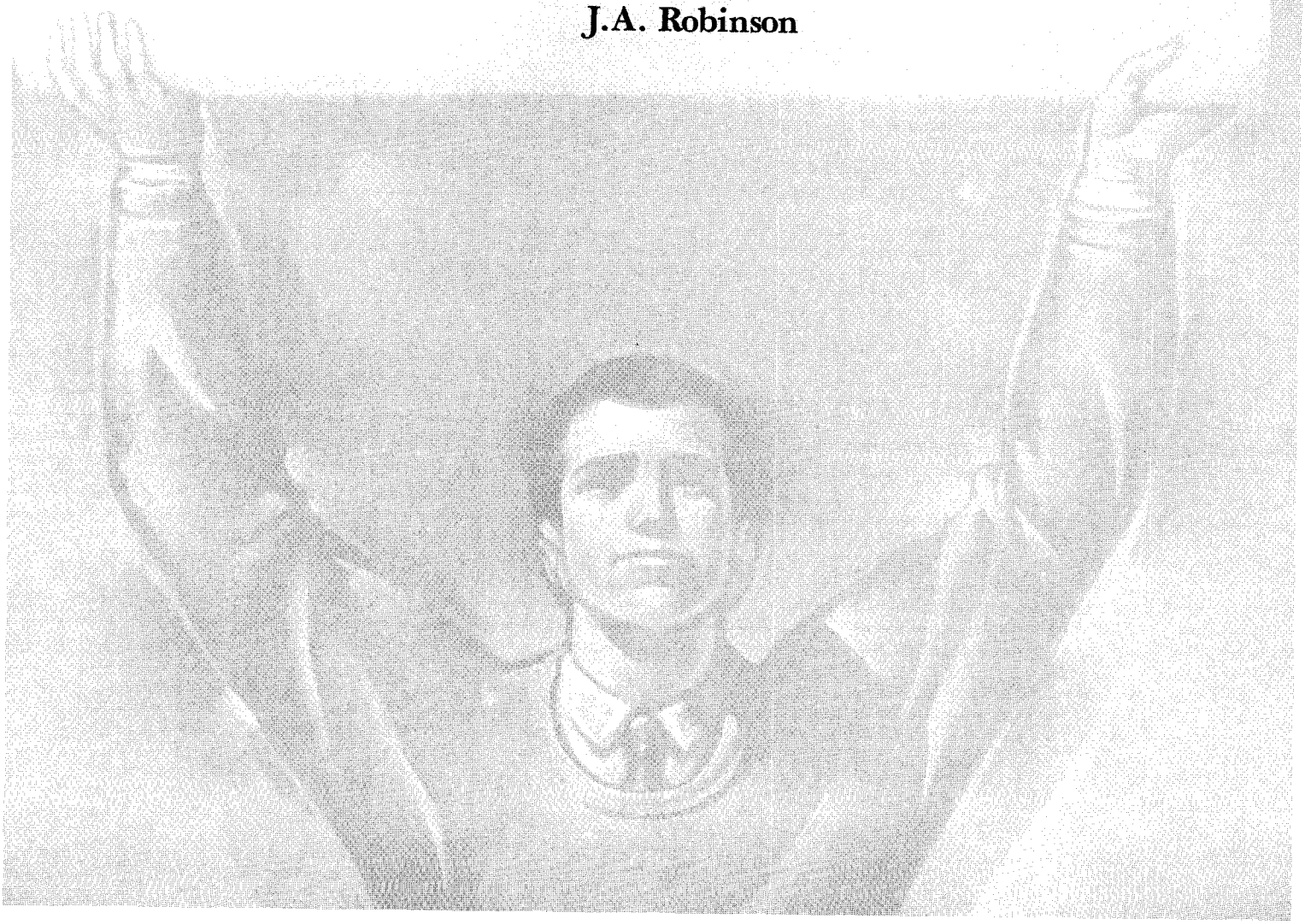
Logic and Logic Programming

J.A. Robinson



Logic has been around for a very long time [23]. It was already an old subject 23 centuries ago, in Aristotle's day (384–322 BC). While Aristotle was not its originator, despite a widespread impression to the contrary, he was certainly its first important figure. He placed logic on sound systematic foundations, and it was a major course of study in his own university in Athens. His lecture notes on logic can still be read today. No doubt he taught logic to the future Alexander the Great when he served for a time as the young prince's personal tutor. In Alexandria a generation later (about 300 B.C.), Euclid played a similar role in systematizing and teaching the geometry and number theory of that era. Both Aristotle's logic and Euclid's geometry have endured and prospered. In some high schools and colleges, both are still taught in a form similar to their original one. The old logic, however, like the old geometry, has by now evolved into a much more general and powerful form.

Modern ('symbolic' or 'mathematical') logic dates back to 1879, when Frege published the first version of what today is known as the *predicate calculus* [14]. This system provides a rich and comprehensive notation, which Frege intended to be adequate for the expression of all mathematical concepts and for the formulation of exact deductive reasoning about them. It seems to be so. The principal feature of the predicate calculus is that it offers a precise characterization of the concept of *proof*. Its proofs, as well as its sentences and its other formal expressions, are mathematically defined objects which are intended not only to express ideas meaningfully—that is, to be used as one uses a language—but also to be the subject matter of mathematical analysis. They are also capable of being manipulated as the data objects of construction and recognition algorithms.

At the end of the nineteenth century, mathematics had reached a stage in which it was more than ready to exploit Frege's powerful new instrument. Mathematicians were opening up new areas of research that demanded much deeper logical understanding and far more careful handling of proofs, than had previously been required. Some of these were David Hilbert's abstract axiomatic recasting of geometry and Giuseppe Peano's of arithmetic, as well as Georg Cantor's intuitive explorations of general set theory, especially his elaboration of the dazzling theory of transfinite ordinal and cardinal numbers. Others were Ernst Zermelo's axiomatic analysis of set theory following the discovery of the logical and set-theoretic paradoxes (such as Bertrand Russell's set of all sets which are not members of themselves, which therefore by definition both is, and also is not, a member of itself); and the huge reductionist work *Principia Mathematica* by Bertrand Russell and Alfred North Whitehead. All of these developments had either shown what could be done, or had revealed what needed to be done, with the help of this new logic. But it was necessary first for mathematicians to master its techniques and to explore its scope and its limits.

Significant early steps toward this end were taken by Leopold Lowenheim (1915), [29] and Thoralf Skolem [45], who studied the symbolic "satisfiability" of formal expressions. They showed that sets of abstract logical conditions could be proved consistent by being given specific interpretations constructed from the very symbolic expressions in which they are formulated. Their work opened the way for Kurt Gödel (1930, [17]) and Jacques Herbrand (1930, [19]) to prove, in their doctoral dissertations, the first versions of what is now called the *completeness* of the predicate calculus. Gödel and Herbrand both demonstrated that the proof machinery of the predicate calculus can provide a formal proof for every logically true prop-

osition, and indeed they each gave a constructive method for finding the proof, given the proposition. Gödel's more famous achievement, his discovery in 1931 of the amazing 'incompleteness theorems' about formalizations of arithmetic, has tended to overshadow this important earlier work of his, which is a result about pure logic, whereas his incompleteness results are about certain applied logics (formal axiomatic theories of elementary number theory, and similar systems) and do not directly concern us here.

The completeness of the predicate calculus links the syntactic property of formal provability with the conceptually quite different semantic property of logical truth. It assures us that each property belongs to exactly the same sentences. Formal syntax and formal semantics are both needed, but for a time the spotlight was on formal syntax, and formal semantics had to wait until Alfred Tarski (1934, [46]) introduced the first rigorous semantical theory for the predicate calculus, by precisely defining *satisfiability, truth* (in a given *interpretation*), *logical consequence,* and other related notions. Once it was filled out by the concepts of Tarski's semantics, the theory of the predicate calculus was no longer unbalanced. Shortly afterward Gerhard Gentzen (1936, [15]) further sharpened the syntactical results on provability by showing that if a sentence can be proved at all, then it can be proved in a 'direct' way, without the need to introduce any extraneous 'clever' concepts; those occurring already in the sentence itself are always sufficient.

All of these positive discoveries of the 1920s and 1930s laid the foundations on which today's predicate calculus theorem-proving programs, and thus logic programming have been built.

Not all the great logical discoveries of this period were positive. In 1936 Alonzo Church and Alan Turing (see [6, 47]) independently discovered a fundamental *negative* property of the predicate calculus. There had until then been an intense search for a positive solution to what Hilbert called the *decision problem*—the problem to devise an algorithm for the predicate calculus which would correctly determine, for any formal sentence B and any set A of formal sentences, whether or not B is a logical consequence of A. Church and Turing found that despite the existence of the proof procedure, which correctly recognizes (by constructing a proof of B from A) all cases where B is in fact a logical consequence of A, there is not and cannot be an algorithm which can similarly correctly recognize all cases in which B is *not* a logical consequence of A. Their discovery bears directly on all attempts to write theorem-proving software. It means that it is pointless to try to program a computer to answer 'yes' or 'no' correctly to *every* question of the form 'is this a logically true sentence?' The most that can be done is to identify useful subclasses of sentences for which a decision procedure can be found. Many such subclasses are known. They are called 'solvable subcases of the decision problem', but as far as I know none of them have turned out to be of much practical interest.

When World War II began in 1939 all the basic theoretical foundations of today's computational logic were in place. What was still lacking was any practical way of actually carrying out the vast symbolic computations called for by the proof procedure. Only the very simplest of examples could be done by hand. Already there were those, however—Turing himself for one—who were making plans which would eventually fill this gap. Turing's method in negatively solving the decision problem had been to design a highly theoretical, abstract version of the modern stored-program, general-purpose universal digital computer (the 'universal Turing machine'), and then to prove that no program for it could realize the decision procedure. His subsequent leading role in the war-time British code-breaking project included his participation in the actual design, construction and operation of several electronic machines of this kind, and thus he must surely be reckoned as one of the major pioneers in their early development.

Logic on the Computer

Apart from this enormously important cryptographic intelligence work and its crucial role in ballistic computations and nuclear physics simulations, the war-time development of electronic digital computing technology had relatively little impact on the outcome of the war itself. After the war, however, its rapid commercial and scientific exploitation quickly launched the current computer era. By 1950, much-improved versions of some of the war-time general-purpose electronic digital computers became available to industry, universities and research centers. By the mid-1950s it had become apparent to many logicians that, at last, sufficient computing power was now at hand to support computational experiments with the predicate calculus proof procedure. It was just a matter of programming it and trying it on some real examples. Several papers describing projects for doing this were given at a Summer School in Logic held at Cornell University in 1957. One of these [37, pp. 74–76] was by Abraham Robinson, the logician who later surprised the mathematical establishment by applying logical 'non-standard' model theory to legitimize infinitesimals in the foundations of the integral and differential calculus. Other published accounts of results in the first wave of such experiments were [12, 16, 35, 49]. There had also been, in 1956, a strange experiment by [33] which attracted a lot of attention at the time. It has since been cited as a milestone of the early stages of artificial intelligence research. The authors designed their 'Logic Theory Machine' program to prove sentences of the propositional cal-

culus (*not* the full predicate calculus), a very simple system of logic for which there had long existed well-known decision procedures. They nevertheless explicitly rejected the idea of using any algorithmic proof procedure, aiming, instead, at making their program behave 'heuristically' as it cast about for a proof. This experiment was intended to model human problem-solving behavior, taking propositional calculus theorem-proving in particular as the problem-solving task, rather than to program the computer to prove propositional calculus theorems efficiently.

No sooner were the first computational proof experiments carried out than the severe combinatorial complexity of the full predicate calculus proof procedure come vividly into view. The procedure is, after all, essentially no more than a systematic exhaustive *search* through an exponentially expanding space of possible proofs. The early researchers were brought face-to-face with the inexorable 'combinatorial explosion' caused by conducting the search on nontrivial examples. These first predicate calculus proof-seeking programs may have inspired, and perhaps even deserved, the disparaging label 'British Museum method' (see [33]), which was destined to be pinned on any merely-generate-and-test procedure which blindly and undiscriminatingly tries all possible combinations in the hope that a winning one, or even an acceptable one, may eventually turn up.

The intrinsic exponential complexity of the predicate calculus proof procedure is to be expected, because of the nature of the search space. There is evidently little one can do to avoid its consequences. The only reasonable course is to look for ways to strengthen the proof procedure as much as possible, by simplifying the forms of expressions in the predicate calculus and by packing more power into its inference rules. This might at least make the search process more efficient, and permit it to find proofs of more interesting examples before it runs into the exponential barrier.

Some limited progress has been made in this direction by reorganizing the predicate calculus in various 'machine-oriented' versions.

Evolution of Machine-Oriented Logic

The earliest versions of the predicate calculus proof procedure were all based on *human-oriented* reasoning patterns—on types of inference which reflected formally the kind of 'small' reasoning steps which humans find comfortable. A well-known example of this is the *modus ponens* inference-scheme. In using *modus ponens,* one infers a conclusion B from two premisses of the form A and (**if** A **then** B). Such human-oriented inference-schemes are adapted to the limitations—and also to the strengths—of the human information-processing system. They therefore tend to involve *simple, local, small* and perceptually *immediate* features of the state of the reasoning. In particular, they do not demand the handling of more than one such bundle of features at a time—they are designed for *serial* processing on a single processor. The massive parallelism in human brain processes is well below the level of conscious awareness, and it is of the essence of deductive reasoning that the human reasoner be fully conscious of the 'epistemological flow' of the proof and of its stepwise assembling of his or her assent and understanding. In logics based on such fine-grained serial inference patterns, proofs of interesting propositions will tend to be large assemblies of small steps. The search space for the corresponding proof procedures will accordingly tend to be dense and overcrowded with redundant alternatives at too low a level of detail.

By about 1960 it had become clear that it might be necessary to abandon this natural predilection for human-oriented inference patterns, and to look for new logics based on larger-scale, more complex, less local, and perhaps even highly parallel, *machine-oriented* types of reasoning. In contemplating these possible new logics it was hoped their proofs would be shorter and (at the top level) simpler than those in the human-oriented logics. Of course, in the interior of any individual inference, there would presumably be a large amount of hidden structural detail. The global search space would be sparser, since it would need to contain only the top-level structure of proofs. The proof procedure itself would not need to be concerned with the copious details of the conceptual microstructure packaged within the inference steps.

This was the motivation behind the introduction, in the early 1960s, of a new logic, based on two highly machine-oriented reasoning patterns: *unification,* and the various kinds of *resolution* which incorporate it.

Clausal Logic

The 1960 paper [12] had already drawn attention to the simplified *clausal* predicate calculus in which every sentence is a *clause.* (A clause is a sentence with a very simple form: it is just a—possibly empty—disjunction of literals. A literal, in turn, is just the application of an unnegated or negated predicate to a suitable list of terms as arguments). In the same year, Dag Prawitz [34] had also forcefully advocated the use of the process which we now call *unification.* Along with Stig Kanger (see [34, footnote 11], p. 170) he apparently had independently rediscovered unification in the late 1950s. He apparently did not realize that it had already been introduced by Herbrand in his thesis of 1930 (albeit only in a brief and rather obscure passage). These were major steps in the right direction. Neither the Davis-Putnam nor the Prawitz improved proof procedures, however, went quite far enough in discarding human-oriented inference patterns, and their algorithms still

became bogged down too early in their searches, to be useful.

This was the situation when I first became interested in mechanical theorem-proving in late 1960. From 1961 to 1964 I worked each summer as a visiting researcher at the Argonne National Laboratory's Applied Mathematics Division, which was then directed by William F. Miller. It was Bill Miller who in early 1961 first introduced me to the engineering side of predicate calculus theorem-proving by pointing out to me the Davis and Putnam paper. He invited me to spend the summer of 1961 at Argonne as a visiting researcher in his division, with the suggested assignment of programming the Davis-Putnam proof procedure on the IBM 704 and more generally of pursuing mechanical theorem-proving research.

Reading the Davis-Putnam paper [12] in early 1961 really changed my life. Although Hilary Putnam had been one of my advisers when I was working on my doctoral thesis in philosophy at Princeton (1953–1956), my research had dealt with David Hume's theory of causation and had little or nothing to do with modern logic, to which I paid scant attention at that time. I did not find out about Putnam's interest in the predicate calculus proof procedure until I read this paper, four years after I had left Princeton. It is a very important paper. They showed how, by relatively simple but ingenious algorithmic reorganization, the original naive predicate calculus proof procedure of Herbrand could be *vastly* improved.

In a 1963 paper I wrote about my 'combinatorial explosion' experience with programming and running the Davis-Putnam procedure in Fortran for the IBM 704 at Argonne [38, pp. 372–383]. Meanwhile, during my second research summer there (1962) an Argonne physicist who was interested in and very knowledgable about logic, William Davidon, had drawn my attention to the important 1960 paper by Dag Prawitz [34], in which I first encountered the idea of unification. After struggling with the woeful combinatorial inefficiency of the instantiation-based procedure used by Davis and Putnam (and by everybody else at that time; it goes back to Herbrand's so-called 'Property B Method' developed in [19]). I was immediately very impressed by the significance of this idea. It is essentially the idea underlying Herbrand's 'Property A Method' developed in the same thesis. Here again was still another paper showing that *even vaster* improvements than those flowing from the Davis and Putnam paper were possible over the 'naive' predicate calculus proof procedure. Instead of generating-and-testing successive instantiations (substitutions) hoping eventually to hit upon the right ones, Prawitz was describing a way of *directly computing* them. This was a breakthrough. It offered an elegant and powerful alternative to the blind, hopeless, enumerative 'British Museum' methodology, and pointed the way to a new methodology featuring deliberate, goal-directed constructions.

The entire academic year of 1962–1963 was consumed in trying to figure out the best way to exploit this Herbrand-Kanger-Prawitz process effectively, so as to eliminate the generation of irrelevant instances in the proof search. Finally, in the early summer of 1963, I managed to devise a clausal logic with a single inference scheme, which was a combination of the Herbrand-Kanger-Prawitz process (for which I proposed the name *unification*) with Gentzen's 'cut' rule. This combination produced a rather inhuman but very effective new inference pattern, for which I proposed the name *resolution*. Resolution permits the taking of arbitrarily large inference steps which in general require very considerable computational effort to carry out (and in some cases even to understand and to verify). Most of the effort is concentrated on the unification involved. Preliminary investigations indicated that resolution-based theorem-provers would be significantly better than any which had been built previously.

I wrote about these ideas at Argonne at the end of the summer of 1963, and sent the paper to the *Journal of the A.C.M. (JACM)*. It then apparently remained unread on some referee's desk for more than a year. It required some urging by the then editor of the Journal, Richard Hamming of Bell Laboratories, before the referee finally responded. The outcome was that the paper, [39], was published only in January 1965. Meanwhile the manuscript had been circulating. In 1964 at Argonne, Larry Wos, George Robinson and Dan Carson programmed a resolution-based theorem prover for the clausal predicate calculus, adding to the basic process search strategies (called *unit preference* and *set of support*) of their own devising, which further speeded the resolution proof process. Because of the refereeing delay, their paper, reached print before mine [52]) and could only cite it as 'to be published'.

Throughout the winter of 1963–64, while waiting for news of the paper's acceptance or rejection by *JACM*, I concentrated on trying to push the ideas further, and looked for ways of extending the resolution principle to accommodate even larger inference steps than those sanctioned by the original binary resolution pattern. One of these turned out to be particularly attractive. I gave it the name *hyper-resolution*, meaning to suggest that it was an inference principle on a level *above* resolution. One hyperresolution was essentially a new integrated whole, a condensation of a *deduction* consisting of *several* resolutions. The paper describing hyperresolution was published at about the same time as the main resolution paper, and was later reprinted in [40, pp. 416–423].

It had been my guiding idea in this research that bigger and (computationally) better inference patterns might be obtained by somehow packaging *entire deductions* at

one level into *single inferences* at the next higher level. As I cast about for such patterns I came across a quite restricted form of resolution—I called it 'P1-resolution'—which I found I could prove was just as powerful as the original *unrestricted* binary resolution. The restriction in P1-resolution is that one of the two premises must be an *unconditional* clause, that is, a clause in which there are no negative literals (or what amounts to the same thing, a sentence of the form: '**if** *antecedent* **then** *consequent*' whose *antecedent* part is empty). From this restriction, it follows that every P1-deduction (that is, a deduction in which every inference is a P1-resolution) can always be decomposed into a combination of what I called 'P2-deductions'. A P2-deduction is a P1-deduction which satisfies the extra restriction that its conclusion, and all of its premises except one, are unconditional clauses. Thus, *exactly one* conditional clause is involved as an 'external' clause in a P2-deduction. By ignoring the *internal* inferences of a P2-deduction tree and deeming its conclusion to have been directly obtained from its premises, we obtain a single large inference—a hyperresolution—which is really a multiinference deduction whose interior details are hidden from view inside a sort of logical black box.

Computational Logic: The Resolution Boom

After the publication of the paper in 1965, there began a sustained drive to program resolution-based proof procedures as efficiently as possible and to see what they could do. In Edinburgh, Bernard Meltzer's *Computational Logic* group and Donald Michie's *Machine Intelligence* group had by 1967 attracted many young researchers who have since become well known and who at that time worked on various theoretical and practical resolution issues: Robert Kowalski, Patrick Hayes, the late Donald Kuehner, Gordon Plotkin, Robert Boyer and J Moore, David H.D. Warren, Maarten van Emden, Robert Hill. Bernard Meltzer had visited Rice University for two months in early 1965 in order to study resolution intensively, and on his return to Edinburgh he set up one of two seminal research groups which were to foster the birth of logic programming (the other being Alain Colmerauer's group in Marseille). Thus began my long and fruitful association with Edinburgh. By 1970 the resolution boom was in full swing. I recall that in that year Keith Clark and Jack Minker were among those attending a NATO Summer School organized by Bernard Meltzer and Nicolas Findler at Menaggio on Lake Como. There we preached the new 'resolution movement' for two weeks, and Clark and Minker decided to join it, soon becoming two notable contributors.

Meanwhile, however, in the U.S., the reaction was mostly muted, except for isolated pockets of enthusiasm at Argonne, Stanford, Rice and a few other places. Bill Miller had left Argonne to go to Stanford at the end of 1964, and I accepted his invitation to spend the summers of 1965 and 1966 as a visiting researcher in his computation group at the Stanford Linear Accelerator Center. It was at Stanford in the summer of 1965 that I met John McCarthy for the first time. I was astonished to learn that after he had recently read the resolution paper he had written and tested a complete resolution theorem-proving program in Lisp *in a few hours*. I was still programming in Fortran, and I was used to taking days and even weeks for such a task. In 1965, however, one could use Lisp easily in only a very few places, and neither Rice University nor Argonne National Laboratory were then among them.

Bertram Raphael, Nils Nilsson, and Cordell Green, at Stanford Research Institute, were building deductive databases for the 'STRIPS' planning software for their robot, and they were adopting resolution for this (see [36]). At New York University, Martin Davis and Donald Loveland were developing Davis's very closely related unification-based 'linked conjunct' method [10, pp. 315–330] in ways which eventually led Loveland independently to his Model Elimination system [28], a linear reasoning method entirely similar to the linear resolution systems developed by the Edinburgh group, and by David Luckham at Stanford [30]. Back at Argonne, Larry Wos and George Robinson had formed a very strong 'automated deduction' group. They broadened the applicability of unification by augmenting resolution with further inference rules specialized for equality reasoning *(modulation, paramodulation)* which further improved the efficiency of proof searches [43]. Today, the Argonne group is still flourishing and remains a major center of excellence in automated deduction.

In 1969 there began a series of noisy but interesting (and, it later turned out, fruitful) academic skirmishes between the then somewhat meagerly funded resolution community and MIT's Artificial Intelligence Laboratory led by Marvin Minsky and Seymour Papert. The MIT AI Laboratory at that time was (it seemed to us) comfortably, if not lavishly, supported by the Pentagon's Advanced Research Projects Agency (then ARPA, now DARPA). The issue was whether it was better to represent knowledge computationally, for AI purposes, in a *declarative* or in a *procedural* form. If it was the former (as had been originally proposed in 1959 by John McCarthy) [31] then it would be the predicate calculus, and efficient proof procedures for it, that would play a central role in AI research. If it was the latter (e.g., see [51]), then a computational realization of knowledge would have to be a system of procedures 'heterarchically' organized so that each could be invoked by any of the others, and indeed by itself. These procedures would be 'agents' that would both cooperate and compete in collectively accomplishing the various tasks comprising intelligent behav-

ior and thought.

Minsky's book, *The Society of the Mind* [32], elegantly summed up the MIT side of this debate in eschewing polemics to outline a grand unified theory of the structure and function of the mind in the tradition of Freud and Piaget. The logic side of the debate has been definitively treated in [25], which eloquently sets forth the role of logic in the computational organization of knowledge and banishes the procedural-declarative dichotomy by insight that Horn clauses (that is, clauses containing at most one unnegated literal) can be interpreted as procedures, and thus can be activated and executed by a suitably designed processor. It is this insight that underlies what we now call *logic programming.*

The never-to-be-implemented but influential 'Planner' system by Carl Hewitt—his first paper on Planner, in [20]—epitomized the MIT procedural approach, while the QA ('Question-Answering') series of programs by [31] carried out McCarthy's logical 'Advice Taker' approach to AI and convinced many skeptics that it would really work. The work by [18] should now be seen and appreciated as the earliest demonstration of a logic programming system. That paper illustrated how to adapt a resolution-based proof procedure to provide an assertion-and-query facility in all essential respects like that provided by the later Prolog systems. Unfortunately, the system was built on the rapidly ramifying full resolution scheme, using unrestricted (rather than Horn-) clauses, so that the program suffered from premature combinatorial explosiveness. Nevertheless, it was largely Green's pioneering work of [18] that encouraged Kowalski and the Edinburgh group to fight off the MIT 'procedural-is-best' attack by developing the highly efficient (LUSH, later called SLD), slowly ramifying linear resolution systems for the restricted case of Horn-clauses [27, pp. 542–577].

The procedural-logical fight was really ended, in a delightfully unexpected way, by Kowalski's inspired *procedural interpretation* of the behavior of a Horn-clause linear resolution proof finder, [24]. He pointed out that in view of the behavior of Horn clause linear-resolution proof-seeking processes, a collection of Horn clauses could be regarded as knowledge organized *both* declaratively *and* procedurally. It suddenly was hard to see what all the fuss had been about. Kowalski was led to this reconciliatory principle by superb implementation of a 'structure-sharing' resolution theorem prover at Edinburgh [5, pp. 101–116], which suddenly completed the transformation of theorem-proving from *generate-and-test searching* to *goal-directed stack-based computation.* When restricted to Horn clauses, the Boyer-Moore approach becomes the obvious precursor of the first implementations of Prolog. David H.D. Warren's enormously influential later software and hardware refinements and advances clearly descend directly from the Boyer-Moore methodology [50].

Only the interaction of the Edinburgh group's ideas with the work of Colmerauer's Montreal [7] and Marseilles [8] groups was required to open up logic programming and launch it on its meteoric career. The interesting story of this interaction was published by [26]. Logic programming is today in excellent health. The logic programming community has settled down to enjoy, after two decades of very rapid growth, a steady mature round of professional conferences and workshops, a plentiful flow of research and expository publication in books and in its own and other journals, an exciting marketplace of new software and hardware enterprises, and such majestic long-range national and international undertakings as Japan's Fifth Generation Project and those sponsored by the European Community.

A Closer Look at Unification and Resolution

What then, is the resolution-based clausal predicate calculus, and what is unification and how does it work?

Clauses

Davis's and Putnam's clauses are quite expressive, despite their apparently restricted form. This is reflected in the many different but equivalent ways in which one can write them. In dealing with clauses computationally, however, it is best to keep them simple and to work with them abstractly.

A clause can in general be taken to be a sentence of the form 'if P then Q', which we will usually write as $P \rightarrow Q$ or sometimes the other way round, as $Q \leftarrow P$. The *antecedent* P is a set of *conditions* and the *consequent* Q is a set of *conclusions.* These conditions and the conclusions are *atomic sentences.* The order in which the atomic sentences perforce are presented in written versions of clauses and has no logical significance. There is usually no visible indication of the fact that the antecedent P is a *conjunction* of its conditions, while the consequent Q is a *disjunction* of its conclusions. Those two facts are assumed to hold by convention. In discussing inferences and manipulations involving clauses, the abstract view of P and Q as sets is both natural and convenient.

We can then classify a clause along three different dimensions, depending on whether its atomic sentences contain any variables or not, whether or not it has any conditions, and whether or not it has any conclusions. A clause with no variables is said to be a *ground* clause, while if it has one or more variables, it is called a *general* clause. A general clause is understood to be a *universally quantified* sentence, each of its variables being tacitly universally quantified with the whole sentence as scope. A clause with one or more conclusions is said to be a *positive* clause; while one with no conclusions is said to be a

negative clause. Finally, a clause with one or more conditions is said to be a *conditional* clause; while one with no conditions is said to be an *unconditional* clause. (There is only one clause that is *both* unconditional *and* negative: it is known as the *empty clause*.)

Various Ways of Reading a Clause

Suppose the variables which occur in the atomic sentences of a clause are $V_1 \ldots V_k$, and that its conditions are $P_1 \ldots P_m$ and its conclusions are $Q_1 \ldots Q_n$. The various ways to read and write the clause will then depend on the values of k, m, and n, as follows:

1	**for all** $V_1 \ldots V_k$: **if** P_1 **and** . . . **and** P_m **then** Q_1 **or** . . . **or** Q_n	$(k > 0, m \geq 1, n > 1)$
2	**for all** $V_1 \ldots V_k$: Q_1 **or** . . . **or** Q_n	$(k > 0, m = 0, n > 1)$
3	**if** P_1 **and** . . . **and** P_m **then** Q_1 **or** . . . **or** Q_n	$(k = 0, m \geq 1, n > 1)$
4	Q_1 **or** . . . **or** Q_n	$(k = 0, m = 0, n > 1)$
5	**for all** $V_1 \ldots V_k$: **if** P_1 **and** . . . **and** P_m **then** Q_1	$(k > 0, m \geq 0, n = 1)$
6	**for all** $V_1 \ldots V_k$: Q_1	$(k > 0, m = 0, n = 1)$
7	**if** P_1 **and** . . . **and** P_m **then** Q_1	$(k = 0, m \geq 1, n = 1)$
8	Q_1	$(k = 0, m = 0, n = 1)$
9	**not** (**for some** $V_1 \ldots V_k$: P_1 **and** . . . **and** P_m)	$(k > 0, m \geq 1, n = 0)$
10	**not** (P_1 **and** . . . **and** P_m)	$(k = 0, m \geq 1, n = 0)$
11	**not true** (or: **false**)	$(k = 0, m = 0, n = 0)$

Horn-clauses are cases 5 onward (where n = 1 or n = 0). The clauses in cases 5 to 8 are *positive* Horn-clauses (n = 1); those from 9 onwards are *negative* Horn-clauses (n = 0). Cases 2, 4, 6, 8 and 11 are *unconditional* clauses (m = 0). The other cases ($m \geq 1$) are *conditional* clauses.

Variants. Separation of Clauses

As we shall soon see, the choice of variables in a general clause is somewhat arbitrary, and neither the essential syntactic structure nor the meaning of a clause are affected if we replace some or all of its variables by other variables. The only proviso is that the correspondence between old and new variables must be one-to-one. Two clauses which differ from each other only in this way are called *variants* of each other. If two clauses have no variables in common, they are said to be *separated* or *standardized apart*. Thus $E(x)\ D(x\ y) \rightarrow A(F(x\ y))$ and $E(u)\ D(u\ y) \rightarrow A(F(u\ y))$ are variants; $E(x)\ D(x\ z) \rightarrow A(F(x\ z))$ and $E(u)\ D(y\ u) \rightarrow A(F(u\ y))$ are not variants. They are, however, separated.

In unification computations and in the resolution inference and proof constructions based on them, we routinely replace a clause by a suitably chosen one of its variants—for example, when we need to ensure that all clauses in a set are separated. As we shall soon see, however, there are ways of representing expressions (as two-dimensional structures of a certain kind) in which this becomes irrelevant and unnecessary because variables are nameless. The familiar one-dimensional notation, however, is the most convenient one for writing expressions, and it is in this representation that we have to be careful to avoid 'name-clashes' when choosing names for variables.

Atoms

Calling atomic sentences 'atoms' may run some risk of confusion with Lisp's usage of that word, but it is well established. There are two noncomposite atoms—the truth values **true, false**—but in general atoms are composite expressions, with two components: a *predicate* and a list of *arguments*. The usual convention for writing a composite atom is to write its predicate immediately before its argument list, as for example:

MOTHER(MARY, THOMAS)
GREATER-THAN(SUM-OF (THREE, SIX), SEVEN)

The predicates are *relational constants* MOTHER, GREATER-THAN, and so on: just identifiers, without any significant internal syntax of their own. In this discussion we will write them as upper-case identifiers. The arguments are *terms*. Noncomposite terms are variables: x, y, z, u_1, and so on. In this discussion we will write variables as lower-case identifiers, possibly subscripted.

Composite terms are like composite atoms in having two parts: an *operator* and list of arguments.[1] Indeed the common convention for writing a composite term is similar to that for writing composite atoms: to write the operator immediately before the list of arguments, as for example:

PLUS (THREE, SIX)
SUCCESSOR(SUCCESSOR (SUCCESSOR(ZERO))).

The operators are *functional constants*. PLUS, SUCCESSOR, and so on. When the argument list of a term is empty, we usually skip explicitly writing the empty list, and write the term as if it consisted of its constant alone, as MARY, THOMAS, instead of MARY( ), THOMAS( ). Every relational and functional constant comes with an *arity*, which is a nonnegative integer, and which is considered to be part of the constant's identity. A constant having arity n is said to be n-ary. Thus MARY is 0-ary, SUCCESSOR is 1-ary, GREATER-THAN is 2-ary, and so on. The basic formation rule for composite expressions (atoms or terms) is that an n-ary constant must always be

[1] In writing a list, we may place a comma after each item (other than the last) to enhance the readability. This is, however, optional, and is not part of the definition of a list.

followed immediately by a list of n arguments (except, as noted above, when n = 0, when the list can be by convention omitted). The common underlying semantic idea is that of an *applicative expression* which represents the result of *applying* a function or relation to a suitable tuple of arguments.

In the clausal predicate calculus, clauses are the only kind of sentence available in which to express the premises and desired conclusion of a proof problem. This is not as limiting as it sounds. It is in fact possible to translate (automatically) a proof problem from the full predicate calculus into the clausal predicate calculus. Detailed discussions of how to do this can be found, in [12].

Substitution

Making the clausal predicate calculus more machine-oriented calls for a much closer analysis of the idea of *instantiation*. When an expression B can be obtained from another expression A by substituting terms for some or all of the variables in A, B is said to be an *instance* of A.

For example, F(H(y z) G(H(y z) A(y)) A(y)) is an instance of F(x G(x y) y). Inspection confirms that F(H(y z) G(H(y z) A(y)) A(y)) can be obtained from F(x G(x y) y) by *simultaneously* replacing each occurrence of x and y by an occurrence of H(y z) and an occurrence of A(y) respectively. It is very interesting that this basic logical operation of *substitution* is essentially a parallel one.

We can represent specific substitutions by sets of equations. For example, the preceding substitution can be represented by the set {x = H(y z), y = A(y) }. Unspecified substitutions are usually denoted by lower-case Greek letters: θ, λ, μ, σ, and the result of applying a substitution to an expression E is indicated by writing Eθ, Eλ. Therefore, if E is F(x G)(x y) y) and θ is {x = H(y z), y = A(y)}, Eθ is F(H(y z) G(H(y z) A(y)) A(y)).

Unification

Let S be a set of expressions. When a substitution θ transforms every expression in S into the same expression, θ is said to *unify* S (or to be a *unifier* of S) and the set S is said to be *unifiable*.

For example, let θ be {x = H(P Q), y = D, u = P, v = Q, z = G(H(P Q),D)}. When we apply θ to the two expressions F(x G(x y)) and F(H(u v) z) both of them become the same expression, namely F(H(P Q) G(H(P Q) D)). Thus θ unifies the set {F(x G(x y)), F(H(u v) z)}.

This set, however, is also unified by the substitution σ = {x = H(u v), z = G(H(u v) y)}, which transforms both its members into the same expression: F(H(u v) G(H(u v) y)). This expression is not only a *more* general common instance of F(x G(x y)) and F(H(u v) z), but is actually a *most* general common instance, and so σ represents the most general way in which the set {F(x G(x y)), F(H(u v) z)} can be unified. We therefore say it is a *most general unifier* ('mgu') of {F(x G(x y)), F(H(u v) z)}. All other common instances of F(x G(x y)) and F(H(u v) z) are instances of the above most general one. In this particular case we have: F(H(u v) z)θ = F(x G(x y))θ = F(H(P Q) G(H(P Q) D)) = (F(x G(x y))σ)μ = F(H(u v) G(H(u v) y)μ where μ = {u = P, v = Q, y = D}. This suggests that θ is some kind of 'product' of the mgu σ and the substitution μ. We can write θ explicitly as $\theta = \sigma{\cdot}\mu$ and we find that, indeed, this notion of the product of two substitutions can be naturally defined and is extremely useful.

The *product* $\alpha{\cdot}\beta$ of two substitutions α and β is the overall substitution which results from first performing α and then performing β. Thus we have E($\alpha{\cdot}\beta$) = (Eα)β for all expressions E. This product operation is associative, and has an identity, namely the 'empty' substitution ϵ which leaves every variable unchanged. However, it is not in general commutative.

It is no accident that in our example we can express the unifier θ as the product of the mgu σ and the 'specialization' substitution μ = {u = P, v = Q y = D}. This is a defining characteristic of mgus.

In fact, to say that σ is an mgu of a set S is to make the following two statements: (1) that σ unifies S and (2) that any unifier λ of S whatsoever satisfies the condition: $\lambda = \sigma{\cdot}\mu$, for some μ.

A unifiable set always has an mgu. Moreover, there are simple algorithms (*unification* algorithms; about which we shall say more later) which *compute* an mgu for any finite unifiable set, and *detect* the non-unifiability of a set which is not unifiable. These algorithms are best stated for the more general case in which we seek a substitution that unifies several disjoint finite sets of expressions simultaneously (or, as we shall say, which unifies a *partition* of a set of expressions). It is the unification of partitions that we shall be concerned with in the remainder of the discussion. The idea is virtually the same as that of unifying a single set: a substitution θ unifies a partition T = {S_1, . . . , S_k} of a set S of expressions if each of the sets $S_1\theta$, . . . , $S_k\theta$ is a singleton. A substitution σ most generally unifies T if (1) σ unifies T and (2) for every unifier λ of T we have $\lambda = \sigma{\cdot}\mu$ for some μ.

We need to be able to compute a most general unifier efficiently, for any partition as input. There is now a rather large specialized literature on this topic, but for our present purposes we need not be concerned with many of the details.

Unification Algorithms

The computation of a most general unifier, when expressed in its most simple and natural form, is a highly parallel one. It was not at first seen to be so. The natural, inherent parallelism is most clearly seen if we think of expressions as being really directed labelled graphs, as follows:

- a variable is a graph with only one node, its root, which is unlabelled.
- a constant K is a graph with only one node, its root, which is la-

belled by the symbol K.

- an applicative expression $K(E_1, \ldots, E_n)$ is a graph whose root is unlabelled and has n + 1 out-arcs which are labelled respectively by the integers 0 to n. The out-arc labelled by 0 points to the node which is the constant K. For i = 1, . . . , n, the out-arc labelled i points to (the root of the graph which is) the term E_i.

If an out-arc goes from N to M and is labelled by j, we say that M is a *jth immediate successor* of N. The arity of a node is the largest integer which labels any of its out-arcs. So, for example, the expressions R(P G(x y) x y) and R(y z H(u K) u) are the two roots (nodes 1 and 2) of the graph in Figure 1, node 12 is a 2d immediate successor of node 10, and the arity of node 5 is 2. In all there are 13 expressions in the graph, one for each node. The graph itself can be thought of as representing the *set* of these expressions.

Note that in the graphical form of expressions we need no names for variables. Distinct variables are simply distinct unlabelled leaves (here, they are nodes 6, 7, 9 and 13, whose names in the linearly written expressions are respectively z, x, y and u). The use of the graphical form of expressions thus avoids the well-known complication of needing to rename variables in order to prevent unwanted identifications of two distinct variables which happen to have been given the same name.

Once we are given a set S of atoms and terms as a graph, we can represent a *partition* P of S by inserting one or more *links* (undirected arcs) between roots of distinct expressions which are in the same part of P. For example, by inserting a link between nodes 1 and 2 of the graph in Figure 1 we represent the 12-part partition

P = {{R(P G(x y) x y), R(y z H(u K) u)}, {G(x, y)}, {H(u, K)}, {R}, {P}, {G}, {H}, {K}, {x}, {y}, {z}, {u}}

by the graph in Figure 2.

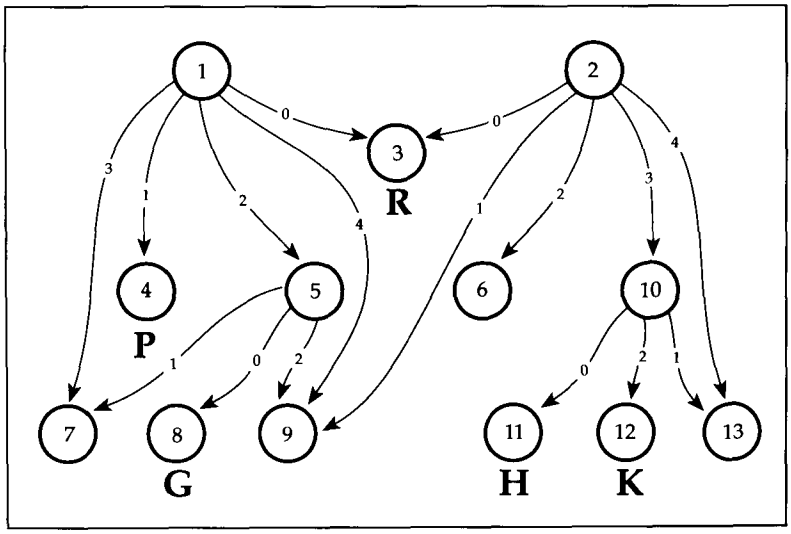


FIGURE 1.

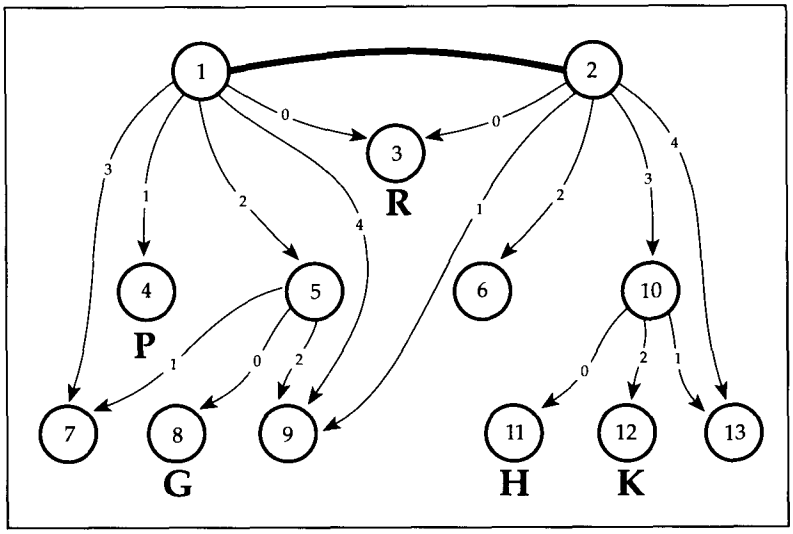


FIGURE 2.

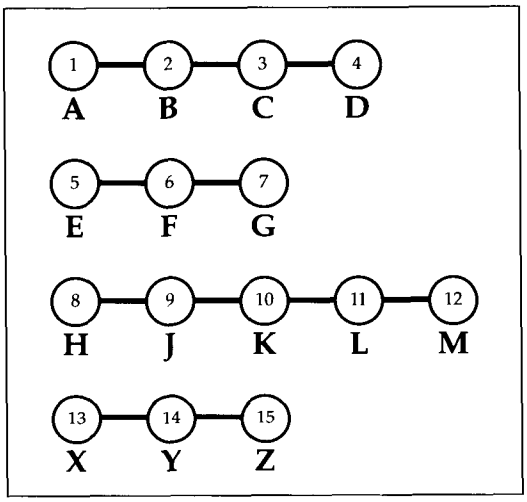


FIGURE 3.

If a part of a partition has more than two members, we do not need to put links between *every* two nodes in it. A part is represented by a *cluster* of nodes—a maximal set of nodes any two of which are *connected* by a *path* of such links.

For example, the six-part partition {{A, B, C, D}, {E, F, G}, {H, J, K, L, M}, {X}, {Y}, {Z}} of the set {A, B, C, D, E, F, G, H, J, K, L, M, X, Y, Z} is represented by the graph in Figure 3.

Given a partition in the form of a graph, the problem to find an mgu of the partition (or to detect its nonunifiability) is solved by the following *unification algorithm:*

while there are clusters in the graph but no clashes
do shrink the graph.

Shrinking a graph requires two steps:

Step 1. Each cluster C in the graph is "collapsed" into a single new node, which inherits all of the in-arcs, out-arcs, and labels of every node in C.

Step 2. New links are inserted between nodes which are equated by step 1.

Two nodes are *equated* if they are both jth successors, for some j, of the same node. A *clash* is a cluster in which there are nonvariable nodes which either (1) are labelled by distinct constants, or (2) are unlabelled, but have different arities.

Each iteration of the loop transforms a graph into another graph, which also in general contains links. For example, the first iteration transforms the graph in Figure 2 into the graph in Figure 4. The second iteration then transforms this into the graph in Figure 5 which is terminal, since there are now no links.

The process in general continues until an iteration either creates no new links, or else creates a clash; whereupon it terminates. This must eventually happen, since each iteration produces a new graph with fewer nodes than the previous graph. If, after termination, the graph contains no clashes and is acyclic, the original partition is unifiable. Otherwise, not.

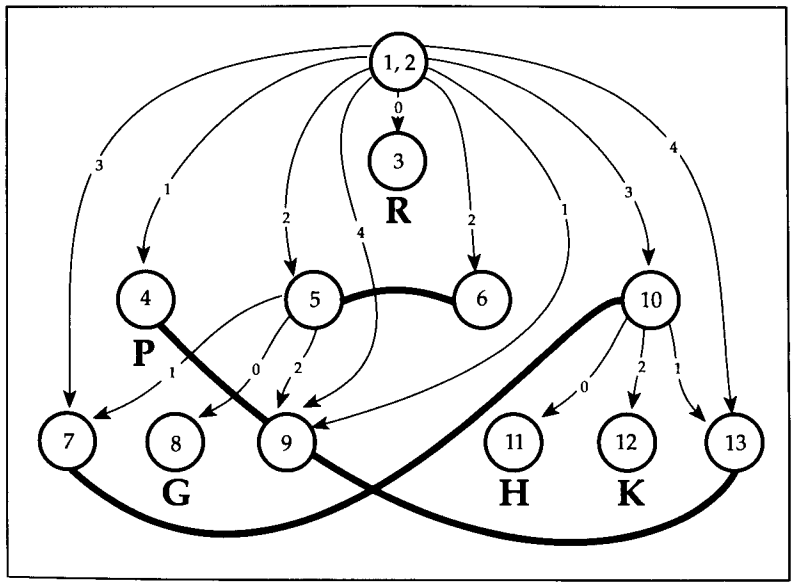


FIGURE 4.

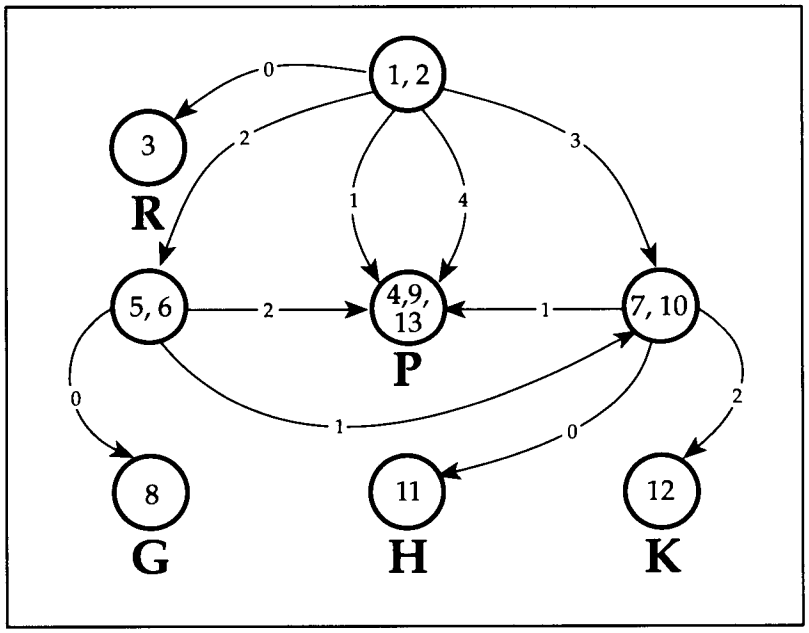


FIGURE 5.

On termination, an mgu for a unifiable partition can be found by comparing the terminal graph with the initial graph. For each node representing a variable in the original graph, we find the node in the terminal graph which contains it. The mgu is represented by equating each such variable with the expression represented by the corresponding node in the terminal graph.

Note that the nonterminal graphs generated during the process do not represent sets of expressions, since some of their nodes have more than one jth successor, for one or more j.

The graph-shrinking parallel unification algorithm is presented here in essentially the version that

was recently developed, analyzed and efficiently implemented in [2]. The elegant data-parallel SIMD implementation for the Connection Machine exploits all the inherent parallelism in the process very effectively.

The sequential version of this "fast unification" algorithm was hit upon independently by [4, 22, 42], improving an earlier formulation by [3]. As far as I know, the first version of a unification algorithm to be explicitly stated and accompanied by correctness and termination proofs was in [39].

Later, in [41], I formulated a more efficient version of the algorithm, using a tabular representation of the graph-representation to gain some of the same computational advantages which were brilliantly orchestrated on a much larger scale by [5] in their important *structure-sharing* resolution theorem-prover. This tabular representation [41] is also the point of departure for [2].

Herbrand's original (1930) version of the unification process is stated briefly, informally, and without proof (see [19]).

In 1984 [13] pointed out that in certain cases there is no opportunity for the parallel graph-shrinking algorithm to achieve any significant speed-up. Thus, for example, in finding the mgu {x = A} of the set

{F(F(F(F(F(F(F(x)))))))),
F(F(F(F(F(F(F(A)))))))}

we can merge only one pair of nodes, and generate only one new link, at each iteration of the loop. These successive minimal modifications of the graph therefore comprise essentially a sequential process. However, such 'worst cases' are more pathological than typical, and experience suggests that they are rarely met in real applications.

Resolution

Once we can compute an mgu for any unifiable partition of a set of expressions (or show the partition not to be unifiable, if that is the case), we are ready to make inferences by resolution.

The fundamental resolution inference pattern is closely related to what logicians call the 'cut' inference. (In Prolog programming parlance, unfortunately, the word 'cut' has come to have another, quite different, meaning). Cut inferences have the form:

from $A \rightarrow (B + \{L\})$ and
$(\{L\} + C) \rightarrow D$
infer $(A \cup C) \rightarrow (B \cup D)$.

We can make a cut inference from two clauses if any only if there is some atom L which is in the antecedent of one clause and the consequent of the other. To form the conclusion of the inference, we first 'cut' out L from both places, and then merge the two antecedents into one and two consequents into one. The 'disjoint union' notation $X + Y$ denotes the union $X \cup Y$, but also carries the further information that $X \cap Y = \varnothing$.

Example 1. From the clauses A B → C D and D E → F G we can infer the clause A B E → C F G by a cut, eliminating the atom D.

The resolution inference pattern generalizes the cut inference pattern by bringing in unification. The resolution inference pattern has the form:

from $A \rightarrow (B + M)$ and
$(N + C) \rightarrow D$
infer $(A \cup C)\sigma \rightarrow (B \cup D)\sigma$
where σ is an mgu of the one-part partition $\{M \cup N\}$.

In making a resolution inference, we must first use unification to deduce a pair of instances of the two premises suitable for a cut to be applied. In the special case that $M = N = \{L\}$, the mgu of the partition $\{M \cup N\}$ is the identity substitution. So in this case, a resolution is the same as a cut.

Example 2. From →P(G(r s) r s) and P(x y u)P(y z v)P(x v w) → P(u z w) we infer P(r z r) → P(s z s) by a resolution in which M = {P(G(r s) r s)} and N = {P(x y u),P(x v w)}, since {M ∪ N} is unifiable with mgu {x = G(r s), y = v = r, u = w = s}.

Example 3. From P(x y u)P(y z v)P(x v w) → P(u z w) and P(a b c)P(b d e)P(c d f) → P(a e f) we infer P(x y a)P(y b v)P(x v c)P(b d e)P(c d f) → P(a e f) by a resolution in which M = {P(u z w)} and N = {P(a b c)}, since {M ∪ N} is unifiable with mgu {u = a, z = b, w = c}.

From two given clauses, only a finite number of clauses can be inferred by resolution—one for each choice of the 'cut' sets M and N for which the partition {M ∪ N} is unifiable. If there are no such choices of M and N, then nothing can be inferred from the two clauses by resolution.

Resolution Deductions and Proofs

A *resolution deduction* is a finite tree whose nodes are labeled by clauses, each nonleaf node being labeled by a clause which is inferred by a resolution inference from the clauses labeling its immediate successors. The *conclusion* of the deduction is the clause labeling its root, and the *premises* of the deduction are the clauses labeling its leaves. A *resolution proof* is a resolution deduction whose conclusion is **false** (= the empty clause). Such a proof establishes that the premises are contradictory (unsatisfiable). If S is any unsatisfiable set of clauses there is always a resolution proof whose premises are all in S. This fact is the completeness of resolution (see [39].

A resolution proof with n + 1 premises can be taken in n + 1 different ways as a proof of the *negation* of one of its premises from the other n premises. For example, a resolution proof with premises A, B, C can be taken as (1) a proof of **not**-A from the premises B and C, (2) a proof of **not**-B from the premises A and C, and (3) a proof of **not**-C from the premises A and B.

P1-Resolution

A resolution *one of whose two premises*

is unconditional is called a *P1-resolution*. Example 2 is a P1-resolution, but Example 3 is not. It turns out that whenever a set of clauses is unsatisfiable, then there is a P1-resolution proof from those premises (see [40]). In other words, P1-resolution is also complete: despite its restricted form, P1-resolution is just as strong as resolution, but its proof-space is sparser than that of unrestricted resolution.

Hyper-Resolution

We get an even sparser proof-space when we take as the only inference rule, instead of the two-premise P1-resolution, the (p + 1)-premise *hyper-resolution* rule in which *exactly one* of the premises is a conditional clause and all of the other p premises, *together with the conclusion,* are unconditional clauses. The hyper-resolution rule is:

from $\rightarrow(C_1 + M_1), \ldots, \rightarrow(C_p + M_p)$, and $N_1 + \cdots + N_p \rightarrow D$
infer $\rightarrow(C_1 \cup \cdots \cup C_p \cup D)\sigma$
where σ is an mgu of the p-part partition $\{M_1 \cup N_1, \ldots, M_p \cup N_p\}$.

The unifiable p-part partition that is the essential ingredient of a hyperresolution is called its *kernel.* The p + 1 premises and the kernel together uniquely determine the conclusion.

A hyperresolution *inference* is really a compacted reorganization of a P1-resolution *deduction* whose conclusion is unconditional. After the reorganization the deduction has had all of its interior nodes suppressed and has become a single integrated transaction instead of a linked system of many transactions. By reorganizing the reasoning as a single inference, we are simply regarding its conclusion as having been obtained directly (or, to use a traditional logic expression, *immediately*—without any 'mediation') from its premises in one step, rather than 'mediately' as the eventual outcome of several linked P1-resolution steps.

Hyperresolution Deductions

A hyperresolution *deduction* is a finite tree each of whose nodes has a *label* and each of whose nonleaf nodes also has a *justification.* The labels are *unconditional* clauses, and the justifications are *conditional* clauses. The clause labeling a nonleaf node N is inferred by a hyperresolution whose unconditional premises are the clauses labeling the immediate successors of N, and whose conditional premise is the justification of N. The *conclusion* of the deduction is the clause labeling its root. The *premises* of the deduction are the labels of its leaf nodes and the justifications of its nonleaf nodes.

Hyperresolution deductions can yield only unconditional clauses. Moreover, they can yield only *positive* unconditional clauses, *unless the justification of the root node is a negative conditional clause* and in that case, but only in that case, the conclusion is a negative unconditional clause; indeed, it is the empty clause. Thus a hyperresolution deduction of the empty clause (a hyperresolution *proof*) always has *exactly one negative conditional clause* among its justifications. As we shall see, it is this feature which adumbrates logic programming.

Completeness and Local Finiteness of the Resolution Clausal Predicate Calculi

The resolution and hyperresolution versions of the clausal predicate calculus are all complete. Also, both systems are *locally finite.* This means that, in each system, there are only finitely many deductions of a given size (number of nodes) having a given set of premises (and this number is much smaller for hyperresolution than for resolution). By contrast, traditional predicate calculi are not even locally finite. This is one reason it is so difficult to make an efficient proof procedure for traditional predicate calculi. For example, most traditional predicate calculi contain the rule of *specialization:*

from $\forall A$ **infer** $\forall(A\theta)$,
where θ is any substitution.

(The sentence $\forall S$ is the *universal closure* of the sentence S: the result of prefixing a universal quantifier to S for every free variable in S). With this inference available, there are infinitely many deductions of size 2 which have the same premise $\forall A$—one for each different substitution θ.

Hyperresolution and Horn Clause Logic

The advantages of hyperresolution are quite striking in the Horn clause predicate calculus. In this subsystem of the clausal predicate calculus every clause is a *Horn clause,* namely, a clause having *at most one conclusion.* Hyperresolution then becomes much simpler. Recall the general definition of hyperresolution:

from $\rightarrow(C_1 + M_1), \ldots, \rightarrow(C_p + M_p)$, and $N_1 + \cdots + N_p \rightarrow D$
infer $\rightarrow(C_1 \cup \cdots \cup C_p \cup D)\sigma$
where σ is an mgu of the p-part partition $\{M_1 \cup N_1, \cdots, M_p \cup N_p\}$.

When all clauses are restricted to having at most one conclusion, the 'cut' sets M_i can only be singletons (say, $\{A_i\}$), and the 'remainder' sets C_i must be empty. Consequently, the definition of hyperresolutions *for Horn clauses* can be restated, in the following much simpler form:

from $\rightarrow A_1, \ldots, \rightarrow A_p$, $\{B_1, \ldots, B_p\} \rightarrow D$
infer $\rightarrow D\sigma$
where σ is an mgu of the p-part partition $\{\{A_1, B_1\}, \ldots, \{A_p, B_p\}\}$.

In this restatement of the rule, D and the A's and B's are all atomic sentences. When we combine hyperresolution inferences into multiinference deductions, we are in effect treating each particular application of this inference pattern

as though it were a special inference rule, 'the $\{B_1, \ldots, B_p\} \rightarrow D$ inference rule', stated as:

from $\rightarrow A_1, \ldots, \rightarrow A_p$
infer $\rightarrow D\sigma$
where σ is an mgu of the p-part partition $\{\{A_1, B_1\}, \ldots, \{A_p, B_p\}\}$.

This is, however, just a pragmatic device to sharpen our understanding of the very special role that conditional Horn clauses play in logic programming.

Ultraresolutions: Horn Clause Hyperresolution Deductions as Single Inferences

We again apply the idea of making a single inference out of an entire deduction. In the case of hyperresolution, instead of thinking of the conclusion of an entire deduction (namely a deduction built from P1-resolution steps and having an unconditional conclusion) as being arrived at stepwise by the performance of each of its inferences separately, we think of the whole construction as one inference step involving a higher and larger-scale inference pattern. We will now treat Horn clause hyperresolution deductions in a similar way, and thereby arrive at a higher- and larger-scale inference pattern which we call *ultraresolution.*

There is really no need, pragmatically, to know the conclusion of every individual inference in a hyperresolution deduction, if all that we are after is the eventual conclusion of the whole deduction. We can instead characterize that eventual conclusion more directly, by a relationship based only on the structure of the premises of the deduction. By omitting in this way all of the interior stepwise conclusions we turn the entire hyperresolution *deduction* into a single *inference,* which immediately yields its conclusion from the premises in one integrated step.

Ultraresolution

To every hyperresolution deduction D there corresponds an ultraresolution inference U which has the same premise and the same conclusion as D, and conversely. We define ultraresolution inferences directly, however, without reference to their corresponding hyperresolution deductions.

The ultraresolution rule is (where $A \rightarrow B$ is a Horn-clause and C is a set of Horn-clauses):

from $A \rightarrow B$ and C
infer $\rightarrow B\sigma$
if there is a cover of $A \rightarrow B$ by C whose kernel is unifiable with mgu σ.

The clause $A \rightarrow B$ is the *main premise* and the clauses in C are the *covering premises.*

Covers and Their Kernels

A *cover* of a clause $A \rightarrow B$ by a set C of clauses is a certain kind of finite tree with nodes labeled by clauses. The root of the tree is labeled by $A \rightarrow B$, while the other nodes are labeled by variants of clauses in C. The extra condition that makes the tree a cover is that for each node N in the tree, every atom in the antecedent of the clause labeling N is assigned to a distinct immediate successor of N. The *kernel* of the cover is the partition:
{{X, Y}|Y is the conclusion of the clause labelling the node to which X is assigned}.

Example 4. To illustrate the notions of a cover and its kernel, consider the clause:

$$A(x_0)\ B(x_0) \rightarrow C(x_0)$$

and the set C of clauses

$\{E(x_1)\ D(x_1\ y_1) \rightarrow A(F(x_1\ y_1)),$
$H(G(x_2)) \rightarrow B(x_2),$
$\rightarrow H(G(x_3)), \rightarrow D(M\ N), \rightarrow E(M)\}.$

The labeled tree given by the table:

node	parent	label
1	—	$A(x_0)\ B(x_0) \rightarrow C(x_0)$
2	1	$E(x_1)\ D(x_1\ y_1) \rightarrow A(F(x_1\ y_1))$
3	1	$H(G(x_2)) \rightarrow B(x_2)$
4	3	$\rightarrow H(G(x_3))$
5	2	$\rightarrow D(M\ N),$
6	2	$\rightarrow E(M)$

is a cover of $A(x_0)\ B(x_0) \rightarrow C(x_0)$ by C, in view of the assignments given by the table:

atom	assigned to node
$A(x_0)$	2
$B(x_0)$	3
$H(G(x_2))$	4
$D(x_1\ y_1)$	5
$E(x_1)$	6

and has the following partition as its kernel:

$\{\{A(x_0), A(F(x_1\ y_1))\}, \{B(x_0, B(x_2)\},$
$\{E(x_1), E(M)\}, \{D(x_1\ y_1), D(M\ N)\},$
$\{H(G(x_2)), H(G(x_3))\}.$

Since this kernel is unifiable, with mgu

$$\sigma = \{x_0 = x_2 = x_3 = F(M\ N)),\ x_1 = M,\ y_1 = N\},$$

we can infer the clause

$$\rightarrow C(x_0)\sigma = \rightarrow C(F(M\ N))$$

by an ultraresolution which has $A(x_0)\ B(x_0) \rightarrow C(x_0)$ as its main premise and C as its set of covering premises.

The intuition behind the notion of a cover of a clause $A \rightarrow B$ is that it depicts *exactly* the pattern of organization of the given clauses. If the kernel of the cover is unifiable with mgu σ, it guarantees that we can easily relabel the tree so it turns into a hyperresolution *deduction,* from these clauses as premises, of the same unconditional clause $\rightarrow B\sigma$ that the ultraresolution *inference* obtains directly from them in one step. In this relabeling, the new label on each leaf node of the tree is the same as the old label. The old

label on each nonleaf node of the cover, however, is removed (it now becomes the *justification* of the hyperresolution inference at that same node), and the node's new label is the unconditional clause which is inferred by a hyperresolution from this justification clause together with the new labels on the immediate successors of the node. The following example illustrates the relationship between a hyperresolution deduction and the corresponding ultraresolution inference.

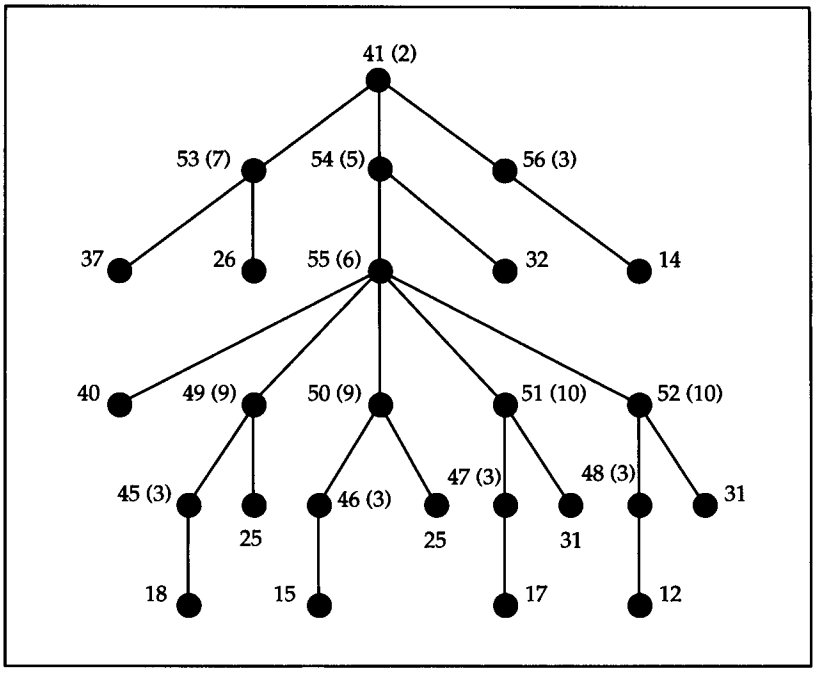


FIGURE 6.

Example 5. Figure 6 is a hyperresolution deduction of the unconditional clause UNCLE(TED ANN)← from a subset of the following set of Horn clauses, which comprises a small 'family relationship' knowledge base. This knowledge base contains (as its 'definitions') the following conditional Horn clauses:

```
 1 UNCLE(u x)      ←BROTHER(u y) PARENT(y x)
 2 UNCLE(u x)      ←HUSBAND(u s) SISTER(s p) PARENT(p x)
 3 PARENT(x,y)     ←CHILD(y,x)
 4 BROTHER(b x)    ←SIBLING(b x) MALE(b)
 5 SISTER(s x)     ←SIBLING(s x) FEMALE(s)
 6 SIBLING(x y)    ←DIFFERENT(x y) FATHER(f x) FATHER(f y) MOTHER(m x) MOTHER(m y)
 7 HUSBAND(h w)    ←MARRIED(h w) MALE(h)
 8 WIFE(w h)       ←MARRIED(h w) FEMALE (w)
 9 FATHER(f x)     ←PARENT(f x) MALE(f)
10 MOTHER(m x)     ←PARENT(m x) FEMALE(m)
```

and (as its 'facts') the following unconditional Horn-clauses:

```
11 CHILD(JIM JOE)←    15 CHILD(JOE TOM)←    19 CHILD(TOD PAT)←
12 CHILD(JOE MEG)←    16 CHILD(ANN SUE)←    20 CHILD(RON PAT)←
13 CHILD(JIM SUE)←    17 CHILD(PAT MEG)←    21 CHILD(TOD TED)←
14 CHILD(ANN JOE)←    18 CHILD(PAT TOM)←    22 CHILD(RON TED)←

23 MALE(JIM)←         29 FEMALE(ANN)←       35 MARRIED(TOM MEG)←
24 MALE(JOE)←         30 FEMALE(SUE)←       36 MARRIED(JOE SUE)←
25 MALE(TOM)←         31 FEMALE(MEG)←       37 MARRIED(TED PAT)←
26 MALE(TED)←         32 FEMALE(PAT)←       38 MARRIED(RON SAL)←
27 MALE(TOD)←         33 FEMALE(SAL)←       39 MARRIED(JIM JAN)←
```

40 DIFFERENT(a b)← a ≠ b & a, b, ∈ {JIM, JOE, TOM, TED, TOD, RON, ANN, SUE, MEG, PAT, SAL, JAN}

Premise 40 is a 'virtual' definition: it is simply a shorthand way of supplying 132 facts (such as DIFFERENT(JOE ANN)←) whose predicate is DIFFERENT and whose two arguments are distinct constants in the displayed set.

From this knowledge base there are, for example, hyperresolution deductions of each of the following unconditional clauses:

41 UNCLE(TED ANN)← 45 PARENT(TOM PAT)← 49 FATHER(TOM PAT)← 53 HUSBAND(TED PAT)
42 UNCLE(TED JIM)← 46 PARENT(TOM JOE)← 50 FATHER(TOM JOE)← 54 SISTER(PAT JOE)←
43 UNCLE(JOE TOD)← 47 PARENT(MET PAT)← 51 MOTHER(MEG PAT)← 55 SIBLING(PAT JOE)←
44 UNCLE(JOE RON)← 48 PARENT(MEG JOE)← 52 MOTHER(MEG JOE)← 56 PARENT(JOE ANN)←

For example, clause 41, UNCLE (TED ANN)←, is the conclusion of the hyperresolution deduction shown in Figure 6. The label on each node is given in the diagram by its number next to the node, and at each nonleaf node is followed by the number, in parentheses, of the clause which is the justification of the node.

The cover of the ultraresolution inference corresponding to this hyperresolution deduction is shown in Figure 7.

Figure 9 displays the cover of this ultraresolution inference in more detail, and shows more clearly that its status as an inference is conceptually independent of the corresponding hyperresolution deduction. In Figure 9, each labeled node of the cover is represented by a box of one of the three types shown in Figure 8. These represent a node labeled respectively by a positive conditional clause $Q \leftarrow P_1 \ldots P_n$, by a negative conditional clause $\leftarrow P_1 \ldots P_n$, and by a positive unconditional clause $Q\leftarrow$.

The thick lines in Figure 9 show the pairs of the unifiable kernel of the cover.

That this kernel *is* unifiable is verified by an easy computation. Its mgu σ is:

{a0 = u1 = h2 = TED,
b0 = x1 = y8 = ANN,
s1 = w2 = s3 = x4 = x5
= y7 = x9 = y11 = PAT,
p1 = x3 = y4 = x6 = x8
= x10 = y12 = y13 = JOE,
f4 = f5 = f6 = x7 = x13 = TOM,
m4 = m9 = m10
= x11 = x12 = MEG}.

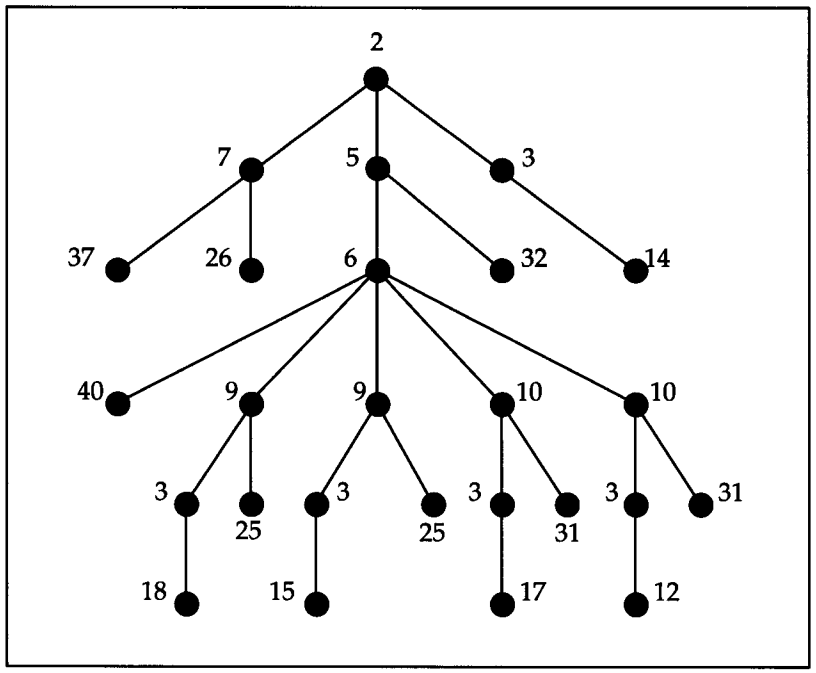


FIGURE 7.

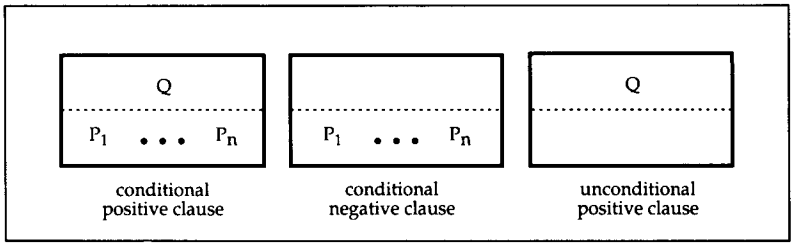


FIGURE 8.

and applying σ to the conclusion of the root clause yields UNCLE(TED ANN)←.

Queries and Their Answers Logic Programming

We can consider any collection K of positive Horn clauses as a knowledge base. A set of positive Horn clauses is necessarily consistent: one cannot deduce **false** from it by hyperresolution (or what is the same, one cannot infer **false** from it by an ultraresolution) if it contains no negative conditional clause. By taking a negative clause **not**-Q as the premise together with a collection of variants of clauses from K, we may be able to infer **false** by an ultraresolution. That is, the set {**not**Q} ∪ K may well be inconsistent and its inconsistency demonstrated by our inference. We then can turn this inconsistency to our advantage, by regarding **not**-Q as the *negation* of a query Q that we want answered, and digging out the answer from the details of the

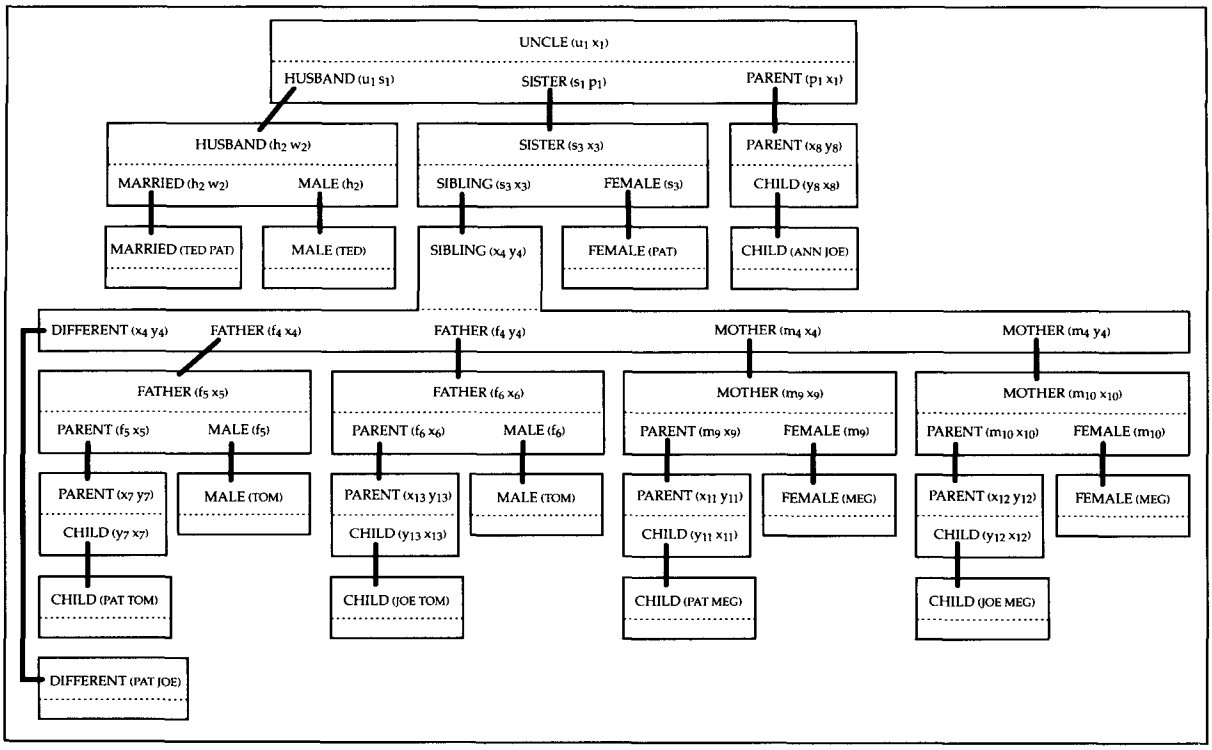


FIGURE 9.

ultraresolution.

Suppose that **not**-Q is the negative clause $\leftarrow\{G_1, \ldots, G_n\}$. Recall that we can read $\leftarrow\{G_1, \ldots, G_n\}$ as:

not $\exists x_1 \ldots \exists x_m\ G_1$ **and** $\ldots$ **and** G_n

where $x_1 \ldots x_m$ are all of the variables occurring in the atoms $G_1, \ldots, G_n$. Then Q is

$\exists x_1 \ldots \exists x_m\ G_1$ **and** $\ldots$ **and** G_n

and so an inference of **false** from {**not**-Q} ∪ K is an inference of Q from K. The usefulness of this fact for logic programming is that the mgu σ of the kernel of the inference can be used to supply directly the 'answer' $\langle x_1 \ldots x_m\rangle = \langle x_1 \ldots x_m\rangle\sigma$ to the 'query' $\exists x_1 \ldots \exists x_m\ G_1$ **and** $\ldots$ **and** G_n.

There may be many different ultraresolution inferences of **false** which have the same negative clause as the main premise and whose covering premises are taken from the same knowledge base K. It is even possible that the covering premises also are the same, with only the underlying cover and kernel being different. In any case, these different inferences will, in general, have different covers and kernels, and will therefore provide different answers from K *to the same query* Q.

To find *all* these answers, what is needed is a suitable way of finding *all* the different ultraresolution inferences of **false** whose main premise is the negative clause **not**-Q and whose covering premises are variants of clauses in K.

LUSH, Alias SLD, Resolution

The original Edinburgh solution to this tricky computational problem was simple and beautiful, and it led directly to Prolog. After much exploration, [27] devised a 'linear' binary resolution inference pattern which they called SL-resolution (for *S*elective *L*inear resolution). When restricted to Horn clauses, SL resolution becomes—as [1] named it—SLD-resolution (for *S*elective *L*inear resolution for *D*efinite clauses). A definite clause is (simply another name for) a positive Horn clause. However, [21] had already, in 1974, coined a more whimsical name for it: LUSH resolution, for *L*inear resolution with *U*nrestricted *S*election function, for *H*orn clauses. It is not clear to me why [1] felt this name was unsuitable. Whatever we call it, this highly specialized and narrowly restricted resolution inference has the form[2]:

from $A \rightarrow B$ and $G \rightarrow H$
infer $(A \cup \downarrow G)\sigma \rightarrow H\sigma$
if σ is a most general unifier of the 1-part partition $\{\{B, \uparrow G\}\}$.

The clause $G \rightarrow H$ is the *main premise* of the inference, and the clause $A \rightarrow B$ is the *side premise*.

The novel feature of this inference rule is its use of the two functions, *selection* (↑) and *remainder* (↓), both of which operate on the set G of conditions of the main premise. The function ↑ yields *the condition which is selected,* while the function ↓

[2]Actually, in the original version and the version contained in the logic programming literature, the conclusion H of the main premiss is omitted, and thus the main premiss is always a *negative* Horn-clause. Here, for various reasons, one of which will shortly become evident, we permit the main premiss to have a conclusion. In addition to its role as the 'answer template' in logic programming computations, the conclusion can be put to other good uses.

yields *the set of conditions which are not selected.* Thus *at most one* LUSH/SLD inference is possible from a given main premise and side premise, and its conclusion *is unique to those two premises.* Hence a LUSH/SLD deduction will necessarily have a linear structure, in which each successive LUSH/SLD resolution will have for its main premise the conclusion of the previous one.

The really interesting and useful, and at first acquaintance amazing property of LUSH/SLD resolution is that *the choice of the selection and remainder functions is completely unrestricted* (whence the 'U' in the name 'LUSH'—more's the pity that the name 'SLD' lacks any acronymic reference to this feature). Thus, in particular, the selection and remainder functions can be chosen so as to make the sets of conditions in the successive main premises behave like a stack, provided we take seriously the order in which the conditions are written, and always form the conclusion by adjoining the new conditions, if any, on the *left* of the remainder, in *their* written order. The selection then yields the leftmost condition (the one at the 'top' of the 'stack'). A LUSH/SLD deduction then does indeed look very much like the trace of a stack-oriented 'computation'.

To compute *all* the answers to a given query $\exists x_1 \ldots \exists x_m$ (G_1 **and** $\ldots$ **and** G_n), we initialize the state of the computation to the 'state'

$$Q_0$$

setting it up to be the clause[3]

$$Q_0 = \text{ANSWER}(x_1 \ldots x_m) \leftarrow G_1 \ldots G_n$$

whose antecedent consists of the initial set of 'goals' and whose conclusion is a special 'system' atom $\text{ANSWER}(x_1 \ldots x_m)$ acting as the formal 'answer template'. We then begin a series of computation steps, each of which is a single LUSH/SLD resolution inference. In general the $(t + 1)$st step transforms the t^{th} state Q_t by using it as the main premise, and a variant of one of the clauses from the knowledge base (or 'program') as the side premise, to make a LUSH/SLD inference whose conclusion is the $(t + 1)^{st}$ state Q_{t+1}. A state is *terminal* if it is an unconditional clause.

Thus each complete *computation* is a LUSH/SLD resolution *proof* of an unconditional clause: $\text{ANSWER}(t_1 \ldots t_m) \leftarrow$, thereby providing the computation with the *answer* $\langle t_1 \ldots t_m \rangle$ as its output. The *different* possible computations are related as the branches of a tree—*the LUSH/SLD computation tree*—since after any step there is in general *more than one choice of positive clause to take as the side premise for the next step.* Each nonterminal state of the computation will in general, therefore, have more than one successor state. It is the *complete* tree of *all possible* computations for the given query which is the total 'internal' response of the logic programming engine to that query; but its 'external' response is simply (some representation of) the *set of all answers* to the query.

Example 5 (continued). The family knowledge base of Example 5 contains enough information to provide four different answers ⟨a b⟩ to the query ∃abUNCLE(a, b), namely: ⟨TED ANN⟩, ⟨TED JIM⟩, ⟨JOE TOD⟩, ⟨JOE RON⟩. This corresponds to the fact that the four unconditional clauses UNCLE(TED ANN)←, UNCLE(TED JIM)←, UNCLE(JOE TOD)← and UNCLE(JOE RON)← can all be deduced by hyperresolution, or, equivalently, inferred directly by an ultraresolution, from the knowledge base.

What is so beautiful about this Edinburgh scheme is that it turns out that the *branches* of the LUSH/SLD computation tree correspond, one-to-one, to all the different ultraresolution *inferences* whose main premise is the initial state of the computation. The entire tree of LUSH/SLD computations is thus a complete survey of all possible ultraresolution *inferences* from that premise and the given knowledge base.

This correspondence now makes it obvious why the selection/remainder functions are unrestricted. Once we see clearly that each LUSH/SLD proof is simply a node-by-node 'top down' or 'backward-chaining' construction of the cover of an ultraresolution inference, starting with the antecedent of its main premise, we can interpret each LUSH/SLD step as a further small increment in that construction. Since the node chosen by the LUSH/SLD selection function as the site of the next increment of constuction is obviously arbitrary,

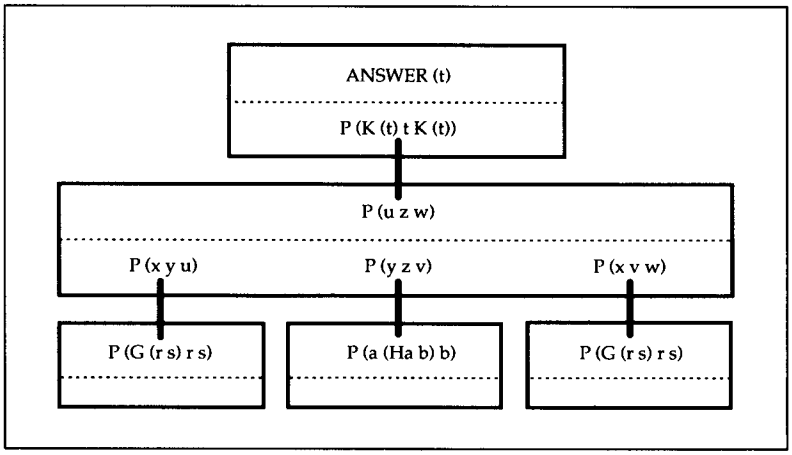


FIGURE 10.

[3]The idea of using a formal answer template in this way was originated by Cordell Green in the QA systems described earlier in this essay.

there is no restriction on what that selection function is taken to be.

Serial vs. Parallel Computation in Logic Programming

The branches of the LUSH/SLD computation tree in the first logic programming (Prolog) systems were generated serially, in a depth-first, backtracking search. This tree-search method is subject to embarrassing 'depth-first' infinite runaways when nonterminating branches are present in the tree, but it is otherwise a simple, natural and effective way to search the complete LUSH/SLD computation tree and thus find the set of all answers to a query. The answers will be generated one at a time, as each terminal state is encountered. If a query has infinitely many answers (and the search tree therefore has infinitely many branches), then the set of all answers will simply (and correctly, in a reasonable sense) be generated as a nonterminating sequence.

It is surely clear, however, that the branches of the LUSH/SLD computation tree need not be constructed one at a time in this depth-first back-tracking manner. One can instead grow the tree breadth-first, with no back-tracking, by computing successive *sets* of states, starting with the singleton set $\{Q_0\}$, and continuing, in general, by computing the $(t + 1)^{st}$ set as the set of *all* the immediate successors of *all* the states in the t^{th} set. The different completed computations, together with their associated answers, will be harvested, at each level, as their corresponding terminal states turn up in these state sets. There is, of course, no logical significance to the *order* in which these answers are generated: the answers logically form a set, not a sequence.

It is easy to find examples of queries which have infinitely many answers. For example, if the knowledge base is the set of clauses:

{NUMBER(0)←, NUMBER(S(x))
←NUMBER(x)}

then the query ∃x NUMBER(x) has the set of answers:

{x = 0, x = S(0), x = S(S(0)), . . . , }.

Each answer comes from an ultra-resolution inference whose main premise is:

ANSWER(x)←NUMBER(x).

All these answers are given by covers which exhibit the same general pattern. The LUSH/SLD computation tree is an infinite binary tree which has only two states at each nonzero depth t. One of these two states is the clause

ANSWER(S(S(. . . 0 . . .)))←

with t occurrences of 's', and produces the answer

x = S(S(. . . 0 . . .));

the other is the clause

ANSWER(S(S(. . . S(x) . . .)))
←NUMBER(x)

with t + 1 occurrences of 's', which has two successors, and so on.

General Queries and Answers

Queries can contain universally quantified variables, and so can their answers. Consider, for example, the knowledge base:

P(u z w)←P(x y u)P(y z v)P(x v w)
P(x v w)←P(x y u)P(y z v)P(u z w)
P(G(a b) a b)←
P(a H(a b) b)←.

and the query: ∃ t ∀k P(k t k). The negation of the query is: ∀t ∃k **not** P(k t k), so (as explained, for example, in [12]) in order to have a clause we must eliminate the existential quantifier. This is done by introducing a 'Skolem term' K(t) in place of the existential variable. The negated query then is: ∀t **not** P(K(t) t K(t)), or in other words the negative clause: ←P(K(t) t K(t)). Thus the initial state for the LUSH/SLD computation is the clause: ANSWER(t)←P(K(t) t K(t)).

An intuitive way to understand the clauses of this knowledge base is to interpret their variables as ranging over the elements of some set which is closed under a binary composition operation ·, and to interpret atoms P(a b c) as saying that a·b = c. The first two clauses then together assert that · is associative. The third says that the equation x·a = b always has the solution x = G(a b), while the fourth says that the equation a·x = b always has the solution x = H(a b). The query is then seen to be asking whether there is a t such that for all k, k·t = k, that is, *whether there is a right identity element.*

The kernel of the cover shown in Figure 10 is unifiable and has the mgu

σ = {t = z = H(y y), x =
G(y K(H(y y))), a = b = r = v = y,
s = u = w = K(H(y y))}

and so the inference yields the unconditional clause ANSWER(H(y y))← containing the universally quantified variable 'y'. In effect, the response to the query *are there right identity elements?* is the general proposition: *yes—for all* y, H(y y) *is a right identity element.*

Parallelism in Ultraresolution Inferences

The potential parallelism in the breadth-first growth of the LUSH/SLD tree is the kind which has come to be known as *or-parallelism.* Since each state may have several immediate-successor states it corresponds to the fact that there may be alternative possible choices of a positive clause as side premise for that state as main premise. As we have seen, the classical LUSH/SLD search (in its breadth-first version) is a clever way to compute all possible ultraresolution inferences which have a given conditional clause Q as main premise, with covering premises taken from a given fixed knowledge base P. So the or-parallel version of the LUSH/SLD process is a way of exploiting at least some of the potential parallelism of the ultraresolution inference scheme.

The challenge to the software and hardware designers of future logic programming systems, however, comes from the clear percep-

tion that there is more potential parallelism 'waiting there' than just the or-parallelism. The computation of the set of all ultraresolutions with main premise Q and covering premises in P is abstractly just a matter of generating all covers of Q and then checking the kernel of each to see if it is unifiable. This, however, is precluded by a 'combinatorial explosion' problem. There are simply too many covers. Even in the small family knowledge base we considered earlier, there are (as we noted) only four ultraresolutions with the main premise

ANSWER(a b)←UNCLE(a b)

and covering premises in the knowledge base. This means that there are only four covers of its antecedent *whose kernels are unifiable.* There are, however, *several billion* covers of this antecedent whose kernels are *not* unifiable. Despite the large size of the space to be searched in this simple example, a breadth-first (quasi-or-parallel) LUSH/SLD computation generates a tree of about 140 states, level by level down to a depth of about 25, in order to produce all four answers and to show that there are no more. The power of the LUSH/SLD search method rests in the fact that entire subtrees of these 'failures' are constantly being eliminated from the search. Its *incremental* unification process in effect detects a source of nonunifiability *as soon as it appears* and therefore never permits a partially grown cover containing *that* 'lethal gene' to 'breed' any progeny at all. Thus the LUSH/SLD pruning of the tree is as drastic as it can be. Delaying *any* of this failure detection 'until later' will only allow these sources of failure to propagate multiplicatively, so that the future computational cost (whether in the extent of time consumed or in the number of parallel resources needed) of detecting all of them will grow at the same rate. Postponing *all* of the unification analysis until the generation of the set of all covers is completed simply lets this effect maximize itself.

Here, however, it is clear that we have arrived at a point where merely logical considerations must yield the center stage to highly technical questions of algorithm design, complexity analysis, and parallel computation, the discussion of which is outside the scope of this article.

Glimpses Beyond

In this article I have discussed only the historical and conceptual background of the logical origins of logic programming. I have concentrated on the resolution theorem-proving ideas which have been my main interest from 1960 until the present. In describing its development to the present, I have briefly sketched the overall framework within which today's specialists are seeking to exploit as much as possible of the potential parallelism which is clearly present in the fundamental processes. The rest of that story is now better left for others, more qualified, to tell.

Acknowledgments

I am grateful to Jacques Cohen, Jack Minker and Jonas Barklund for their excellent suggestions after reading an earlier version of this article. I have tried to follow all of them.

References

1. Apt, K.R. and van Emeden, M.H. *Contributions to the Theory of Logic Programming. J. Assoc. Comput. Mach. 29* (1982), 841–862.
2. Barklund, J. Parallel unification. Ph.D. thesis, Computing Science Department, Uppsala University, 1990.
3. Baxter, L.D. An efficient unification algorithm. Tech. Rep. CS-73-23, Department of Computer Science, University of Waterloo, 1973.
4. Baxter, L.D. The complexity of unification. Ph.D. thesis, University of Waterloo, 1976.
5. Boyer, R.S. and Moore, J.S. The Sharing of Structure in Theorem Proving Programs. *Mach. Intell. 7* (1972), 101–116.
6. Church, A. A note on the *entscheidungsproblem. J. Symbolic Logic 1* (1936) 40–41. (Reprinted in [11]).
7. Colmerauer, A. Les Systemes-Q ou un formalisme pour analyser et synthesizer des phrases sur ordinateur. Rep. 43, Department of Computer Science, University of Montreal, 1970.
8. Colmerauer, A., Kanoui, H., Pasero, R. and Roussel, P. Un systeme de communication homme-machine en Français. Tech. Rep., Groupe d'Intelligence Artificielle, Université d'Aix Marseille II, Luminy, France, 1973.
9. Davidon, W. Personal communication, 1962.
10. Davis, M. Eliminating the irrelevant from mechanical proofs. In *Proceedings, Symposia of Applied Mathematics 15,* American Mathematical Society, 1963, pp. 15–30. (Reprinted in [44], vol. 1, 315–330).
11. Davis, M., Ed. *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems, and Computable Functions.* Raven Press, 1965.
12. Davis, M. and Putnam, H. A computing procedure for quantification theory. *J. Assoc. Comput. Mach. 7* (1960), 201–216. (Reprinted in [44], vol. 1, 125–139).
13. Dwork, C, Kanellakis, P., and Mitchell, J.C. On the sequential nature of unification. *J. Logic Program. 1,* (1984), 35–50.
14. Frege, G. *Begriffsschrifft, a Formula Language, Modelled Upon that of Arithmetic, for Pure Thought.* English translation in [48], 1–82.
15. Gentzen, G. *The Collected Papers of Gerhard Gentzen.* M.E. Szabo Ed., North-Holland, 1969.
16. Gilmore, P.C. A Proof Method for Quantification Theory: its Justification and Realization. *IBM J. Res. Dev. 4* (1960), 28–35. (Reprinted in [44], volume 1, 151–158).
17. Gödel, K. *The Completeness of the Axioms of the Functional Calculus of Logic,* 1930. English translation, with commentary, in [48], 582–291.
18. Green, C.C. The application of theorem-proving to problem solving. In *Proceedings of the first International Joint Conference on Artificial Intelligence* (Washington, D.C., 1969), pp. 219–240.
19. Herbrand, J. *Investigations in Proof Theory* (1930). English translation of main parts, with commentary, in [48], 525–581, and of entire thesis in *Jacques Herbrand: Logical Writings*

(edited by Warren Goldfarb), Harvard, 1971, 44–202.

20. Hewitt, C. *PLANNER: A language for proving theorems in robots.* In *Proceedings of the first International Joint Conference on Artificial Intelligence,* (Wash., D.C., 1969), pp. 295–301.
21. Hill, R. LUSH Resolution and its Completeness. DCL Mem. 78, Department of Artificial Intelligence, University of Edinburgh, 1974.
22. Huet, G. *Resolution des equations dans langages d'order 1, 2, . . . ω.* These d'Etat, Universite Paris VII, 1976.
23. Kneale, W.C. and Kneale, M. *The Development of Logic.* Oxford, 1962.
24. Kowalski, R.A. Predicate Calculus as a Programming Language. In *Proceedings of Sixth IFIP Congress,* North Holland, 1974, pp. 569–574.
25. Kowalski, R.A. *Logic for Problem Solving.* North-Holland, 1979.
26. Kowalski, R.A. The early years of logic programming. *Commun. ACM 31* (1988), 38–43.
27. Kowalski, R.A. and Kuehner, D. *Linear Resolution with Selection Function.* Artificial Intelligence 2 (1971), 227–260. (Reprinted in [44], volume 1, 542–577).
28. Loveland, D.W. Mechanical theorem proving by model elimination. *J. ACM 15* (1968), 236–251. (Reprinted in [44], volume 2, 117–134).
29. Löwenheim, L. *On Possibilities in the Calculus of Relatives,* 1915. English translation in [48], 228–251.
30. Luckham, D. *Refinement Theorems in Resolution Theory. Lecture Notes in Mathematics 125,* Springer, 1970, 163–190.
31. McCarthy, J. Programs with Common Sense. In *Proceedings of a Symposium on the Mechanization of Thought.* H.M. Stationery Office, London, 1959. (Reprinted in *Semantic Information Processing* MIT Press, 1968).
32. Minsky, M. *The Society of Mind.* Simon and Schuster, 1985.
33. Newell, A., Shaw, J.C. and Simon, H.A. Empirical explorations with the logic theory machine: A case study in heuristics. In *Proceedings of the Western Joint Computer Conference* (1957), pp. 218–239. (Reprinted in [44], volume 1, 49–73).
34. Prawitz, D. *An Improved Proof Procedure.* Theoria 26 (1960) 102–139. (Reprinted in [44], volume 1, 159–199, with a preface by the author).
35. Prawitz, D., Prawitz, H. and Voghera, N. A mechanical proof procedure and its realization in an electronic computer. *JACM 7* (1960) 102–128. (Reprinted in [44], volume 1, 200–228).
36. Raphael, B. Programming a robot. In *Proceedings of Fourth IFIP Congress,* North Holland, 1968, 135–139.
37. Robinson, A. Proving a theorem (as Done by Man, Logician, or Machine). Summaries of talks presented at the Summer Institute for Symbolic Logic. Communications Research Division, Institute for Defense Analysis, Princeton, 1957. (Reprinted in [44], volume 1, 74–76).
38. Robinson, J.A. Theorem proving on the computer. *J. Asso. Comput. Mach. 10* (1963) 163–174. (Reprinted in [44], volume 1, 372–383).
39. Robinson, J.A. A machine-oriented logic based on the resolution principle. *JACM 12* (1965), 23–41. (Reprinted in [44], volume 1, 397–415).
40. Robinson, J.A. Automatic Deduction with Hyper-resolution. *Int. J. Comput. Math. 1* (1965), 227–234. (Reprinted in [44], volume 1, 416–423).
41. Robinson, J.A. Computational logic: the unification computation. *Machine Intell. 6* (1970), 63–72.
42. Robinson, J.A. *Fast Unification.* Tagung über Automatisches Beweisen, Mathematisches Forschungsinstitut Oberwolfach, 1976.
43. Robinson, G.A. and Wos, L.T. Paramodulation and theorem proving in first-order theories with equality. *Machine Intell. 4* (1969), 103–133.
44. Siekmann, J.H. and Wrightson, G. Eds. *Automation of Reasoning. Classical Papers on Computational Logic (1957–1966).* Two volumes, Springer, 1983.
45. Skolem, T. *Logico-combinatorial Investigations in the Satisfiability or Provability of Mathematical Propositions* (1920). English translation, with commentary, in [48], 252–263.
46. Tarski, A. *Logic, Semantics, Metamathematics: Papers from 1923 to 1938,* translated by J.H. Woodger, Oxford, 1956.
47. Turing, A.M. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society,* 1937. (Reprinted in [11]).
48. van Heijenoort, J. Ed. *From Frege to Godel; A Source Book in Mathematical Logic, 1879–1931.* Harvard University Press, 1967.
49. Wang, H. Towards mechanical mathematics. *IBM J. Res. Dev. 4* (1960), 2–22. (Reprinted in [44], volume 1, 244-264).
50. Warren, D.H.D. *Implementing PROLOG*—compiling predicate logic programs (Res. Rep. 39 and 40), and Logic programming and compiler writing (Res. Rep. 44). Department of Artificial Intelligence, University of Edinburgh, 1977.
51. Winograd, T. *Understanding Natural Language.* Academic Press, 1973.
52. Wos, L.T., Carson, D. and Robinson, G.A. The unit preference strategy in theorem proving. *AFIPS Conference Proceedings 26,* Spartan Books, Wash. D.C., 1964, pp. 615–621. (Reprinted in [44], vol. 1, 387–393).
53. Wos, L.T., Carson, D. and Robinson, G.A. Efficiency and completeness of the set of support strategy in theorem proving. *J. Assoc. Comput. Mach. 12,* 1965, pp. 536–541, (Reprinted in [44], volume 1, 484–489).

CR Categories and Subject Descriptors: D.1.6 [**Programming Techniques**]: Logic Programming; D.3.2 [**Programming Languages**]: Language Classifications—*PROLOG;* F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*complexity of proof procedures, pattern matching;* F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*computational logic, logic programming, mechanical theorem proving, proof theory;* I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*answer/reason extraction, deduction, logic programming, metatheory, resolution;* I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*predicate logic*

Additional Key Words and Phrases: Unification

Further Reading

Several excellent and recent further sources are: J.W. Lloyd's *Foundations of Logic Programming* (second, extended edition, Springer-Verlag 1987) and *Logic,*

Programming and Prolog by Ulf Nilsson and Jan Maluszynski (Wiley, 1990) provide rigorous but readable accounts not only of much of the material covered in the present article but also of many noteworthy later developments. Among these are:

- the addition of imperative control features such as the *cut*;
- the elegant *negation as failure* technique by which all modern Prolog systems permit *negative* conditions in both positive and negative conditional clauses;
- the inclusion of *arithmetical, list-processing, metalinguistic* and other applied predicates and operators among the atoms and terms;
- alternative logic programming paradigms, such as *concurrent logic programming, constraint logic programming,* and *higher-order logic programming.*

For the reader who wishes to learn more about *applications* and *methodology* of logic programming, about Prolog, and about exploiting the potential parallelism in logic, I also recommend the following recent books:

- *The Art of Prolog* by L. Sterling and E. Shapiro (MIT Press, 1986);
- *Prolog Programming for Artificial Intelligence* by I. Bratko (second edition, Addison-Wesley, 1990);
- *The Craft of Prolog* by R. O'Keefe (MIT Press, 1990).
- *Essentials of Logic Programming* by C.J. Hogger (Oxford: Clarendon Press, 1990).
- *Parallelism in Logic: its potential for performance and program development* by Franz Kurfess (Braunschweig, Vieweg, 1991).
- *Parallel Logic Programming* by Evan Tick (MIT Press, 1991).

About the Author:

J.A. ROBINSON teaches philosophy and computer science at the University of Syracuse, where he is now University Professor. His research interests include computational logic and automated deduction. He is currently working on a massively parallel logical computation system combining the lambda calculus (for functional programming) with the predicate calculus (for logic programming) at the University of Tokyo, where he is on a year's leave. **Author's Present Address:** Office of the University Professor, Syracuse University, Syracuse, NY 13244-2010