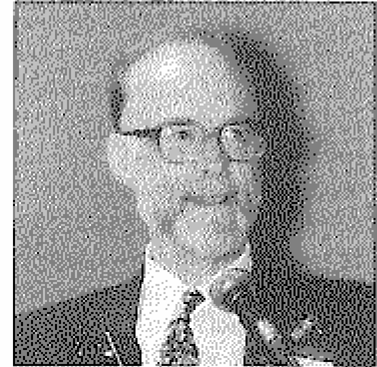
Invited Lecture

Programs are Predicates

C.A.R.Hoare
Professor
Oxford University



Tanaka: Good afternoon, ladies and gentlemen. From now we would like to have an invited lecture given by Professor Charles Antony Richard Hoare from Oxford University. Today's title of the speech is, "Programs are Predicates". It is my great honor to introduce Professor Hoare.

Professor Hoare has a degree from Oxford in 1956 in classical languages, Latin and Greek. He has always taken an interest in program languages. His first job was with a small computer manufacturer and his first task was an implementation of ALGOL 60. After eight years in industry he moved as a professor to the Queen's University, Belfast. There, he pursued his research interest in the logical foundations of programming and programming notations. In 1977 he moved to Oxford University and began to transfer some of ther results of the basic research into practical and profitable application by industry. He has made a lot of contribution to conventional programming, object-oriented programming, and parallel programming. Occam and the Transputer are now quite famous.

Recently, he has turned his attention to logic programming. His research goal is to discover theories which aid in the specification and design of hardware and software. He aims to achieve a level of abstraction to cover wide range of programming paradigms and products constructed from a mixture of technologies. Such a theory would help to avoid expensive errors occurring at system interfaces. It would also clarify the structure of computer science as an academic discipline. Today, he will talk about an inspiration he derived ten years ago at the initiation of the Fifth Generation Computer Project. The title is, "Programs are Predicates". Professor Hoare, please.

Hoare: Thank you very much. It is a great honor for me to address at this conference which celebrates the termination of the Fifth Generation Computer Systems Project. I can add my own congratulations to those of your many admirers and followers for the great advances and achievements that you have made.

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 for the use and benefit of man. Many challenges remain, but the goals are ones that have inspired the best efforts of scientists and engineers throughout the ages.

I admire the foresight of the Japanese Ministry of Trade and Industry which has recognized the need for a ten-year funding period to allow radically new ideas to emerge and to achieve maturity. I admire the courage and dedication of the many who led and worked on this project, many of you are here today. You have made both theoretical and practical advances in computing science. And even more important, you have discovered and taken advantage of connections between the

theory and the practice of the subject.

From the beginning, you have concentrated your efforts on two of the most important problems of practical computing science. They are the efficient use of highly parallel processing technology and the advanced application of highly symbolic data processing. You took your inspiration from the logical properties of logic programming and of the language Prolog. You took a view that only a logical approach could solve the daunting task of writing programs for the new generation of computers and their users.

I have been inspired by the same philosophy. I have long shared the view that the programming task should always begin with a clear and simple statement of the users' requirements and objectives, which can be formalized as a specification of the purposes which the program is desired to meet. Such specifications are predicates which describe all permissible observations of the behavior of the program that you want.

Predicates

The free variables of the predicate stand for observations that can be made of the behavior of that program under execution; the behavior, under all circumstances. A predicate describes all permitted values which these variables may take when the program is executed. And the overriding requirement on a specification is clarity, which can be achieved by a notation of the highest possible abstraction, modularity and expressive power. If the specification does not obviously describe what is wanted, there is a great danger that it describes what is not wanted; and it can be difficult, expensive, and any way 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 logic, so that specifications can be connected by conjuction, disjunction, and negation: simple 'AND', 'OR', and 'NOT'. Conjunction is needed to connect requirements both of which must be met; for example, the program must control pressure AND temperature. Disjunction is needed to allow tolerances in implementation. The system may deviate from the optimum by one OR two degrees; and negation is needed for even more important reasons: it must NOT explode.

As a consequence of the expressive power of ordinary logic, it is possible to write specifications like 'P or not P', which is always true, of course, and so describes every possible observation of every possible program. Such a tolerant specification is easy to implement. It can be implemented by a program even that goes into an infinite loop.

Another extreme case is the specification 'P and not P', which, of course, is always false. No product could ever meet such a contradictory specification. It is a symptom of an error that should be removed before implementation starts.

Another inspiring insight which I share with the designers of logic programming languages is that programs too are predicates. When given an appropriate reading, a program describes all possible observations of its behavior under execution. A programming language provides a number of connectives for assembling complex programs out of simpler components. For example, Prolog provides a form of sequential composition denoted by a comma (,) and pronounced 'and then'; a form of sequential disjunction denoted by a semicolon (;) which is pronounced 'or then', and a form of strong negation denoted by a tilde (~) and pronounced, perhaps, 'impossible'. These operators are specifically designed for efficient execution on computers, and they are quite different from the Boolean connectives used for specifications, which are intended to be clear and simple rather than executable.

Nevertheless, even executable operators can be given an accurate (though somewhat more complicated) interpretation entirely within the predicate calculus, as I will show in this talk. I believe the insight is much more general than just Prolog, and it applies to many other languages and, indeed, to any engineering product that can be described in any meaningful design notation.

This view gives rise to a philosophy of engineering which I will illustrate briefly by application to hardware design, to conventional programs, and to the procedural interpretation of Prolog programs. But it will be wholely invalid to claim that all predicates can be read as programs.

Consider the example we've seen before; the contradictory predicate, 'P and not P', which is always false. No computer program could ever produce an answer with this contradictory property. So, this predicate is not a program, and no processor could translate it into one; and any theory that I might invent, which abscribes to an implementable program a behavior which is known to be unimplementable must be an incorrect theory and dangerous to use.

In contrast, of course, if 'P' is a program expressed wholly in Prolog notation, the program (P, ~ P) is a perfectly meaningful program, although it will always fail either finitely or infinitely. My reading of Prolog program as predicates will describe exactly this behavior of the failing program.

A programming language can therefore, be identified only with the subset of the predicate calculus. Each predicate in this subset is a precise discription of all possible behaviors of some program expressible in the language. And the subset is designed to exclude contradictions and all the other unimplementable predicates, and the notations of the language are carefully designed to maintain this exclusion.

In principle, restrictions in the 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.

Since both programs and specifications are predicates, the correctness of the program can be established by a simple implication. Given a specification 'S', the programmer's task is to find a program 'P' which satisfies it in the sense that every possible observation of every possible run of the program 'P' will be described and therefore permitted by this specification 'S'. In logic, this can be assured with mathematical certainty by a proof of the simple implication: 'P implies S'. This simple explanation of what it means for a program or a product to meet its specification is one of the main reasons for interpreting both programs and specifications within the uniform predicate calculus.

Now we can explain the necessity for excluding the contradictory predicate 'false' from a programming notation. It is a theorem of elementary logic that 'false' implies S for any specification S. So, 'false' enjoys the miraculous property of satisfying every specification whatsoever. Anything that you want to do, that program will achieve. Such miracles do not exist, which is fortunate, because, if they did, we would never need anything else, certainly not programs or programming languages, nor computers, nor even computer scientists or programmers.

A very simple example of this philosophy can be taken from the realm of conventional procedural programming. Here, the most important observable values are those which can be observed of the states of the machine before the program starts, and the state which can be observed when the program has terminated.

Let us use the name X to stand for the initial value of some variable, perhaps called X, and let X' to be the final value of that same integer variable; the only one that need be of concern to us now.

Let the specification S say that 'X' is greater than X'; in other words, the value of the variable X must be increased. Let the program P be defined as adding one to X. When this program is interpreted as a predicate, its effect is always that 'the final value X' is equal to one plus the initial value,X'. (Fig. 1)

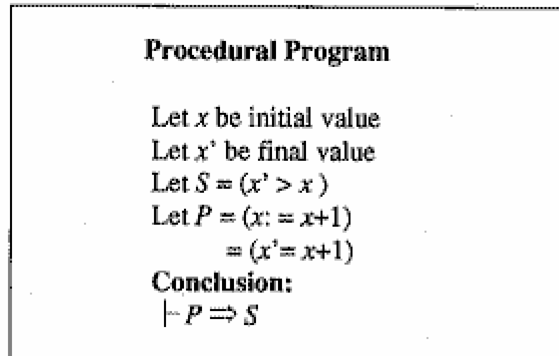


Fig. 1

Every observation of the behavior of P in any initial state (independent of the initial value of X) will satisfy this predicate. Consequently, the validity of the implication that 'P implies S' will ensure that P correctly meets the specification that it

strictly increases the value of X.

Hardware Design

To illustrate the generality of my philosophy my next examples will be drawn from the realm of combinational hardware circuits. These too can 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 the simple equation 'X equal the lesser of A and B'. The values of these free variables are taken to be 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 certain to be the lesser of the two voltages on the input wires A and B.

Similarly, an OR-gate can be described by a different predicate with a different set of wire names: 'D is equal to the greater of Y and C'. A simple wire is device which maintains the same voltage at each of its ends. For example, in this case, 'X equals Y'. (Fig. 2)

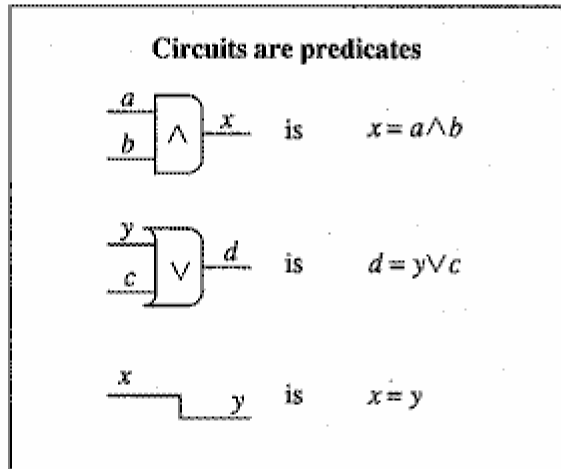


Fig. 2

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 no connection between the two components. Consequently, the simultaneous joint observations of the behavior of the assembly consists solely of their two behaviors evolving in parallel concurrently and independently of each other. And these behaviors are correctly described by just the conjunction of the two predicates describing their individual behaviors. (Fig. 3)

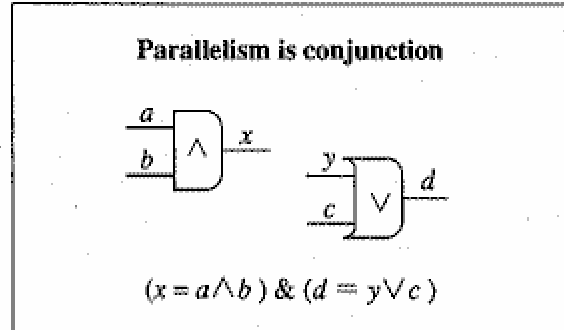


Fig. 3

This simple example is a convincing illustration of the principle that parallel composition of components can often be modeled by a conjunction of the predicates describing their behavior, 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 shown in Fig. 4. The triple conjunction still accurately describes the behavior of the whole assembly. I have simplified the predicate to a mathematically equivalent form which is shown on the last line of this figure.

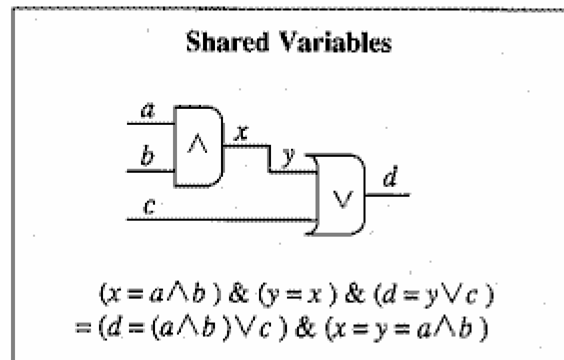


Fig. 4

When components are connected together in this way by the sharing of variable names, for example, X and Y in this case, the values of these 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 of the assembly (as it were) in a black box. The variables, therefore, need to be removed from the predicate which describes the behavior of this black box. And the standard way of eliminating such free variables in the predicate calculus is by quantification.

In the case of engineering design, existential quantification is the right choice. It is necessary that there exist potentially observable values for the hidden variables, X and Y, but no-one cares exactly what those values are going to be. In our hardware example, the existentially quantified formula shown on the figure (Fig. 5) can be very simply reduced to just 'D equals the lesser of A and B, and the maximum of that with C'. The final formula mentions only the visible external wires of the circuit, and probably expresses the intended function or specification of our little assembly. And if that is so, then what I have shown is just a simple proof that the assembly meets its specification.

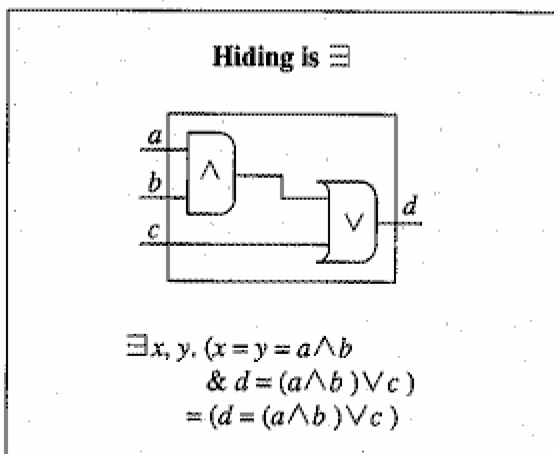


Fig. 5

Unfortunately not all conjunctions of predicates lead to implementable designs in this theory. Consider, for example, the conjunction of a negation circuit 'X equals not Y', with the wire 'Y equals X', which connects its output back again to its input. In practice, this assembly leads to something like an electrical short circuit or an infinite oscillation which is completely useless. In fact it is even worse than useless, because it will prevent the proper operation of other circuits in its vicinity; it might even interfere with your transistor radio. (Fig. 6)

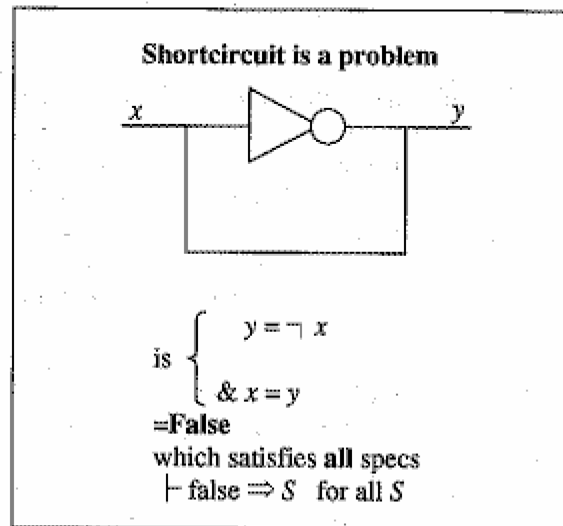


Fig. 6

So, there is no specification other than the trivial specification TRUE which can be met in practice by that circuit. But, in the oversimplified theory which I have just described to you, the predicted effect is exactly the opposite. The predicate describing the behavior of this circuit is a contradiction, false, which is necessarily unimplementable.

The standard solution to this kind of problem is to introduce into the theory a sufficient number of new values and new variables to ensure that one can describe all possible ways in which an actual assembly can go wrong. In the example of circuits, this requires, perhaps, a three-valued logic. In addition to high voltage and low voltage, we introduce an additional value called bottom and written as $\perp$. And this is the value which we take to be observed on a wire which is engaged in oscillation or short circuit. (Fig. 7)

Solution: three-valued logic

$\perp$ **means "shortcircuit"**
$\neg \perp = \perp, \perp \wedge x = \perp$, **etc.**
$\vdash y = \neg y \Rightarrow y = \perp$
Moral:
Don't ignore errors.

Fig. 7

We define the result of any Boolean operation on the bottom element to give the answer bottom. And now, we can solve the problem of the circuit with feedback whose behavior is described by the conjunction 'X equals not Y', and 'Y equals X'. In three-valued logic this no longer a falsehood; in fact it correctly implies that both X and Y are short circuitted.

And the moral of this example is: 'don't ignore errors'. Predicates must describe the behavior of a design including all the ways in which that design can go wrong. It is only in a theory which mathematically models the possibility that your design is incorrect, that you can actually prove or calculate that your design does not fall into that particular kind of error.

Programming Languages

If parallelism is just a 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 or Q' describes a product which may behave like P, or it may behave like Q, but it does not determine which of these it shall be. Consequently you cannot predict or control the result. If you want 'P or Q' to satisfy a specification S, it is necessary and also 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 in the implication ordering of predicates.

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 trace of execution. The behavior of a program defined recursively can most simply be described by using recursion in the definition of the corresponding predicate.

Let 'P of 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. The existence of such a fixed point and its unique identity as a least fixed point is guaranteed by the famous Tarski theorem, provided that 'P of X' is a monotonic function of X, in the space of all predicates regarded as a complete lattice. This is a completely non-operational definition of recursion, equally applicable to programs and specifications.

In a conventional sequential programming language, it is essential to distinguish the values of program variables in the initial state from those in the final state. Let X stand for an observation of the initial values of all the variables of the program, and let X' stand for the final state. Either or both of these maybe take the special value bottom, which, in this case, stands for non-termination, or infinite failure, which is one of the worst ways in which a recursive program can go wrong. Each program is a predicate describing a relationship between the initial state X and the final state X'.

For example, the identity program, which I call ' II ' and is otherwise known as a null-operation; it does nothing, but terminates successfully without making any change to its initial state. But it can be guaranteed to do this only if it starts in a proper state: one which has not already failed. So, the null-operation program can be defined as an implication that if the starting state is not bottom, then the final state is equal to the starting state.

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 X of P and the final state X' of Q. A formal definition of the composition of two predicates is the same as the definition of conventional relational composition in the relational calculus. (Fig. 8)

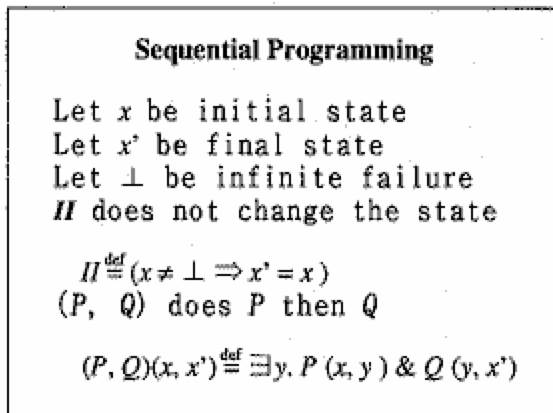


Fig. 8

Now, care must be taken in the definition of a programming language to ensure that sequential composition never becomes self-contradictory. For example, if P states that its final value is non-zero and Q states that its initial value is zero, then their composition immediately reduces to the contradiction 'false'.

The solution in this case is to ensure that all programs expressed in the restricted notations of your programming language do in fact satisfy certain 'healthiness conditions'. In the case of sequential programs, these conditions state that whenever X or X' take the failure value bottom, then the behavior of the program is entirely unpredictable; anything whatsoever may happen. (Fig. 9)

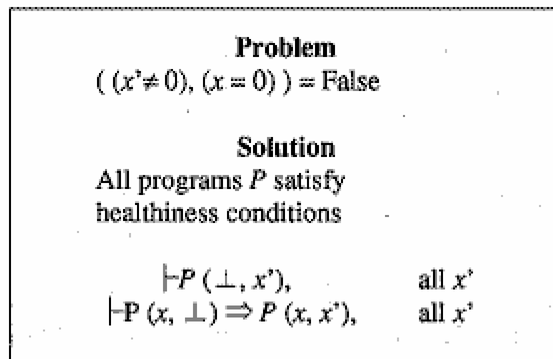


Fig. 9

The imposition of these conditions certainly complicates the theory and it requires the theorist to prove that all programs expressed in the programming notations do satisfy the healthiness conditions. You may note that the predicate false does not satisfy these healthiness conditions, and therefore, the same proof will guarantee that the theory does not make the mistake of ascribing the value 'false' to any of the programs in the language.

Now, the reason for undertaking 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 reasoning is by symbolic calculation using algebraic equations which have been proved to be correct for the theory. For example, to compose a null-operation II, either in front of or after any program P, leaves the observable effect of that program unchanged. Also, composition is associative: to follow the pair of operations P, Q by the operation R is the same as following P by the pair of operations Q and R. And finally, composition of programs distributes through non-deteministic choice 'or', in both directions. (Fig. 10)

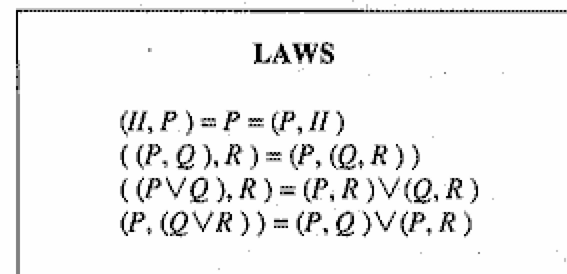


Fig. 10

Prolog

I now come to the boldest and most frightening claim of my whole talk; it is the claim that even Prolog programs are predicates. Each predicate describes the possible behavior of the program when it is actually executed. I am talking about the procedural reading of Prolog in which the language is much more like a sequential programming language. It has an initial state and a final result, and its behavior is defined as a relation between those two. Of course, this is quite different from the predicate ascribed to the program by the logical reading. I am interested in describing the actual behavior of computing a Prolog program, namely its procedural reading.

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 another on request.

I can now define the two simplest of all Prolog programs, namely the program NO which is defined always to fail finitely; when started in any properinitial state θ, it will deliver a result θ ' which is equal to the empty sequence of answers. (Fig. 11)

PROLOG programs are predicates

Let θ be the initial substitution
Let θ ' be the sequence of answers
$\perp$ denotes infinite failure
[ ] denotes finite failure

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

$$YES\,(\theta,\ \theta') \stackrel{def}{=} (\theta \neq \perp \Rightarrow \theta' = [\,\theta\,])$$

Fig. 11

The next simplest program is the program YES; when started in any non-failing state θ, it will give an answer which is exactly the same as the original state θ which it started with but wrapped up as a sequence with only one element.

Let me give you a more substantial example, the familiar Prolog program APPEND. Suppose we start this in an initial state θ equal to Z = [1, 2].

Then, if you apply APPEND (X, Y, Z) to that initial state, it will produce a sequence of answers shown on three successive lines of figure? (Fig. 12) In the first answer X will take the empty sequence, Y will take the full sequence, and Z will be unchanged. In the second answer, the two elements of the sequence will be split between X and Y, and in the third answer the X will take the value of the full sequence, and Y will be empty.

append (X, Y, Z)

Let θ = "Z = [1, 2]"
Then θ '=

"X = [ ], Y = [1, 2], Z = [1, 2]"
"X = [1], Y = [2], Z = [1, 2]"
"X = [1, 2], Y = [ ], Z = [1, 2]"

Fig. 12

The effect of Prolog OR (sequential OR), is obtained by just appending the sequence of answers produced by the first operand in front of those porduced by the second operand. Each of P and Q start in the same initial state θ. The program P produces the answers X, the program Q produces the answers Y, and the answers produced by (P; Q) are obtained by appending X and Y to give θ '. (Fig. 13) The definition of APPEND is

PROLOG or (;) is append

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

where append ([$\perp$], Y, Z)
append ([ ], Y, Y)
append ([X |Xs], Y, [X |Zs])
if append (Xs, Y, Zs)

Fig. 13

the same as you will usually see, except I have an additional clause to define an arbitrary result in the case that the initial sequence is an infinite failure.

In all good mathematical theories, every definition should be followed by collection of useful and memorable theorems. For example, since NO gives no answer, its addition to a list of answers supplied by P makes no difference. So, NO is a

unit of the Prolog semicolon. Similarly, the associative property of appending lifts to the composition of programs. And finally, the sequential OR of Prolog distributes through the truly non-deterministic OR, which I have described as Boolean disjunction. (Fig. 14)

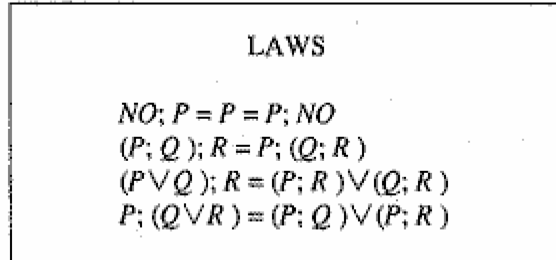


Fig. 14

Prolog conjunction is very similar to sequential composition of a conventional language, modified systematically to deal with the 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 the sequences are all concatenated together using the concat function, complicated but it obeys quite simple algebraic laws. (Fig. 15)

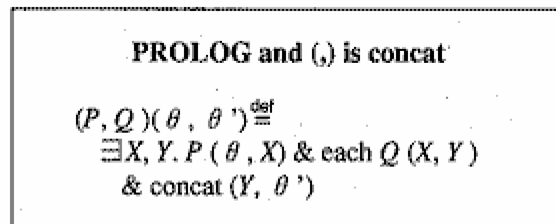


Fig. 15

The first law states that the answer YES is a unit of the sequential composition, and makes no difference to the behavior of the program. Sequential composition as you would expect is associative, and it has a zero NO. But note that this is a left zero only; the converse law does not hold exactly because of the possibility that P may fail infinitely. (Fig. 16)

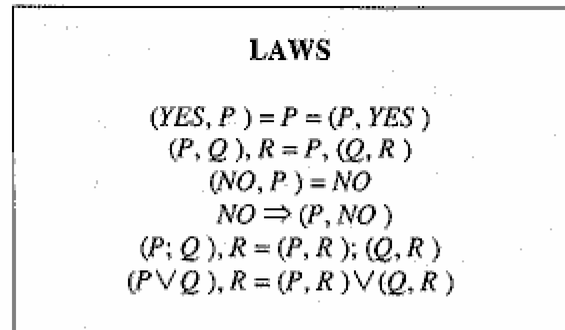


Fig. 16

Finally, sequential composition distributes leftwards through the Prolog sequential OR, and through disjunction or non-determinism; but the complementary laws of rightward distribution do not hold.

The test of our procedural semantics for a Prolog is its ability to deal with the so-called non-logical features of the language like the CUT, which I will treat here 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 any way. (Fig. 17) This result is

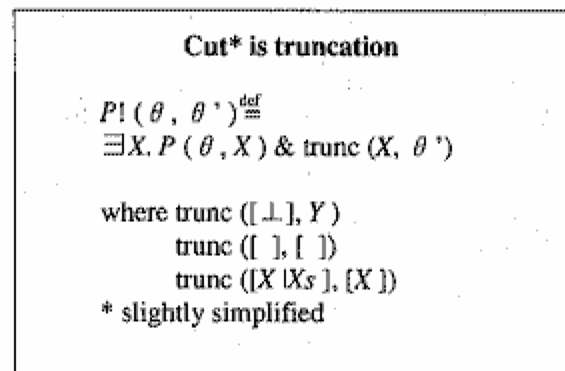


Fig. 17

obtained simply by truncating the sequence. The truncation operator is defined to preserve both infinite failure and finite failure, but in other circumstances it merely removes the redundant elements of the list of answers produced by its argument. And the laws describing the behavior of this CUT are really also quite simple. The first law states that cutting is idempotent, and expresses the obvi-

ous fact that if you cut a program which already produces at most one answer, you will not change its observable behavior.

The next two laws are interesting, because they show that, if only one result is wanted from the compositive of two programs, then in many cases it is sufficient to compute only one result from the components. (Fig. 18) For example, if

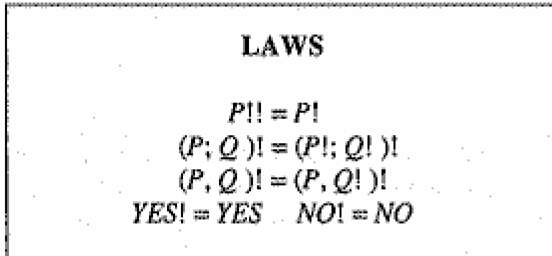


Fig. 18

you cut the Prolog OR (semicolon), you might as well cut the two operands P and Q first; and the same is true of just the second operand of sequential conjunction. These laws, of course, may be used significantly to improve the efficiency of execution of programs involving cuts. Finally, it is obvious that both YES and NO produce only one answer anyway, and so they are unchanged by cutting.

Prolog negation is no more difficult to treat than the cut. It, too, just applies a simple list operation NEG to the result produced by its operand. The NEG predicate is defined to preserve infinite failure, to turn finite failure into success, and in all other cases to give finite failure. (Fig. 19)

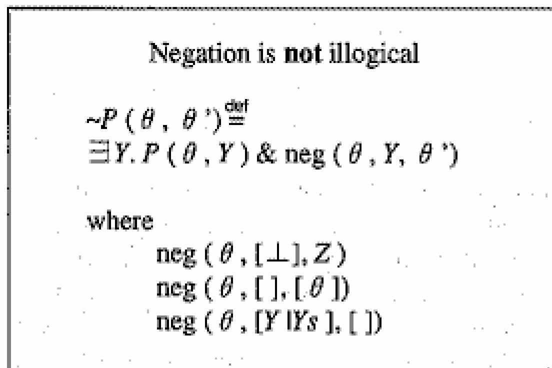


Fig. 19

The laws governing Prolog negation of truth values are the same as those for Boolean negation in the case of YES and NO. The classical law for double negation does not hold; it has to be weakened to an intuitionistic triple negation law. (Fig. 20)

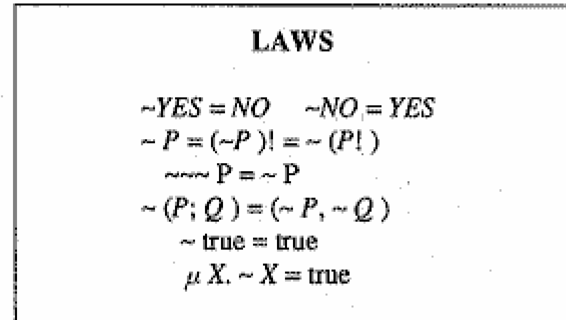


Fig. 20

Finally, there is an astonishing analog of one of the familiar laws of De Morgan in Boolean algebra. Negation distributes through sequential OR, changing it to sequential AND. The AND formula is obviously much more efficient to compute than the OR formula, so this could be effective in optimization. The dual De Morgan law, however, does not hold.

That concludes my simple account of the basic structures of Prolog. All these structures are deterministic in the sense that, in the absense of infinite failure, for any given initial substitution θ, there is at most one answer substitution sequence θ' that can be produced by the program. But the great advantage of reading programs as predicates is the simple way in which we can introduce non-determinism.

For example, many researchers have proposed to improve the sequential OR of Prolog. (Fig. 21) One improvement to the sequential OR is to make it commute like Boolean disjunction, and another is to allow parallel execution of both the operands with an arbitrary interleaving of the results. These two advantages can be combined simultaneously by the definition of a parallel OR, in which the results X and Y produced by P and Q are interleaved instead of being appended to each other.

Now, the parallel OR preserves many of the algebraic laws of the sequential OR and, in addi-

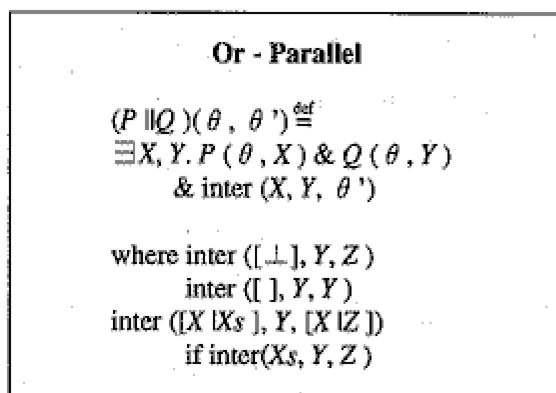


Fig. 21

tion, it is symmetric. Because appending is just a special case of interleaving, we know that APPEND (X, Y, Z) implies interleaves (X, Y, Z). (Fig. 22) As a result, the sequential OR is a special

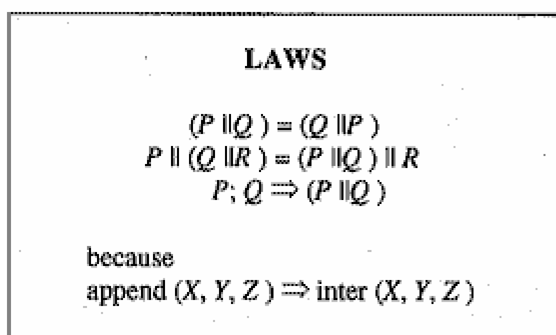


Fig. 22

case of parallel OR, and is always a valid implementation of it. Sequential OR is more deterministic than parallel OR. It is easier to predict; it is easier to control, and it meets every specification that is met by the other. In short, the sequential OR is, in all ways and all circumstances and for all purposes, better than the parallel OR. In all ways except one: it may be slower to implement on a parallel machine.

In principle, my non-determinism represented by Boolean disjunction is demonic; it never makes programming easier because the programmer has to assume that the machine (or the demon) will always choose the alternative that he or she does not want. However, in many cases including this one, non-determinism simplifies specifications and designs, and facilitates reasoning about them at higher levels of abstraction.

KL1

At the beginning of my talk, I gave credit for what I have learned from the original designers of the Prolog language and the logic programming movement. To conclude my talk, I would like to summarize what I have learned from the designers of the KL1 language, designed, implemented, and used by the Fifth Generation Computing Systems Project.

I would like to compare it briefly with the occam programming language designed ten years ago by David May and his colleagues at Inmos in England. In September 1982, I visited Japan with them to announce and present this language occam to the newly formed ICOT. Like KL1, occam was designed for parallel execution on a new generation of parallel computers known as the transputer. And transputers were designed primarily for embedded real-time applications running on small networks of machines. This required the utmost efficiency of implementation, with static allocation of processors and storage to logical processes. Your language KL1 was designed ten years later, when networks of a hundreds of machines have become feasible. That means that from the start you must accept the overhead of dynamic processor allocation. Your language KL1 was designed for a different but equally challenging range of applications, which require the processing of highly irregular and dynamically evolving symbolic data structures. That means that from the start you have to accept the overhead of dynamic storage allocation and re-allocation. These, I believe, are the main reasons for the radical differences between KL1 and occam. But the qualitative similarities between the languages are even more remarkable. Both languages are small and simple; they can be quickly learned and used by specialists in other domains who are not professional programmers. But because of the dynamic storage allocation, KL1 is simpler.

Both languages are highly efficient and permit full advantage to be taken of the power of the individual processors of the system. But, because of

static storage allocation, perhaps occam is still somewhat more efficient. Both languages have a clear abstract semantics independent of the operational interpretation. This permits a complete compatability of implementation on parallel machines with widely differing structures and architectures.

The semantics of both the languages is expressible in terms of predicates describing the actual behavior of an executing mechanism. This permits a reliable development path from specifications expressed as more abstract predicates describing the user requirements in the real world. The semantics permits the derivation of a number of elegant and understandable algebraic laws. These are not only an aid to the human intellect and understanding; they validate both local optimizations and global restructuring of programs to match the architectural features of a particular executing mechanism. I would much rather optimize a non-deterministic KL1 program than try to look for probably non-existent parallelism in a FORTRAN program.

Both languages promote a high degree of concurrency, and permit fine level of granularity, but KL1 permits an even finer degree of granularity than occam. The fine granularity ensures that each physical processor in the system can be time-shared between many logical processes, and the user need not worry whether this is being done or not. This is called parallel slackness, and it ensures a high machine utilization and conceals the effects of message latency and cache faults on the execution efficiency.

Further efficiency is gained by a carefully controlled degree of non-determinacy in the sequencing of the execution of the program, and this permits even further reduction of latency delays. But both languages, occam and KL1 have this most important characteristic that they give complete protection to the programmer against the unspeakable horrors of updating shared storage by parallel processes.

And finally, both languages have inspired the design and implementation of a new generation of actual parallel programming architectures, a family of releases of the transputer and a family of various shapes and architectures of parallel inference machine. These have made available sufficient crude computing power to persuade new classes of potential users to learn new methods of programming to solve new classes of problem.

And that is my brief survey of the technical merits that I have discovered in the KL1 language. I sincerely wish it and you a brilliant future and I would like to make a prediction to that effect. But the future will depend on many accidental, commercial, political, and economic factors of which I have little knowledge and competence. The magnitude of the technical achievement is undeniable. It is measured partly, at least, by the skepticism, disbelief, and even laughter of your critics and detracters. The more they tell you that it could not be done, the greater your achievement that you have actually done it.

In addition to your technical and scientific qualities of insight and judgement and invention you have displayed a high degree of sheer courage. I thank you for your courage, and thank you again for inviting me to address you in this last session of the last day of the final conference devoted justly to the celebration of your achievement. Thank you.

Tanaka: Thank you very much for your insightful and heartful talk. We would like to have some discussions if you can. Are there any comments or questions?

Question: I was struck by the examples that you gave in the very beginning of AND and NOT, when you said things like the specification must control the temperature AND the pressure, or NOT explode. And I was wondering, from everything that I've seen about this kind of literature, there's very little connection to how specifications really interface with the real world where there's causality; there's other agents in the real world that are changing things; and obviously it could explode if somebody else does something to interfere with things. So, I was wondering if you could comment on how this view fits in with programs that really have to control and interact with the real world.

Hoare: Thank you for the question. There is al-

ways a problem of establishing a link between a mathematical and scientific theory and what you can observe or want to observe in the real world. That is a question that will continue to exercise philosophers for many years to come. The only way in which I know of establishing that vital link is to do it informally; no formal mechanism could ever do it. And the informal mechanism is to rely on the understanding of domain specialists to check that the first formalization of the capture of requirements does indeed correspond to the physical reality of the process which is to be controlled. The way that I would look at the formal specification of a real-time process control system is much more as a description of the plant --- of the physics, of the behavior of the plant, and the failure modes of the plant. If you konw those things, then the behavior of your program can be specified (as it were) just by the mirror image or the inverse image of all the states of the plant which are considered to be desirable, optimum, or at least tolerable.

Since I am not a process physicist, I would not expect to be able to check the validity of the formal capture of specifications at this level. But it seems to me that it is at this level that the formalization of the development process for critical programs must start. We must find a development path for critical programs which permits a computing scientist and software engineer reliably to carry through the exact specifications which have been laid down by those who understand the physical reality being controlled. I think, in the first invited speech of this conference, Dines Bjørner has already described our mutual interest in developing these techniques in a project which we know, in Europe, as PROCOS.

Question: You compared two concurrent programming languages occam and KL1. Would you care to comment about other approaches to concurrency being investigated? They are on the road.

Hoare: Would I care to comment on other approaches to concurrency? I would like to spend all evening commenting on other approaches to concurrency.

The problem of concurrency and highly parallel computers is still with us. It is extremely difficult to foresee how it is that we can persuade the world simultaneously to adopt new paradigms of programming and new hardware structures. The world has learned from bitter experience that the most important consideration is that your next computer should execute exactly the same programs as your previous computer did. And so, the manufacturers of computers cannot find markets for a computer architecture which requires any new programs to be run. I do not have skill in foreseeing the outcome of this. I am quite convinced, though, that when programmers first discover that non-determinacy makes program testing impossible, they will reject the new technologies and the new machines and the new power out of hand. It is only by the use of languages like occam and KL1 that give sound scientific control over non-determinacy that we will ever reliably be able to use these machines; it will be very much a professional programmers' task to make sure that these machines do not contain errors which will not appear during program testing, but will return to plague the user afterwards. I am quite convinced that people who are determined to program using methods and languages which give them no handle and no protection against that kind of error, will in the end, find that the errors accumulate to destroy them.

Tanaka: Is there another small question?

Okada: I hope this is a small question. You collected the predicate: that is the logical concepts and programs --- extending, maybe, traditional doctrine like predicate. For example, specification is predicate; and you said the program is predicate. Then, at that time, if you consider, for example,the correctness proof of the logical implications which you showed, like, the program predicates implied specification. Now, at that time, that is the logical concept or mathematical concept; and this is related to some concept in programming language theory or computer science. Then, at that time, what is the correspondent things of proof in that case ? I mean, you have the proof of the correctness; that is the proof for the logical implications; then, that proof is interpreted by what? What is the

proof in computer science sense in your framework?

Hoare: Thank you. I have, I am afraid, no skill at all in proof theory. The proofs that I am considering are ones which are as close as I can make them to the conventional practice of mathematical proving, which clearly have been well formalized by logicians and set theorists. I believe that mathematical proof and calculation is something that can be done quite reliably by human beings, and even more reliably checked by other human beings. Therefore, I have not given a great deal of attention to proof as a formalized or mechanically checkable activity, I think of proof at the design phase as being no different from proof of the kind that you will see in any mathematical textbook.

Okada: ....in the sense of Hoare logic, for example, in your earlier, former work, you specified the proof system of the correctness of the program; that is so-called the Hoare logic. And you don't think you consider any more that kind of formal framework of your current theory?

Hoare: You're quite right. I think there are now much better ways of specifying and establishing the correctness of programs than Hoare logic. Ever since 1974, when Professor Dijkstra published his work on weakest pre-conditions, I have regarded Hoare logic as obsolete.

Tanaka: Okay. Time is up, and I would like to say thank you again for your distinguished and influential talk. I would like to close this session with....

From now, about twenty minutes, we will take a coffee break. And at three thirty, we will begin the last plenary panel.