigm for the Fifth Generation project. But I believe that the insight is much more general, and can be applied to programs expressed in other languages, and indeed to engineering products described in any meaningful design notation whatsoever. It gives rise to a general philosophy of engineering, which I shall illustrate briefly in this talk by application to hardware design, to conventional sequential programs, and even to the procedural interpretation of PROLOG programs.

But it would be wholly invalid to claim that all predicates can be read as programs. Consider a simple but dramatic counter-example, the contradictory predicate

$$P \ \& \ \neg P$$

which is always false. No computer program (or anything else) can ever produce an answer which has a property P as well as its negation. So this predicate is not a program, and no processor could translate it into one which gives an answer with this self-contradictory property. Any theory which ascribes to an implementable program a behaviour which is known to be unimplementable must itself be incorrect.

A programming language can therefore be identified with only a subset of the predicates of predicate calculus; each predicate in this subset is a precise description of all possible behaviours of some program expressible in the language. The subset is designed to exclude contradictions and all other unimplementable predicates; and the notations of the language are carefully restricted to maintain this exclusion. For example, predicates in PROLOG are restricted to those which are definable by Horn clauses; and in conventional languages, the restrictions are even more severe. In principle, these gross restrictions in expressive power make a programming language less suitable as a notation for describing requirements in a modular fashion at an appropriately high level of abstraction.

The gap between a specification language and a programming language is one that must be bridged by the skill of the programmer. Given specification S, the task is to find a program P which satisfies it, in the sense that every possible observation of every possible behaviour of the program P will be among the behaviours described by (and therefore permitted by) the specification S. In logic, this can be assured with mathematical certainty by a proof of the simple implication

$$\vdash P \Rightarrow S.$$

A simple explanation of what it means for a program to meet its specification is one of the main reasons for interpreting both programs and specifications within the predicate calculus.

Now we can explain the necessity of excluding the contradictory predicate *false* from a programming notation. It is a theorem of elementary logic that

$$\vdash false \Rightarrow S,$$

so *false* enjoys the miraculous property of satisfying every specification whatsoever. Such miracles do not exist; which is fortunate, because if they did we would never need anything else, certainly not programs nor programming languages nor computers nor fifth generation computer projects.

2 Examples

A very simple example of this philosophy is taken from the realm of procedural programming. Here the most important observable values are those which are observed before the program starts and those which are observed after the program is finished. Let us use the variable x to denote the initial value and let x' be the final value of an integer variable, the only one that need concern us now. Let the specification say that the value of the variable must be increased

$$S = (x' > x)$$

Let the program add one to x

$$P = (x := x + 1)$$

The behavioural reading of this program as a predicate describing its effect is

$$P = (x' = x + 1)$$

i.e., the final value of x is one more than its initial value.

Every observation of the behaviour of P in any possible initial state x will satisfy this predicate. Consequently the Validity of the implication

$$\vdash P \Rightarrow S$$

i.e., $$\vdash x' = x + 1 \Rightarrow x' > x$$

will ensure that P correctly meets its specification. So does the program

$$x := x + 7,$$

but not

$$x := 2 \times x.$$

To illustrate the generality of my philosophy, my next examples will be drawn from the design of combinational hardware circuits. These can also be interpreted as predicates. A conventional *and-gate* with two input wires named a and b and a single output wire named x is described by a simple equation

$$x = a \wedge b.$$

The values of the three free variables are observed as voltages on the named wires at the end of a particular cycle of operation. At that time, the voltage on the output wire x is the lesser of the voltages on the input wires a and b. Similarly, an *or-gate* can be described by a different predicate with different wires

$$d = y \vee c,$$

i.e., the voltage on d is the greater of those on y and c. A simple wire is a device that maintains the same voltage at each of its ends, for example

$$x = y.$$

Now consider an assembly of two components operating in parallel, for example the and-gate together with the or-gate. The two predicates describing the two components have no variables in common; this reflects the fact that there is absolutely no connection between them. Consequently, their simultaneous joint behaviour consists solely of their two independent behaviours, and is correctly described by just the conjunction of the predicates describing their separate behaviours

$$(x = a \wedge b) \ \& \ (d = y \vee c)$$

This simple example is a convincing illustration of the principle that parallel composition of components is nothing but conjunction of their predicates, at least in the case when there is no possibility of interaction between them.

The principle often remains valid when the components are connected by variables which they share. For example, the wire which connects x with y can be added to the circuit, giving a triple conjunction

$$(x = a \wedge b) \ \& \ (x = y) \ \& \ (d = (y \vee c)).$$

This still accurately describes the behaviour of the whole assembly. The predicate is mathematically equivalent to

$$(d = (a \wedge b) \vee c) \ \& \ (x = y = (a \wedge b)).$$

When components are connected together in this way by the sharing of variable names (x and y), the values of the shared variables are usually of no concern or interest to the user of the product, and even the option of observing them is removed by enclosure, as it were, in a black box. The variables therefore need to be hidden or removed or abstracted from the predicate describing the observable behaviour of the assembly; and the standard way of eliminating free variables in the predicate calculus is by quantification.

In the case of engineering designs, existential quantification is the right choice. It is necessary that there exist an observable value for the hidden variable; but no one cares exactly what value it is. A formal justification is as follows. Let S be the specification for the program P, and let x be the variable to be hidden in P. Clearly, one could never wish to hide a variable which is mentioned in the specification, so clearly x will not occur free in S. Now the designer's original proof obligation without hiding is

$$\vdash P \Rightarrow S;$$

and the proof obligation after hiding is

$$\vdash (\exists x.P) \Rightarrow S.$$

By the predicate calculus, since x does not occur in S, these two proof obligations are the same.

But often quantification simplifies, as in our hardware example, where the formula

$$\exists x, y. \quad x = a \wedge b \ \& \ y = x \ \& \ d = y \vee c,$$

reduces to just

$$d = (a \wedge b) \vee c.$$

This mentions only the visible external wires of the circuit, and probably expresses the intended specification of the little assembly.

Unfortunately, not all conjunctions of predicates lead to implementable designs. Consider for example the conjunction of a negation circuit ($y = \neg x$) with the wire ($y = x$), connecting its output back to its input. In practice, this assembly leads to something like an electrical short circuit, which is completely useless — or even worse than useless, because it will prevent proper operation of any other circuit in its vicinity. So there is no specification (other than the trivial specification *true*) which a short-circuited design can reasonably satisfy. But in our oversimplified theory, the predicted effect is exactly the opposite. The predicate describing the behaviour of the circuit is a self-contradiction, equivalent to *false*, which is necessarily unimplementable.

One common solution to the problem is to place careful restrictions on the ways in which components can be combined in parallel by conjunction. For example, in combinational circuit design, it is usual to make a rigid distinction between input wires (like a or c) and output wires (like x or d). When two circuits are combined, the output wires of the first of them are allowed to be connected to the input wires of the second, but never

the other way round. This restriction is the very one that turns a parallel composition into one of its least interesting special cases, namely sequential composition. This means that the computation of the outputs of the second component has to be delayed until completion of the computation of the outputs of the first component.

Another solution is to introduce sufficient new values and variables into the theory to ensure that one can describe all possible ways in which an actual product or assembly can go wrong. In the example of circuits, this requires at least a three-valued logic: in addition to high voltage and low voltage, we introduce an additional value (written $\perp$, and pronounced "bottom"), which is observed on a wire that is connected simultaneously both to high voltage and to low voltage, i.e., a short circuit. We define the result of any operation on $\perp$ to give the answer $\perp$. Now we can solve the problem of the circuit with feedback, specified by the conjunction

$$x = \neg y \ \ \& \ \ y = x$$

In three-valued logic, this is no longer a falsehood: in fact it correctly implies that both the wires x and y are short circuited

$$x = y = \perp.$$

The moral of this example is that predicates describing the behaviour of a design must also be capable of describing all the ways in which the design may go wrong. It is only a theory which correctly models the possibility of error that can offer any assistance in avoiding it.

If parallelism is conjunction of predicates, disjunction is equally simply explained as introducing non-determinism into specifications, designs and implementations. If P and Q are predicates, their disjunction $(P \vee Q)$ describes a product that may behave as P or as Q, but does not determine which it shall be. Consequently, you cannot control or predict the result. If you want $(P \vee Q)$ to satisfy a specification S, it is necessary (and sufficient) to prove both that P satisfies S and that Q satisfies S. This is exactly the defining principle of disjunction in the predicate calculus: it is the least upper bound of the implication ordering. This single principle encapsulates all you will ever need to know about the traditionally vexatious topic of non-determinism. For example, it follows from this principle that non-deterministic specifications are in general easier to implement, because they offer a range of options; but non-deterministic implementations are more difficult to use, because they meet only weaker specifications.

Apart from conjunction (which can under certain restrictions be implemented by parallelism), and disjunction (which permits non-deterministic implementation), the remaining important operator of the predicate calculus is negation. What does that correspond to in programming? The answer is: nothing! Arguments about computability show that it can never be implemented, because the complement of a recursively enumerable set is not in general recursively enumerable. A commonsense argument is equally persuasive. It would certainly be nice and easy to write a program that causes an explosion in the process which it is supposed to control. It would be nice to get a computer to execute the negation of this program, and so ensure that the explosion never occurs. Unfortunately and obviously this is impossible. Negation is obviously the right way to *specify* the absence of explosion, but it cannot be used in *implementation*. That is one of the main reasons why implementation is in principle more difficult than specification. Of course, negation can be used in certain parts of programs, for example, in Boolean expressions: but it can never be used to negate the program as a whole. We will see later that PROLOG negation is very different from the kind of Boolean negation used in specifications.

The most important feature of a programming language is recursion. It is only recursion (or iteration, which is a special case) that permits a program to be shorter than its execution trace. The behaviour of a program defined recursively can most simply be described by using recursion in the definition of the corresponding predicate. Let $P(X)$ be some predicate containing occurrences of a predicate variable X. Then X can be defined recursively by an equation stating that X is a fixed point of P

$$X \stackrel{\text{def}}{=} P(X).$$

But this definition is meaningful only if the equation has a solution; this is guaranteed by the famous Tarski theorem, provided that $P(X)$ is a monotonic function of the predicate variable X. Fortunately, this fact is guaranteed in any programming language which avoids non-monotonic operators like negation. If there is more than one solution to the defining equation, we need to specify which one we want; and the answer is that we want the weakest solution, the one that is easiest to implement. (Technically, I have assumed that the predicate calculus is a complete lattice: to achieve this I need to embed it into set theory in the obvious way .)

The most characteristic feature of computer programs in almost any language is sequential composition. If P and Q are programs, the notation (P, Q) stands for a program which starts like P; but when P terminates, it applies Q to the results produced by P. In a conventional programming language, this is easily defined in predicate notation as relational composition, using conjunction followed by hiding in exactly the same way as our earlier combinational circuit. Let x stand for an observation of the initial state of all variables of a program, and let x' stand for the final state. Either or both of these may take the special value $\perp$, standing for non-termination or infinite failure, which is one of the worst ways in which a program can go wrong. Each program is a predicate $P(x, x')$ or $Q(x, x')$, describing a relation

between the initial state x and the final state x'. For example, there is an identity program $\mathbb{I}$ (a null operation), which terminates without making any change to its initial state. But it can do this only if it starts in a proper state, which is not already failed

$$\mathbb{I} \stackrel{\text{def}}{=} (x \neq \bot \Rightarrow x' = x).$$

Sequential composition of P and Q in a conventional language means that the initial state of Q is the same as the final state produced by P; however the value of this intermediate state passed from P to Q is hidden by existential quantification, so that the only remaining observable variables are the initial state of P and the final state of Q. More formally, the composition (P, Q) is a predicate with two free variables (x and x') which is defined in terms of P and Q, each of which are also predicates with two free variables

$$(P, Q)(x, x') \stackrel{\text{def}}{=} \exists y.\ \ P(x, y)\ \ \&\ \ Q(y, x').$$

Care must be taken in the definition of the programming language to ensure that sequential composition never becomes self-contradictory. A sufficient condition to achieve this is that when either x or x' take the failure value $\bot$, then the behaviour of the program is entirely unpredictable: anything whatsoever may happen. The condition may be formalised by the statement that for all predicates P which represent a program

$$\forall x'.P(\bot, x')$$

and

$$\forall x.\ \ P(x, \bot) \Rightarrow \forall x'.P(x, x').$$

The imposition of this condition does complicate the theory, and it requires the theorist to prove that all programs expressible in the notations of the programming language will satisfy it. For example, the null operation $\mathbb{I}$ satisfies it; and for any two predicates P and Q which satisfy the condition, so does their sequential composition (P, Q), and their disjunction $P \vee Q$, and even their conjunction $(P \wedge Q)$, provided that they have no variables in common. As a consequence any program written only in these restricted notations will always satisfy the required conditions. Such programs can therefore never be equivalent to *false*, which certainly does *not* satisfy these conditions.

The only reason for undertaking all this work is to enable us to reason correctly about the properties of programs and the languages in which they are written. The simplest method of reasoning is by symbolic calculation using algebraic equations which have been proved correct in the theory. For example, to compose the null operation $\mathbb{I}$ before or after a program P does not change P. Algebraically this is expressed in a law stating that $\mathbb{I}$ is the unit of sequential composition

$$(P, \mathbb{I}) = P = (\mathbb{I}, P).$$

Also, composition is associative; to follow the pair of operations (P, Q) by R is the same as following P by the pair of operations (Q, R)

$$((P, Q), R) = (P, (Q, R)).$$

3 PROLOG

In its procedural reading, a PROLOG program also has an initial state and a result; and its behaviour can be described by a predicate defining the relation between these two. Of course this is quite different from the predicate associated with the logical reading. It will be more complicated and perhaps less attractive; but it will have the advantage of accurately describing the behaviour of a computer executing the program, while retaining the possibility of reasoning logically about its consequences.

The initial state of a PROLOG program is a substitution, which allocates to each relevant variable a symbolic expression standing for the most general form of value which that variable is known to take. Such a substitution is generally called θ. The result θ' of a PROLOG program differs from that of a conventional language. It is not a single substitution, but rather a *sequence* of answer substitutions, which may be delivered one after the other on request. For example, the familiar PROLOG program

$$\text{append}\ (X, Y, Z)$$

may be started in the state

$$Z = [1, 2].$$

It will then produce on demand a sequence of three answer states

$$\begin{array}{lllll} X & = & [\ ], & Y & = & [1,2] \\ X & = & [1], & Y & = & [2] \\ X & = & [1,2], & Y & = & [\ ]. \end{array}$$

Infinite failure is modelled as before by the special state $\bot$; when it occurs, it is always the last answer in the sequence. Finite failure is represented by the empty sequence [ ]; and the program NO is defined as one that always fails in this way

$$NO(\theta, \theta') \stackrel{\text{def}}{=} (\theta \neq \bot \Rightarrow \theta' = [\ ]).$$

The program that gives an affirmative answer is the program YES; but the answer it gives is no more than what is known already, packaged as a sequence with only one element

$$YES(\theta, \theta') \stackrel{\text{def}}{=} (\theta = \bot \Rightarrow \theta' = [\theta]).$$

A *guard* in PROLOG is a Boolean condition b applied to the initial state θ to give the answer YES or NO

$$b(\theta,\theta') \stackrel{\text{def}}{=} \theta' = [\theta] \;\&\; (b\theta) \\ \vee\; \theta' = [\,] \;\&\; (\neg b\theta).$$

Examples of such conditions are VAR and NONVAR.

The effect of the PROLOG $or(P;Q)$ is obtained by just appending the sequence of answers provided by the second operand Q to the sequence provided by the first operand P; and each operand starts in the same initial state

$$(P;Q)(\theta,\theta') \stackrel{\text{def}}{=} \exists X,Y.\;\; P(\theta,X) \;\&\; Q(\theta,Y) \\ \&\; \text{append}(X,Y,\theta).$$

The definition of *append* is the same as usual, except for an additional clause which makes the result of infinite failure unpredictable

$$\text{append}([\bot],Y,Z) \\ \text{append}([\,],Y,Y) \\ \text{append}([X|Xs],Y,[X|Zs]) \\ \quad :-\; \text{append}\;(Xs,Y,Zs).$$

In all good mathematical theories, every definition should be followed by a collection of theorems, describing useful properties of the newly defined concept. Since NO gives no answer, its addition to a list of answers supply by P can make no difference, so NO is the unit of PROLOG semicolon

$$NO;P = P = P;NO.$$

Similarly, the associative property of appending lifts to the composition of programs

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

The PROLOG conjunction is very similar to sequential composition, modified systematically to deal with a sequence of results instead of a single one. Each result of the sequence X produced by the first argument P is taken as an initial state for an activation of the second argument Q; and all the sequences produced by Q are concatenated together to give the overall result of the composition

$$(P,Q)(\theta,\theta') \stackrel{\text{def}}{=} \exists X,Y. \quad P(\theta,X) \\ \&\quad \text{each}\;(X,Y) \\ \&\quad \text{concat}\;(Y,\theta')$$

where

$$\text{each}([\,],[\,]) \\ \text{each}([X|Xs],[Y|Ys]) \\ \quad :-\; Q(X,Y) \;\&\; \text{each}\;(Xs,Ys)$$

and

$$\text{concat}([\,],[\,]) \\ \text{concat}([X|Xs],Z) \\ \quad :-\; \text{append}\;(X,Y,Z) \;\&\; \text{concat}(Xs,Y)$$

The idea is much simpler than its formal definition; its simplicity is revealed by the algebraic laws which can be derived from it. Like composition in a conventional language, it is associative and has a unit YES

$$P,(Q,R) = (P,Q),R$$

$$(YES,P) = P = (P,YES).$$

But if the first argument fails finitely, so does its composition with anything else

$$(NO,P) = NO.$$

However (P,NO) is unequal to NO, because P may fail infinitely; the converse law therefore has to be weakened to an implication

$$NO \Rightarrow (P,NO).$$

Finally, sequential composition distributes leftward through PROLOG disjunction

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

But the complementary law of rightward distribution certainly does not hold. For example, let P always produce answer 1 and let Q always produce answer 2. When R produces many answers, $(R,(P;Q))$ produces answers

$$1,2,1,2,\ldots$$

whereas $(R,P);(R,Q)$ produces

$$1,1,1,\ldots,2,2,2\ldots.$$

Many of our algebraic laws describe the ways in which PROLOG disjunction and conjunction are similar to their logical reading in a Boolean algebra; and the absence of expected laws also shows clearly where the traditional logical reading diverges from the procedural one. It is the logical properties of the procedural reading that we are exploring now.

The acid test of our procedural semantics for PROLOG is its ability to deal with the non-logical features like the cut (!), which I will treat in a slightly simplified form. A program that has been cut can produce at most one result, namely the first result that it would have produced anyway

$$P!(\theta,\theta') \stackrel{\text{def}}{=} \exists X.\;\; P(\theta,X) \;\&\; \text{trunc}\;(X,\theta').$$

The truncation operation preserves both infinite and finite failure; and otherwise selects the first element of a sequence

$$\begin{aligned}&\text{trunc}([\bot], Y)\\&\text{trunc}([\ ],[\ ])\\&\text{trunc}([X|Xs],[X]).\end{aligned}$$

A program that already produces at most one result is unchanged when cut again

$$P!! = P!$$

If only one result is wanted from a composite program, then in many cases only one result is needed from its components

$$(P;Q)! = (P!;Q!)!$$
$$(P,Q)! = (P,Q!)!$$

Finally, YES and NO are unaffected by cutting

$$YES! = YES, \quad NO! = NO.$$

PROLOG negation is no more problematic than the cut. It turns a negative answer into a positive one, a non-negative answer into a negative one, and preserves infinite failure

$$\sim P(\theta,\theta') \stackrel{\text{def}}{=} \exists Y.\ P(\theta,Y)\ \&\ \text{neg}\ (Y,\theta')$$

where

$$\begin{aligned}&\text{neg}([\bot], Z)\\&\text{neg}([\ ],[\theta])\\&\text{neg}([X|Xs],[\ ]).\end{aligned}$$

The laws governing PROLOG negation of truth values are the same as those for Boolean negation

$$\sim YES = NO \ \text{ and } \ \sim NO = YES.$$

The classical law of double negation has to be weakened to intuitionistic triple negation

$$\sim\sim\sim P \ = \ \sim P.$$

Since a negated program gives at most one answer, cutting it makes no difference

$$\sim P = \ \sim(P!) = (\sim P)!$$

Finally, there is an astonishing analogue of one of the familiar laws of de Morgan

$$\sim(P;Q) = (\sim P, \sim Q).$$

The right hand side is obviously much more efficient to compute, so this law could be very effective in optimisation. The dual law, however, does not hold.

A striking difference between PROLOG negation and Boolean negation is expressed in the law that the negation of an infinitely failing program also leads to infinite failure

$$\sim \text{true} \ = \ \text{true}.$$

This states that *true* is a fixed point of negation; since it is the weakest of all predicates, there can be no fixed point weaker than it

$$(\mu X.\sim X) \ = \ \text{true}.$$

This correctly predicts that a program which just calls its own negation recursively will fail to terminate.

That concludes my simple account of the basic structures of PROLOG. They are all deterministic in the sense that (in the absence of infinite failure) for any given initial substitution θ, there is exactly one answer sequence θ' that can be produced by the program. But the great advantage of reading programs as predicates is the simple way in which non-determinism can be introduced. For example, many researchers have proposed to improve the sequential *or* of PROLOG. One improvement is to make it commute like true disjunction, and another is to allow parallel execution of both operands, with arbitrary interleaving of their two results. These two advantages can be achieved by the definition

$$(P\|Q)(\theta,\theta') \stackrel{\text{def}}{=} \exists X, Y.\ \ P(\theta,X)\ \&\ Q(\theta,Y) \ \&\ \text{inter}\ (X,Y,\theta')$$

where the definition of interleaving is tedious but routine

$$\begin{aligned}&\text{inter}([\bot],Y,Z) \qquad\qquad \text{inter}(X,[\bot],Z)\\&\text{inter}([\ ],Y,Y) \qquad\qquad \text{inter}(X,[\ ],X)\\&\text{inter}([X|Xs],Y,[X|Z]) \ :-\ \text{inter}(Xs,Y,Z)\\&\text{inter}(X,[Y|Ys],[Y|Z]) \ :-\ \text{inter}(X,Ys,Z).\end{aligned}$$

Because appending is just a special case of interleaving, we know

$$\text{append}(X,Y,Z) \Rightarrow \text{inter}\ (X,Y,Z).$$

Consequently, sequential *or* is just a special case of parallel *or*, and is always a valid implementation of it

$$(P;Q) \Rightarrow (P\|Q).$$

The left hand side of the implication is more deterministic than the right; it is easier to predict and to control; it meets every specification which the right hand side also meets, and maybe more. In short, sequential *or* is in all ways and in all circumstances better than the parallel *or* — in all ways except one: it may be slower to implement on a parallel machine. In principle non-determinism is demonic; it never makes programming easier, and its only possible advantage is an increase in performance. However, in many cases (including this one) non-determinism also simplifies specifications and

designs, and facilitates reasoning about them at higher levels of abstraction.

My final example is yet another kind of disjunction, one that is characteristic of a commit operation in a constraint language. The answers given are those of exactly one of the two alternatives, the selection being usually non-deterministic: the only exception is in the case when one of the operands fails finitely, in which case the other one is selected. So the only case when the answer is empty is when both operands give an empty answer

$$\begin{aligned}(P \| Q) \stackrel{\text{def}}{=} \; & ((\theta' = [\,]) \;\&\; P \;\&\; Q) \\ & \vee ((\theta' \neq [\,]) \;\&\; (P \vee Q)) \\ & \vee P(\theta, \bot) \vee Q(\theta, \bot).\end{aligned}$$

(The last two clauses are needed to satisfy the special conditions described earlier). The definition is almost identical to that of the alternative command in Communicating Sequential Processes, from which I have taken the notation. It permits an implementation which starts executing both P and Q in parallel, and selects the one which first comes up with an answer. If the first elements of P and Q are guards, this gives the effect of flat Guarded Horn Clauses.

4 Conclusion

In all branches of applied mathematics and engineering, solutions have to be expressed in notations more restricted than those in which the original problems were formulated, and those in which the solutions are calculated or proved correct. Indeed, that is the very nature of the problem of solving problems. For example, if the problem is

- Find the GCD of 3 and 4

a perfectly correct answer is the trivially easy one

- the GCD of 3 and 4;

but this does not satisfy the implicit requirement that the answer be expressed in a much more restricted notation, namely that of numerals.

The proponents of PROLOG have found an extremely ingenious technique to smooth (or maybe obscure) the sharpness of the distinction between notations used for specification and those used for implementation. They actually use the same PROLOG notation for both purposes, by simply giving it two different meanings: a declarative meaning for purposes of specification, and a procedural meaning for purposes of execution. In the case of each particular program the programmer's task is to ensure that these two readings are consistent. Perhaps my investigation of the logical properties of the procedural reading will assist in this task, or at least explain why it is such a difficult one.

Clearly, the task would be simpler in a language in which the logical and procedural readings are even closer than they are in PROLOG. This ideal has inspired many excellent proposals in the development of logic and constraint languages. The symmetric parallel version of disjunction is a good example. A successful result of this research is still an engineering compromise between the expressive power needed for simple and perspicuous specification, and operational orientation towards the technology needed for cost-effective implementation.

Such a compromise will (I hope) be acceptable and useful, as PROLOG already is, in a wide range of circumstances and applications. In the remaining cases, I would like to maintain as far as possible the inspiration of the Fifth Generation Computing project, and the benefits of a logical approach to programming. To achieve this, I would give greater freedom of expression to those engaged in formalisation of the specification of requirements, and greater freedom of choice to those engaged in the design of efficiently implementable programming languages. This can be achieved only by recognition of the essential dichotomy of the languages used for these two purposes. The dichotomy can be resolved by embedding both languages in the same mathematical theory, and using logical implication to establish correctness.

But what I have described is only the beginning,—nothing more than a vague pointer to a whole new direction and method of research into programming languages and programming methodology. If any of my audience is looking for a challenge to inspire the next ten years of research, may I suggest this one? If you respond to the challenge, the programming languages of the future will not only permit efficient parallel and even non-deterministic implementations; they will also help the analyst more simply to capture and formalise the requirements of clients and customers; and then help the programmer by systematic design methods to exercise inventive skills in meeting those requirements with high reliability and low cost. I hope I have explained to all of you why I think this is important and exciting. Thank you again for this opportunity to do so.

Acknowledgements

I am grateful to Mike Spivey, He Jifeng, Robin Milner, John Lloyd, and Alan Bundy for assistance in preparation of this address.