# The Role of Computational Logic as a Hinge Paradigm among Deduction, Problem Solving, Programming, and Parallelism

**M. Hermenegildo**

Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid - Spain
herme@fi.upm.es

**Abstract**

This paper presents some brief considerations on the role of Computational Logic in the construction of Artificial Intelligence systems and in programming in general. It does not address how the many problems in AI can be solved but, rather more modestly, tries to point out some advantages of Computational Logic as a tool for the AI scientist in his quest. It addresses the interaction between declarative and procedural views of programs (deduction and action), the impact of the intrinsic limitations of logic, the relationship with other apparently competing computational paradigms, and finally discusses implementation-related issues, such as the efficiency of current implementations and their capability for efficiently exploiting existing and future sequential and parallel hardware. The purpose of the discussion is in no way to present Computational Logic as the unique overall vehicle for the development of intelligent systems (in the firm belief that such a panacea is yet to be found) but rather to stress its strengths in providing reasonable solutions to several aspects of the task.

**Keyword Codes:** I.2.3, I.2.4, I.2.5
**Keywords:** Deduction and Theorem Proving, Knowledge Representation Formalisms and Methods, Programming Languages and Software.

# 1 The Role of Logic

Logic has been the foundation of computing since the seminal work of Church [7], Turing [55], and Gödel [16] which was materialized by von Neumann and by Turing himself in the first digital computers. Today it is used as an essential tool both in computer design and in computer programming. Regarding the latter task, logic can be viewed to have two primary roles. The first one, part of which is generally referred to as the "logic of programs", is to provide a tool in order to formally reason about properties of programs

such as, for example, their correctness with respect to some specification. This was already mentioned by Turing and von Neumann and then developed by Hoare, Dijkstra, Floyd and others [44, 15, 26]. The second role, part of which is generally referred to as "logic programming" (in the widest meaning of the term), is as the instrument of the task of programming itself: Turing already suggested this possibility when he stated, referring to the ideal means to communicate with digital computers, that "in principle, one should be able to communicate in any symbolic logic". This prediction has actually materialized in practice since two symbolic logic formalisms – Church's lambda calculus [8] and Robinson's Horn-clause resolution predicate calculus [47] – are the formal underpinning of two of the four main programming paradigms – functional programming (as exemplified by Lisp [38], ML, etc.) and relational programming (as exemplified by Prolog [48, 9]). The other two paradigms, imperative and object-oriented programming, will be returned to later.

Above, and following a suggestion made by Robinson, the term "relational programming" has been used instead of the usual "logic programming" since functional programming also has its roots in logic. The term "declarative" or "deductive" programming is frequently used as well to refer to programming which is based on logic formalisms. In any case the relation between functional and relational programming is very tight, and quite successful languages have been recently proposed which combine the characteristics of both (e.g. [41, 5, 34, 1]). The term "logic programming" will be used for simplicity in the rest of the paper to refer to logic-based programming formalisms, although our emphasis will be to some extent on systems based on relations.

The issue of programming paradigm is directly relevant to the designer of intelligent systems from two points of view: knowledge representation and procedural knowledge encoding. From the knowledge representation point of view it is well known that logic is a natural formal basis for the standard representation mechanisms (e.g. semantic nets or frames) usually used to perform this task. This is not to say that logic should be the only representation tool, of course, since other, higher-level representations may offer syntactical convenience, but rather to stress that these representations map naturally to logic and find in logic their natural formal underpinning. We will return to the issue of higher level abstractions later. From the procedural knowledge encoding point of view it is important to note that the development of intelligent systems generally results in large programs (in the sense of both consisting of a large program text and also needing large amounts of memory and many machine cycles to execute) which typically include a mix of symbolic and numerical processing. Examples are expert systems, intelligent design systems, and AI systems in general. Two characteristics common to most of these systems are their difficulty in programming due to the large complexity involved and the often limited performance of the resulting systems (at least, slower than one would ideally like). It is obviously desirable to remedy this situation. We will take a practical, "problem-oriented" view in order to deal with the issues involved in building of such systems: we will concentrate on identifying the fastest and easiest economical path that will take us from *problem description* to *high-performance solution.* Thus we will be concerned simultaneously with the issues of ease and speed of programming, execution performance, and overall cost.

In fact, in "logic programming" the two roles of logic (as an instrument of both reasoning about programs and programming) merge since the programs themselves are their specifications, and their correctness is given by the correctness of the underlying proof procedure used to "execute" them (resolution, for example). This corresponds to the "declarative" view of of logic programming. Logic programming also has a "procedural" view to which we will return to later. It is the fact that one can freely choose between the declarative or procedural interpretations that gives logic programming its hinge or

mediator role between traditional and declarative programming, or between the "formal" or "ad-hoc" views of AI.

## 2 The Declarative View

The case for the use of logic programming in a declarative way can be supported for example as follows. At our (relatively modest) current state of development in human interaction with computers, our main means for instructing them what to do is by writing a program, i.e. a list of instructions which are to be executed by it.[1] These instructions are expressed in a particular language that the computer can understand: a *programming language*. Conventional programming languages generally express these instructions as a series of precise "actions" that are to be performed by the computer one after the other. This is known as an imperative style, a program being a sequence of commands. Programs today come in this format largely as a consequence of the fact that the first computers were no more sophisticated than one of today's hand-held calculators and that early programs were just sequences of the basic instructions that a particular machine could directly execute. Programming languages then emerged as a tool for making it easier for a human being to express the actions required from the computer. However, it is a fact that computers already existed in a particular form before these languages were designed, and this undoubtedly invited a "machine-oriented" style in these designs which still lingers in today's programming languages [3]. These languages are often so apart from the natural way in which humans think and express themselves that programming a computer is frequently a difficult and error-prone task for any sizeable problem, let alone trying to reproduce intelligence.

The question of course is whether we can design a means of communicating with the computer which is free from the imperative style. If we avoid any machine-oriented considerations, the first language to come to mind is, of course, the human natural language: the user's mother tongue. However, such a language presents a number of serious drawbacks. These drawbacks include its verboseness, which is only made worse by its vagueness and ambiguity if not provided with a suitable context or a great deal of (normally assumed) knowledge. This fact was already realized by mathematicians in ancient times (and therefore long before computers came to being) and "logic" was devised as a means of clarifying and/or formalizing the human thought process. Logic lets us express facts and rules about the world in a precise and concise way and draw conclusions from them which can be formally proven to be correct. Symbolic logic is simply a shorthand for expressing conventional logic.

It is the ability of symbolic logic to express knowledge in a way that is very precise and compact, while at the same time close to the natural way in which humans express themselves, that led to the concept of using logic as a means for communicating with computers. But simply describing a problem is not sufficient by itself for actually solving it. Being able to solve problems specified in logic is akin to being able to automatically prove theorems in that logic, as pointed out by Green [17] in his early work on question-answering systems. An effective proof procedure is a necessary addition. Such a proof

[1] The neural computing approach, where rather than explaining the task and/or its solution to the computer the computer itself learns to solve the problem through training, is an alternative [49]. This is an area in which interest has recently grown again, and several successes have been reported in a number of application areas, most of them related to pattern recognition, where other approaches have obtained weak results. However, the neural approach, although promising, does not seem yet to be capable of solving the large range of problems that general-purpose computers need to tackle, so we will limit the discussion for the moment to "programmed" approaches.

procedure was first found by Robinson with his discovery of the resolution principle [47]. With all the necessary components in place the idea of "programming in logic" was first proposed in a formal manner by Kowalski not many years ago [32, 33]. The resolution principle gives such logic programs a computational capacity which is Turing complete, i.e. it can express and implement all computable recursive functions. Its practical realization in the form of the Prolog language, proposed by the group led by Colmerauer [9] and efficiently implemented by Warren [59, 60], has granted logic programming acceptance as one of the well established programming paradigms. Logic provides more concreteness than natural language, but it is also far less machine oriented than conventional computer languages. The main difference with them is its declarative nature: in logic, statements express facts, knowledge about the problem to be solved, rather than precise instructions to be followed step by step. It implements to a certain degree a class "automatic programming" and "automatic problem solving" – in fact, the widest class possible with formal means.

It also turns out that the logic programming model is specially suitable for the storing and retrieval of information in large knowledge bases. In fact, the model is quite suitable for the general class of database applications and a good part of modern database theory (and some practical systems) is largely based on the "deductive database" approach, i.e. on logic programming [57]. In this application the full power of first order logic is sometimes not required and proof procedures can be used which are completely decidable, thus implementing to a greater extent the "automatic problem solving" or "automatic programming" concept in this reduced context. One problem often quoted in relation with knowledge bases implemented using the deductive approach is the difficulty in modeling changes in the database – the view update problem– since traditional logic has a static view of the set of axioms. Much work has been done by the database and logic programming communities in this area and several solutions have been found which provide a logical view of these updates, generally by having a meta-notion of "contexts" or theories which can be updated and within which proofs can be carried out in the normal way [57].[2]

# 3 Merging with the Procedural View

Despite the advantages of the declarative approach, one can also argue sensibly against it. In particular it is true that while simply specifying a problem in logic and letting an automatic decision procedure solve it is probably the easiest way of getting to that solution, it may on the other hand be a very inefficient way of achieving such a goal. One can argue that, even if more laborious, it is better in some cases to work out an efficient algorithm. But it is important to realize that logic programming *does not prevent this.* This controversy between procedural and declarative programming (which is in essence the same controversy found between the Minsky and McCarthy –"ad-hoc" and "formal"– points of view, which appeared in the late sixties and is still mentioned sometimes today [40]) was settled by Kowalski in an elegant way by *viewing the "workings of resolution" on a logic program as its step-by-step procedural interpretation.* Kowalski thus found an effective and practically most useful procedural interpretation of logic programs. Practical logic programming systems exploit this procedural interpretation to offer not just a declarative language but also one that allows the practical and elegant description and efficient execution of algorithms. One can look at the logic statements as simple logic statements

[2] "Linear" logic, which allows the addition and "consumption" of axioms is another approach which is currently generating considerable interest.

or one can –by thinking of the steps that resolution would do for this program– think of them as step-by-step commands given to the computer. Thus, logic programs can also be used to perform the exact same tasks performed by imperative languages when needed, in that case still offering advantages of compactness, rich data structures, automatic memory management, efficient implementation of recursion, etc. and with comparable efficiency. Thus, within the same framework one can specify a problem and have it be solved automatically (within the limits of first order logic and perhaps somewhat inefficiently), or, alternatively, "code" an algorithm for solving it more efficiently. One can leave parts that are not performance sensitive coded very close to specifications and, staying within the same paradigm, speed up other parts by building in "control". Thus, these languages can achieve when needed the algorithmic efficiency of the imperative approach.

A related issue is that of the impossibility of building a system capable of solving automatically all problems which stems from Gödels incompleteness results. It is true that even first-order logic is only semi-decidable and that there are limits to what can be done with the formal approach. But it is also true that many useful and difficult problems fall within these limits and can be tackled. Logic programming provides an ideal framework to do so because, as pointed above, it allows us to "go as far as the formal approach will go". But it is important to point out that its procedural interpretation allows the straightforward "coding" of any other solutions using heuristic or other approaches in an elegant way. Again, the procedural/declarative dichotomy allows the merging of the formal and ad-hoc approaches.

Another argument often used against logic programming is that it is based on first order predicate logic, and that other types of higher-order and modal logics are more suitable for solving AI problems because of their expressive power. The point on the expressive power is valid in principle. On the other hand the difficulties can be overcome even in the framework of first order logic by formalizing things in different ways. It is also important to point out the results due to Lindström [36] that first-order logic occupies a unique place among logical systems because there is no logical system with more expressive power for which we can have an adequate formal concept of proof and many other useful properties that first order-logic does have. Thus it seems a good idea to provide first-order logic and an effective proof procedure (resolution) as the "built-in" paradigm and provide hooks and mechanisms for higher-order and modal logic to be dealt with through "programming" in the particular way the user needs to, which is the approach taken in current logic programming systems.

# 4 Constraint Logic Programming

Another of the beauties of the relational nature of most logic programming systems is that they can be used without change for tasks which are different from the one that the original programmer might have had in mind. In that way, a sorting program can be used to generate permutations (and the other way around), a program designed with symbolic derivation in mind can do symbolic integration, and a program for computing squares can solve square roots. It has been argued that the class of programs for which this is possible is reduced in practical systems, in particular in Prolog systems. This limitation appears generally in programs which use Prolog's arithmetic, which is non-logical and, therefore, not "reversible" (it is also related to Prolog's search rule).

The sound implementation of most mathematical operations has been made possible by the extension of logic programming to Constraint Logic Programming (CLP) by Jaffar, Lassez, and Colmerauer [28, 29, 10]. This framework, which is still in a form

first order predicate logic, not only overcomes most of the limitations in the reversibility of traditional logic programming, but also greatly augments its practical expressive power. CLP programs can perform computations over both symbolic and non-symbolic domains. In such programs unification is replaced with tests for *constraint satisfaction.* Expressive power is thus greatly enhanced since the domains used can be richer than the usual Herbrand domain and unification is only a special case of the aforementioned tests for constraint satisfaction. Traditional logic programming is viewed in this context as a special case where the domain is that of terms, the only constraint equality, and the constraint solving algorithm unification. In general, practical CLP systems support several domains and types of constraint operations over such domains as linear equations and inequations over reals or rationals, interval arithmetic, constraints over finite sets, boolean constraints, etc.[30, 37, 10, 45]. The solving algorithms which are used in this context beside unification can be incremental gaussian elimination, incremental simplex, forward checking, several types of finite domain relaxation, lookahead, etc. CLP systems have the capability of solving a much larger set of problems in a direct way than traditional systems. On the other hand the constraint satisfaction tests can be much more expensive than unification and may result in low run-time performance. Thus, there is a considerable amount of interest in the efficient sequential and parallel implementation of CLP systems. This is similar to the efforts made in logic programming which eventually produced efficient sequential and parallel Prolog implementations.

# 5 Reactiveness, Concurrency, Modeling

In the above, the deductive approach has been mostly compared with the imperative approach. Object oriented programming, although in a way part of the imperative approach, has some special characteristics which deserve special mention. In a comparison between the "object-oriented" and "logic programming" approaches one can present some arguments in favour of the former which are relevant to our discussion. One could of course also conversely quite easily argue that a good number of the very desirable characteristics of declarative programming, presented in the previous sections, are missing from object-oriented programming, but that is not the issue here.

A first argument which could be made for object-oriented programming is that it is reactive in nature and provides message communication. Actually, logic programs can also be reactive (and concurrent) and incorporate sophisticated forms of message communication, as exemplified by concurrent logic programming systems such as Concurrent-Prolog [52], PARLOG [18] and GHC [56]. It is true, however, that these systems have until now been to some extent unsatisfactory because they did not implement any of the search capabilities usually associated with logic programming systems. Thus, they have been largely unsuited for the "problem solving" applications for which logic programming is well known. However, recent models based on the "Andorra principle" proposed by Warren [50, 21] successfully combine reactiveness with traditional logic programming.

Another argument for object-oriented programming is that it could be seen as more flexible and better capable of modeling application domains because it incorporates data abstraction and information hiding, inheritance, and object identity and persistence. It has been shown that one way of supporting the functions of data abstraction, information hiding, and inheritance is by considering the objects to be the terms of the language and by enriching the structure and properties of such terms, as well as the unification algorithm, in a suitable way [1]. This, which was in principle a departure from standard logic programming, fits however now quite naturally in the context of constraint logic

programs. The issue of object identity is also covered in this approach. Persistence can be considered a matter of point of view (similar to assignment): an object which changes can be implemented logically by a new object which incorporates the change. There is an obvious efficiency issue if this is implemented directly this way but it not need to, and it is in the end an implementation / garbage collection issue, just as updates to a list only make copies of the list when strictly necessary in current implementations of logic programs.

In any case, and when departing from more theoretical considerations, combining object orientation and logic languages *in practice* is not a problem and a good number of object-oriented versions of Prolog have been developed, as, for example, [27] which provide the advantages of both of these quite interesting and promising programming paradigms. Finding elegant forms of merging logic- and object-oriented programming is, however, still quite an interesting research topic (see, for example [39] and its references).

# 6 Parallelism and Efficiency Issues

Having dealt with high-level issues such as programming paradigm and expressive power, lower level issues such as efficiency and practicality are now addressed. Some arguments supporting the suitability of declarative languages (which seem to be the languages of choice for the implementation of knowledge-based and other highly complex systems) have been given above. In fact, most of the landmark implementations in the field of AI have used such declarative languages. On the other hand, lower-level languages are very often used in more "practical" applications and some commercial products. Two arguments are often quoted as justifying such a move. First, the higher execution speed of low-level, imperative languages. Second, their availability on standard platforms, in particular on UNIX-based systems and their seamless interface with that environment.

Starting with the last point above, there is a certain belief that logic and functional languages need "strange" and expensive computers and environments. The appearance of Lisp- and Prolog-machines a few years ago gave rise to this impression. In fact, this belief is nowadays quite unfounded: modern versions of well established languages such as Prolog and Lisp run well on standard platforms, often having excellent public domain implementations – SICStus Prolog [6] and Kyoto Common Lisp are good examples of widely available, excellent quality implementations which run on standard platforms (e.g. UNIX) and interface seamlessly with the operating system environment.

Regarding the first point above, there is also a belief that imperative languages such as C are faster than Prolog or Lisp. This has been, on the other hand, true in some cases and has motivated the use of such imperative languages in some commercial products. Speed, thus, would appear to make a good case for such languages. However, it cannot be forgotten that these languages suffer from intrinsic drawbacks when used for programming knowledge-based systems: their lower level makes programming more tedious, having to deal with very low-level issues (such as dynamic memory management) which are otherwise automatically taken care of in logic and functional languages. This need of attention to low-level details can be a serious handicap when developing large systems. Also, the imperative nature of such languages makes programming more prone to error and debugging more difficult. And their lack of a mathematical foundation makes their formal treatment (and thus the development of advanced program analysis tools) a hard problem. The real argument left in their favor seems to be, then, execution speed.

Our thesis is that this argument may very soon be unfounded. First, sequential implementations of logic and functional languages are getting highly competitive through the

use of advanced implementation and compilation technology based on efficient abstract machines [60] and program analysis techniques such as abstract interpretation [12, 13]. Current Prolog implementation technology offers performance which slowly approaches that of, for example, C [58]. And, while still slower than C, if the speed of current sequential systems is preserved in each processor, an order of magnitude speedup in logic programming obtained from parallelism would already represent a faster execution than C.

Thus, parallelism greatly affects the balance. In fact, efficient, high-performance multiprocessors are now a practical reality and in the very short term desktop workstations with several processors will be the norm. Unfortunately, making effective use of such computing power from lower-level imperative languages is a tedious and error-prone manual task. Furthermore, any automation of this task is very much complicated by the very nature of such languages. However, if more declarative languages are used, then most of the additional work involved in programming for multiprocessors can be simplified or eliminated altogether by using advanced program analysis tools. In addition, the declarativeness and mathematical foundation of such languages makes it feasible to prove the correctness of the parallelization techniques.

One question that may immediately come to mind is why one should consider parallelism and multiprocessors at all. In fact, such machines offer clear advantages at (at least) two levels of cost/performance tradeoff. At the high end, i.e. when trying to achieve the ultimate performance, parallelism is a must because technology has been pushed to the limit and we simply don't know how to make a faster machine. At this point the only way to achieve better performance is by adding more CPUs as exemplified by the Cray-XMP. At a lower and more practical end there is another point where parallelism can offer extremely good performance in a cost-effective way: at any point in time there is a "fastest, reasonably priced" machine, which is often exemplified by the high-end workstation. Elements being used in such a workstation are the fastest current microprocessors (probably with RISC architecture) and a large cache and memory. Having used most of the "tricks" in the bag of the architecture designer a further increase in speed can be obtained by moving to a different technology (at a very large cost), or, alternatively, by putting together several of these microprocessors to build a multiprocessor, the latter being often a much more cost-effective alternative. A similar argument can be made at the "mini" level. In general, in this view parallelism makes it possible to increase performance linearly with cost while avoiding a highly expensive jump to a different technology.

The question, then, might remain on whether multiprocessors are practical today. In fact, efficient, practical, high-performance UNIX multiprocessors are now a market reality as exemplified by machines such as the Sequent Balance/Symmetry [51], Encore, BBN Butterfly, and, most importantly, by the current generation of desktop multiprocessor workstations pioneered by the Sun Galaxy, which offer extremely good cost/performance ratios. DOS-based multiprocessors are also starting to appear in the market and to be widely used as compute and file servers. Most of these machines use the latest generation of microprocessors.

The question then may not be so much whether practical multiprocessors are available but how one can tap a performance potential that is already available on one's desktop. However, the answer to this question is the real key to multiprocessing: using the power of multiprocessors is still difficult. The amount of software that can exploit the performance potential of these machines is still very small. This is largely due to the difficulty in mapping the inherent parallelism in problems on to different multiprocessor organizations. There are in general at least two ways in which such a mapping can be performed: it can be done explicitly in the program by the user by means of a language that includes

parallel constructs, or it can be automatically uncovered by a "parallelizing" compiler from a program which has no explicit parallelism in it. Both approaches have their merits and drawbacks. A parallelizing compiler makes it possible to tap the performance potential of a multiprocessor without burdening the programmer. However, the capabilities of current parallelizing compilers are relatively limited, specially in the context of conventional programming languages. Parallelism explicitly expressed by the programmer using specialized constructs can be used when the programmer has a clear understanding of how the parallelism in the problem can be exploited in the target machine. However, this adds in general an additional dimension of complication to the already complicated and bug-prone task of programming.[3] In reality, although experienced users may often have a correct intuition on which of the parts of a problem (and which of the parts of the associated program) can be solved in parallel, the task of correctly determining the *dependencies* among those parts and the sequencing and synchronization needed to reflect such dependencies is proving to be very difficult and error-prone. This was recently also pointed out by Karp [31] who states that "the problem with manual parallelization is that much of the work needed is too hard for people to do. For instance, only compilers can be trusted to do the dependency analysis needed to parallelize programs on shared-memory systems."

Therefore, the best programming environment would appear to be one in which the programmer can freely choose between only concentrating on the conventional programming task itself (letting a parallelizing compiler uncover the parallelism in the resulting program) or, alternatively, performing, in addition, the task of explicitly annotating parts of the program for parallel execution. In the latter case, the compiler should be able to aid the user in the dependency analysis and related tasks. Ideally, different choices should be allowed for different parts of the program.

However, automatic parallelization has up to now been an elusive goal. It can be done in some trivial cases: for example, operating system processes can be executed in parallel. An interesting case is parallel compilation, which can also often be done in parallel in UNIX systems. In this case the hard-to-find dependency information is routinely provided by the user in the "makefile." However, efforts in parallelizing user programs in imperative languages, mainly represented by parallelizing compilers for FORTRAN, have met with only limited success. Karp also points out the lack of good parallelizing compilers, and predicts that the technology is still several years away. One reason for this is that the programming languages that are conventionally parallelized have a complex imperative semantics which makes compiler analysis difficult and forces users to employ control mechanisms that hide the parallelism in the problem. It is very difficult to develop parallelization algorithms for such languages that are both effective and amenable to proof.

On the other hand, declarative languages require far less explication of control (thus preserving much more of the parallelism in the problem). In addition, their semantics makes them comparatively more amenable to compile-time analysis and program parallelization. In other words, such programs preserve more the intrinsic parallelism in the problem, make it easier to extract in an automatic fashion, and allow the techniques being

[3]In fact, the progress from systems which require from the programmer explicit creation and mapping of processes to a particular processor interconnection topology and extensive granularity control, to systems which don't require at least some of these tasks appears to be a leap forward comparable to the appearance of the concept of virtual memory (compared to overlays) or even high-level languages (compared to programming in machine code). Of course, in the same way as there is sometimes a case for assembler programming in particularly performance sensitive parts of a program, there will be some cases in which complete explicit control of parallelism is indicated.

used to be proved correct. It is our thesis that through advanced compiler techniques, such as abstract interpretation, automation of parallelization is indeed feasible for languages that have a declarative foundation.

The two main types of parallelism which can be exploited in logic programs are well known[11]: (1) and-parallelism and (2) or-parallelism. Several models have been proposed to take advantage of such opportunities (see, for example, [14], [46], [4], [25], [35], [63], [20], [62], [54], [2] and their references). Significant research effort has been and is being applied to developing or-parallel execution models (see, for example, [61] and its references). The associated performance studies have shown good performance for non-deterministic programs in a number of practical implementations, as exemplified by the Aurora [54] and MUSE [2] systems. The resulting speedups obtained over state-of-the-art sequential systems for programs which have this type of parallelism support the thesis defended in this paper.

Exploiting (independent) and-parallelism [23, 24, 14, 25] is more complicated because several kinds of dependency analysis must be performed on the programs. Goal independence is a function of the run-time instantiations of the variables in the goals being considered, and, therefore, is in general query-dependent. Efficient annotation requires either a priori knowledge of the binding patterns of the variables in the programs at run-time, or introducing in the program checks which can dynamically determine at run-time which goals should be executed in parallel. The latter can be done quite simply with only minor modifications to Prolog (as exemplified by the &-Prolog language [22]). The former has been one of the central research issues in the implementation of and-parallelism. Promising results have recently been obtained on how correct and efficient annotations can be generated automatically [43], how the peculiar characteristics of practical Prolog programs (for example, those with side-effects) are dealt with [42] and how much parallelism can be obtained from such automatic annotations. The latter point, which is the most relevant to our thesis is discussed in [22], by giving results for the &-Prolog system. The main conclusions are that for programs which contain this type of parallelism quite significant speedups can be obtained over state-of-the-art sequential systems.

Together, or-parallelism and and-parallelism appear to be capable of producing speedups of more than an order of magnitude over sequential systems in a large class of programs [53, 19]. This can put the performance of parallel logic programming systems beyond the performance of C on a commercial 10-processor machine and makes declarative programming an attractive implementation vehicle even when all arguments (including raw performance) are taken into account.

# 7 Conclusions

This paper has presented some brief considerations on the role of Computational Logic in the construction of Artificial Intelligence systems and in programming in general. It has succinctly addressed the interaction between the declarative (deduction) and procedural views of programs, the extent of the impact of the intrinsic limitations of logic, the relationship with other apparently competing computational paradigms, and finally it has discussed implementation-related issues, such as the efficiency of current implementations and their capability of efficiently exploiting existing and future sequential and parallel hardware. We have presented ways in which possible shortcomings of Computational Logic which may appear at first sight have been overcome by older and more recent results – from Kowalski's elegant merge of the procedural and declarative views to recent extensions, such as constraint logic programming, non-deterministic concurrent logic

programming, etc. These extensions greatly enhance declarativeness and the range of application domains and incorporate important features of the object oriented paradigm. We have also pointed out how many efficiency issues are already solved to a great extent, specially in view of the growing ubiquity of parallel computers. Sequential implementations approach the speed of lower level languages like C, while parallel implementations can surpass it. We have of course also pointed out that work does remain to be done in many areas from the improvement of expressive power (for example, a better merging of object orientation features) to further improvements in implementation technology. As mentioned before, the ultimate purpose of the discussion has been in no way to present Computational Logic as the unique overall vehicle for the development of intelligent systems (in the firm belief that such a panacea is yet to be found) but rather to stress its strengths in providing reasonable solutions to several aspects of the task, to encourage intelligent system designers to consider the use of Computational Logic systems, and to encourage further work in this uniquely promising area. Intelligent systems designers certainly need to look for rich high-level abstractions and knowledge-oriented description paradigms that allow them to easily model complex systems. At the same time logic offers an ideal formal underpinning for such systems, and a sound and efficient high-level implementation vehicle.

# References


[1] Hassan Ait-Kaci, Roger Nasr, and Pat Lincoln. E An Overview. Technical Report AI-420-86-P, Microelectronics and Computer Technology Corporation, 9430 Research Boulevard, Austin, TX 78759, December 1986.

[2] K.A.M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming.* MIT Press, October 1990.

[3] J. Backus. Can programming be liberated from the von neumann style? *Communications of the ACM*, 21(8):613–641, 1978.

[4] P. Biswas, S. Su, and D. Yun. A Scalable Abstract Machine Model to Support Limited-OR/Restricted AND Parallelism in Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 1160–1179, Seattle,Washington, 1988.

[5] P. G. Bosco, C. Cecchi, C. Moiso, M. Porta, and G. Sofi. Logic and functional programmin on distributed memory architectures. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 325–339, Jerusalem, 1990. The MIT Press.

[6] M. Carlsson. *Sicstus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[7] A. Church. A set of postulates for the foundation of logic. In *Annals of Mathematics Studies*, volume 2. 1933.

[8] A. Church. The calculi of lambda conversion. In *Annals of Mathematics Studies*, volume 6. Princeton University Press, 1941.

[9] A. Colmerauer. Les gramaire de metamorphose. Technical report, Univ. D'aix-Marseille, Groupe De Ia, 1975.

[10] A. Colmerauer. Opening the Prolog-III Universe. In *BYTE Magazine*, August 1987.

[11] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[12] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Rec. 4th Acm Symp. on Prin. of Programming Languages*, pages 238–252, 1977.

[13] S. Debray, editor. *Journal of Logic Programming, Special Issue: Abstract Interpretation*, volume 13(1–2). North-Holland, July 1992.

[14] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[15] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall series in automatic computation. Prentice-Hall, Inc., Englewood Cliffs, N. J., 1976.

[16] K. Goedel. Uber Formal Unentschiedbare Sätze de Principia Mathematica und Vervandter System i. *Mat. Phys.*, 38:173–198, 1931.

[17] C. Green. Theorem proving by resolution as a basis for question-answering systems. *Machine Intelligence*, 4, 1969.

[18] S. Gregory. *Parallel Logic Programming in PARLOG: the Language and its Implementation.* Addison-Wesley Ltd., Wokingham, England, 1987.

[19] G. Gupta and M. Hermenegildo. Recomputation based Implementation of And-Or Parallel Prolog. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems.* Institute for New Generation Computer Technology (ICOT), June 1992.

[20] G. Gupta and B. Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 332–349. MIT Press, October 1989.

[21] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: A Model Integrating Dependent-, Independent-, and Or-parallelism. In *1991 International Logic Programming Symposium.* MIT Press, October 1991.

[22] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[23] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[24] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[25] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[26] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the CM*, 12(10), 1969.

[27] ICOT. ESP Guide. Technical Memorandum TM-388, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, JAPAN, 1987.

[28] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. Technical report, Dept. of Computer Science Monash University, June 1986.

[29] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Fourteenth Ann. ACM Symp. Principles of Programming Languages*, pages 111–119. ACM Press, 1987.

[30] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Fourth International Conference on Logic Programming*, pages 196–219. University of Melbourne, MIT Press, 1987.

[31] A.H. Karp and R.C. Babb. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*, September 1988.

[32] R. A. Kowalski. Predicate Logic as a Programming Language. In *Proceedings IFIPS*, pages 569–574, 1974.

[33] Robert A. Kowalski. *Logic for Problem Solving.* Elsevier North-Holland Inc., 1979.

[34] P. Lescanne and W. Wechler, editors. *Journal of Logic Programming, Special Issue: Algebraic and Logic Programming*, volume 12(3). North-Holland, February 1992.

[35] Y.-J. Lin. *A Parallel Implementation of Logic Programs.* PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, August 1988.

[36] P. Lindstroem. On extensions of elementary logic. *Theoria*, 35:1–11, 1969.

[37] H. Simonis M. Dincbas and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1 & 2):72–93, 1990.

[38] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, MA, 1965.

[39] J. Meseguer. Multiparadigm logic programming. In H. Kirchner and G. Levi (eds.), editors, *Proceedings of the 3rd. Int'l. Conf. on Algebraic and Logic Programming.* Springer LNCS, 1992.

[40] M. Minsky. Logical vs. analogical or symbolic vs. scruffy. *AI Magazine*, Summer Issue, 1991.

[41] Juan José Moreno-Navarro and Mario Rodriguez-Artalejo. Babel: a Functional and Logic Programming Language Based on Constructor Discipline and Narrowing. In Grabowski *et al.*, editor, *Lecture Notes in Computer Science. #343. Algebraic and Logic Programming.* Springer-Verlag, 1988.

[42] K. Muthukumar and M. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In *1989 International Conference on Logic Programming.* MIT Press, June 1989.

[43] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[44] C. Hoare O. Dahl, E. Dijkstra. *Structured Programming.* APIC Studies in Data Processing. Academic Press, 1976.

[45] W. Older and A. Vellino. Extending Prolog with Constraint Arithmetic in Real Intervals. In *Canadian Conference on Electrical and Computer Engineering*, September 1990.

[46] B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 313–331. MIT Press, October 1989.

[47] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(23):23–41, January 1965.

[48] P. Roussel. Prolog: Manuel de reference et d'utilisation. Technical report, Univ. d'Aix-Marseille, Groupe de IA, 1975.

[49] D. Rumelhart and J. McLelland. *Parallel Distributed Processing.* MIT Press, 1986.

[50] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 825–839. MIT Press, June 1991.

[51] Sequent Computer Systems, Inc. *Balance 8000/21000 Technical Summary*, 1986.

[52] E. Y. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. Technical Report TR-003, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, January 1983.

[53] K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *1991 International Logic Programming Symposium.* MIT Press, October 1991.

[54] P. Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.

[55] A. Turing. On computable numbers with an application to the entschiedungs problem. *Proc. London Mathematical Society*, 2(42):230–265, 1936.

[56] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.

[57] J. D. Ullman. *Database and Knowledge-Base Systems, Vol. 2.* Computer Science Press, Maryland, 1990.

[58] P. Van Roy and A. M. Despain. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. In *Proceedings of the North American Conference on Logic Programming*, pages 501–515. MIT Press, October 1990.

[59] D. H. D. Warren. *Applied Logic—Its Use and Implementation as Programming Tool.* PhD thesis, University of Edinburgh, 1977. Also available as SRI Technical Note 290.

[60] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.

[61] D. H. D. Warren. OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science. Springer-Verlag, March 1987.

[62] D. H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *International Symposium on Logic Programming*, pages 92–102. San Francisco, IEEE Computer Society, August 1987.

[63] H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR-Parallelism. In *Symp. of Logic Prog.*, pages 436–448, August 1987.