Aspects of PROLOG History: Logic Programming and Professional Dynamics

Philippe Rouchy, Blekinge Institute of Technology, Sweden.
Philippe.Rouchy@bth.se

PLAN:

1. INTRODUCTION

This paper presents aspects of PROLOG history in a three part argument: (1) the birth of an algorithm and the historical background of artificial intelligence and PROLOG and its relation with logical programming, (2) different research done on PROLOG in order to expand, adapt or transform its features for implementing different data structures and system architecture. Notably, I present the development of logical and parallel architecture (Kowalski, Clark and Gregory, 1982), hybrid systems such as POPLOG using elements of PROLOG with LISP and the use of other languages and non-logical augmentation (Mellish and Hardy, 1982) and some reflection on computer and education dealing with the definition of the user in relation to the development of micro-PROLOG (Ennals, 1982, 1984) and (3) a reflection on the infra-transformation of the computing field in terms of professional dynamics and its implication for a social studies of technology grasp of computer sciences along the line of its epistemic practices.

Since 1970, PROLOG has been through several evolutions and implementations reflecting the modern history of formal mathematical thinking in computing. It has been made possible largely by disseminating a working version of PROLOG, known as the Edinburgh version, which became ISO standard in 2003. In this paper, I suggest it is not enough to appreciate PROLOG's role in computing communities from its practical results (such as relational databases, problem solving, design automation, symbolic equation solving, biochemical structure analysis etc.). A more fruitful road demands to understand the process linking the operational framework of implementation with PROLOG logical structure.

My paper seeks to convey a sense of research dynamics that went into PROLOG implementations[1] through a selected review of historical cases. It approaches research as a process not a product. The arrangement between real condition of implementation and PROLOG internal structure is a subtle domain. It delimits computer scientists' area of implementations and skills developments distinguishing and animating a specific professional community. This is a strong suggestion for the sociologist of science to understand not only the basic features of PROLOG in order to perceive the shapes of implementations but also to convey the way in which technical objectives and hardware prospects of exploitation determine PROLOG implementations. If this research objective is attained, one should have a sense of computer scientists' epistemic framework of activities. The depth of this concern is revealed by the way one understands how PROLOG formal computing logic takes its shape from the ordinary features of computing systems' exploitations.

2. HISTORICAL BACKGROUND

2.1 PROLOG in the history of AI

[1] Ehud Shapiro (1989) and Gupta et al (2001) deserves special mention by providing encyclopaedic surveys of families of PROLOG, respectively concurrent logic programming and parallel PROLOG programs.

In 1972, Alain Colmerauer and his colleagues Bob Pasero and Philippe Roussel (Roussel, 1975) invented the programming language PROLOG at the University of Marseilles, France. PROLOG demands the programmer to specify task in logic rather than conventional instructions. Hence, its name PROLOG stands for Programming in Logic (*Programmation en Logique* in French). The software implements man-machine communication in natural language with a question – answering system. It has a similar objective as Terry Winograd's use of micro-PLANNER (1972), but the novelty comes from the syntactic-arithmetic liaison for programming rather than semantic.

In the 1960's Artificial Intelligence research context, early computer scientists were investigating the possibilities for computers to prove theorems automatically (Bowen, 1979). Several ambitious programs of research sought to explore the possibilities of human reasoning in problem solving, means-ends analysis, and rational choices theory working out programming structures in LISP (Newell, 1958). John McCarthy, LISP creator, devised to program by setting instructions in declarative form. LISP was the first program constituted of instructions which involved sub-instructions and, this, virtually *ad finitum*. Hewitt's (1970) program PLANNER offers an alternative for instructing the computer based on a procedural plan which demands programming by pattern of goal and assertion. Sussman, Charniak (1970) Micro-PLANNER offered the possibilities to deal with a more complex structure of information by working with assertions and goals (Winograd, 1972). From Hewitt's view point, one can see PROLOG has reinvented a subset of Micro Planner, i.e. its assertion component. In fact, the story of PROLOG is not so linear, as it does not follow from the early development of mainly US development in Artificial Intelligence. One of the issues with the complexity of assertion in programming was the issue of control structure. For example, Colmerauer noticed (Kowalski, 1988: 41) that it is impossible to operate the list 'append' in PLANNER without resorting to LISP and this may bring an issue of coherence since one has to deal with large lists, or complex lists of information.

The original idea of PROLOG was to program not with a set of instructions but directly with formal arithmetic logic as a total solution[2]. This idea has its own history. In 1970, Colmerauer et al. (1993) came in touch with research done (Hutchins, 2001) at Montreal. Canadian computer scientists worked on a syntactic transfer system for English-French translation. From the TAUM project (Traduction Automatique de l'Université de Montréal) comes two major achievements: (1) the Q-system formalism for manipulating linguistic strings and trees and (2) the Météo system for translating weather forecasts. This preliminary research on natural language communication was combined with Kowalski's research on automated theorem-proving (Kowalski, 1971). Colmerauer, invited Kowalski to discuss the logic of their question-answer system. This followed up on Philippe Roussel and Jean Trudel's (University of Montreal) works on the deductive aspects of language parsing and its relation to the SL-resolution theorem prover. Kowalski worked out with Colmerauer (Colmerauer, 1982) a way to represent grammar by using logic and to parse sentences by using the principle of resolution (see next section). Colmerauer and his team worked with resolution, as a uniform general-purpose theorem.

In 1971, Colmerauer and his colleagues' aimed at developing a deduction machine based on text written in French. The first step toward the logical system in PROLOG as we know it is the research on a preliminary system of natural language communication (Colmerauer, 1971). It consists in connecting logical formulae and French sentences with 50 rules written in Q system (a paraphrase generator). Colmerauer uses his work on Q-grammar (Colmerauer, 1970) to device another logical method of representing language. He sought to avoid associative logic (like the LISP function, see Boyer & Moore, 1975) by a formal representation of strings of language coming from his work on Q-grammar. In 1972, the second step securing the arithmetical aspect of logic took place. Thanks to work done from Kowalski's SL-resolution prover (linear resolution by selection function), it helped finalise the connection between natural language processing and automated problem-solving.

The creation of a language working with arithmetic logic demands notational adaptation. Robinson's (1969) notation became the standard for programming in PROLOG. For example, a function in lambda-calculus notation such as:

$\sqrt{[x(x+1)]}$

Can be expressed as an application notation such as:

Γx(SQRT((TIMEx)((PLUSx)ONE)))

To finish on the relationship between PROLOG and Artificial intelligence, I would like the reader to sensitise himself to the context of development: (1) the 1970s artificial intelligence development in

[2] The formulation of this idea is to be found in Robert Kowalski (1982) 'Logic as a computer language' in K.L. Clark and S-A Tärnlund (eds.) Logic Programming, Academic Press, London: 3-18.

regard of the hardware and software of the time. (2) Its implication for the unsettled distinction between heuristics and algorithm in early artificial intelligence and the place of PROLOG vis-à-vis it.
(1) It is essential to realise that limits in programming reflected limits of the processor power and memory. In 1971, PROLOG early implementations (Colmerauer, Roussel, 1992: 2) were done, in France, with an IBM 360-44 in Marseilles University with 900 K of memory and an operating system without virtual memory. This machine put 1 Mb of memory available to execute the programs and an operator console as interface between user and programs.
(2) The early work on machine and program demanded from the computer scientists to find a way for the machine to execute instructions. Early computer scientists were working in applied mathematics departments whereby they started by matching their knowledge of mathematical procedures to a language that the machine could process. In this context, the issue of the difference between heuristics and algorithm (Campbell, 1985) became obvious. To clarify the contrast, I would say that different philosophies of the implication of mathematics for computing came into play. I would suggest that Artificial Intelligence researchers such as Allen Newell assumed a cognitive and behaviourist theory of computing. His research in heuristics starts with the idea that arithmetic reasoning and thinking process are essentially identical and therefore their matching is the way to go in the exploration and development of machine process. I would suggest that PROLOG developers may certainly hold a cognitivist theory of reasoning because they start with an applied perspective of arithmetical logic to computing restricted features of natural language. In this sense, Newell's heuristic belongs to Skinner's behaviourist paradigm whereas Colmerauer's algorithm belongs to Horn's arithmetic paradigm.

2.2. PROLOG origin: Predicate Calculus and the definition of clause, the Horn clause and resolution theorem proving.

(1) Predicate calculus and the definition of clause form.

According to Colmerauer & Roussel (1993: 26), PROLOG's form of logic is called Predicate Calculus. PROLOG integrates in the calculation of its clauses according to Robinson's (1965, 1970) principles of first order calculus predicate. It allows the treatment of a statement in arithmetic as a logic program. The logic program (Shapiro, 1989: 415) is a finite set of definite clauses. The first order calculus is the basic arithmetic that permits to elaborate the vocabulary of PROLOG (or any logical program.) as a statement and the terms of the statement. In each PROLOG clause, the programmer has to decide its vocabulary in terms of a set of predicates (a set of data), a function symbol (their relations), a goal (corresponding to the quest).

In predicate calculus, objects are called terms. Terms may be composed of a constant symbol, a variable symbol and a compound term. For example, a basic proposition in PROLOG:

'Mary is human' is written

Human (Mary)
'Human' is a constant symbol.

Man likes wine:
Likes (man, wine)
'man' is a variable symbol.

X owns donkey (x):
Owns (x, donkey (x))
'x' is a compound term.

In PROLOG, those linguistic propositions are called a goal if starting a program and an argument if belonging to a larger argument structure. If one wants to build up further proposition from the compound term (see 'x' above), logician and computer scientists use either logical connectives (not, and, or, implies and is equivalent to) or/and universal (for all v, …) and existential quantifiers (there exist v such that …).

The essential work done by Colmerauer and his team was to transform this form of logic into a programming language. Robinson's first order calculus forms the resolution principle at the heart of PROLOG. It performs the basic arithmetical logic guiding the declarative language. In LISP, FORTRAN or COBOL, the program is made of lines of code or sets of procedural instruction. In PROLOG, the program starts by defining clauses and sub-clauses, i.e. the domain of the problem. A question equals a set of calculation of the predicates of the clauses. It is a bit like a micro-world of logical procedures, a territory of sub-procedures on which basic calculation furnish the answer to a search procedure (a question).

Those terms in PROLOG are also called PROLOG clauses. Clauses are essential to understand the relation between logic and PROLOG. The form of clause specifies the different kind of relationship between variables within a clause. I will not review them here (see Clocksin and Mellish, 1984: 240-6). Those forms are essential for creating hybrid systems and implementing other forms of logic.

(2) Horn Clause and Resolution Theorem Proving.

To understand the resolution principle, one can say that every statement in natural language is written in a way to perform mechanical theorem proving. Alan Robinson's resolution principle offers a rule of inference permitting one proposition to follow another. The programmer's work is to decide which proposition implies another, and how a valid conclusion can be proceed automatically.

The resolution principle serves to treat a hypothesis as true or false according to arithmetic. Clocksin & Mellish set up a perfect example (1981, 2003: 249). One may try to show that Arthur is the king by setting up a problem is those terms:

'If every person respects somebody then that person is a king':

(1) person (f1(X)); king (X):-
(2) king(Y):-respects(f1(Y),Y).

'every person respects Arthur':
(3) respect (Z, Arthur) :- person (Z)

The resolution in 2 steps goes:
'Every person respects the king Arthur'
In arithmetical terms, you resolve the term 2 by the term 3, by matching them (called unification).

(4) King (Arthur):- person (f1(Arthur))

'Arthur is the king'
(5) king(Arthur); king (Arthur):-.

Let us notice that even if the matching of one side of a proposition with another is called unification, this does not cover all the cases of unification in PROLOG. Here, in our example, we rely on ordinary language to understand the logic of the relationship between the terms of the clause. In applied cases, unification may represent parts of a database, the dealing of information in parallel between two computers, or anything else relevant to apply.

There are also clauses that are restrictive. Horn's clause is one of them. The Horn clause[3] is a logical statement of the form:

A1, A2, … An.

where $n \geq 0$ and A is called a positive literal representing, in PROLOG, a unique statement symbolised by a letter. In PROLOG, this statement corresponds to a headless Horn Clause:

:- A1, A2, …, An.
A 'headless' literal is a statement of the form:
:-bachelor(x)

A 'headed' literal is as follows:
Bachelor(X):-male(X), unmarried(X)

In PROLOG, one can use linear input resolution as a way to say that one headless proposition is solved by the next proposition in line. The following proposition is considered to be a hypothesis which applied to the case solves the theorem. If we have the following proposition:

the mother (of the couple X and Y), X is the mother of John:

:- mother (john, X), mother (X, Y)

And being a mother (of the couple U, V) is being the female parent:

Mother (U, V):-parent(U,V), female(V)

Then, the mother (of the couple X, Y) is a female X who is a parent of John.

:-parent(john, X), female(X), mother(X,Y).

In PROLOG, those statements can take four different forms and, therefore, play different connecting roles : (1) to be a rule, (2) to be a fact or unit, (3) to be a negated goal and (4) a be a null clause. Complexity of the basic clause depends on the predicate one used in a program.

I have superficially overviewed the Horn clause to provide the reader with a sense of how PROLOG specialists define the way to attach a list of qualities (sub-clauses) to a definite clause. PROLOG can be defined as a collection of Horn clauses (Kahn & Carlsson, 1984: 117). This programming method is considered powerful because it offers formal means to draw inferences from a simple logical unit to more and more complex sub-units. This in turn allows carrying further correlations between the components of the sub-units.

2.3 Remarks about Syntax.

One important way one can appreciate the maturation of one language is by noticing the way communities of practitioners have tried to implement it. Different syntactic versions of PROLOG have started from various practical configurations of cases. Since 1972, in the early,

[3] The name "Horn Clause" comes from the logician Alfred Horn in his article (1951) "On sentences which are true of direct unions of algebras", *Journal of Symbolic Logic*, 16, 14-21. His theory of clause is the paradigm of PROLOG whereby sentences are considered to literally equate algebraic correspondence.

prototypic, developments of PROLOG by Colmerauer and his team, many computer scientists have added their stone to the edifice. One can trace the expansion of PROLOG to the 1980s whereby one sees the development of several types of syntax. For example, the original Marseilles interpreter (Kluzniak, 1984) is followed by the Exeter PROLOG (Fogelholm, 1984) and the Edinburg syntax (Clocksin, 1984a).

The Edinburg syntax needs special mention as it emerged over the years as the standard PROLOG language. This is certainly due to the active work of Clocksin and Mellish proposing a standard in their textbook *Programming in PROLOG*. This normalisation of PROLOG syntax is essential for the diffusion and constitution of a community of active programmers. PROLOG does not stipulate the kind of syntax one should use. Clocksin (1984: 94) provides a convenient frame within which one can start arranging clauses according to three arithmetical components[4] (constant, structures and variables). The Edinburgh syntax served as a guide from which the 2003 ISO norm of PROLOG is now established.

2.4 PROLOG and Logical Programming.

PROLOG today benefits from the software implementation work of many computer scientists (Warren, 1979, Clocksin, 1984: 94). It is a program that is efficient to use if one want to work out problems by representing objects symbolically according to a type of relationships. PROLOG (Colmerauer, 1985) is composed of a set of procedures whose execution depends upon both (a) the kind of predicate it is made of (the kind of logical relationship) and (b) the kind of clauses (the type of assertion) associated being either facts (a 'declaration': no procedure to perform) or rules (a 'procedure': a procedure to perform) represented by a term.

In the first level, we have a logical articulation clauses-predicate forming the procedures.

For example: meal (a, m, d) where a stands for appetizer, m for main and d for dessert.

In the second level, we have a logical articulation of the predicate themselves. This corresponds to the composition of a meal.
For example:

Main (m)
(m = sole)
(m = tuna)
(m = pork)
(m = beef)

Appetizer (a)
(a = radishes)
(a = pâté)

Dessert (d)
(d = fruit)
(d = cake)

Given that we have 4 main dishes, we have 16 possible combination of dishes (in arithmetic terms is $4^2 = 16$.)

Those clauses can be defined in the jargon as 'a fact' i.e. have no procedure to perform, or can be 'a rule' implying to perform a procedure. The computer system performs a procedure called 'a goal' by matching the different argumentative combination constituting the clause to the clause itself. This is considered as 'answering a question or a quest' about the predicate. It is clear, in the interpretive context of logical programming that 'answering a question' is a metaphor for performing a matching procedure. In our case, the quest is: produce a light meal with less than 10 units of calories.

The way to answer this question is equivalent to manipulate terms of arithmetic through the parameters of the clause (Horn, 1951) i.e. according to logical definition of the clauses. In this case, one has to: (1) associate caloric units to each components' of the meal, (2) define a light meal as under 10 units of calories, (3) perform basic arithmetic to calculate the caloric value of a meal, by associating each possible meal component with each possible caloric values under 10 (4) get the result of the light meals.

(1) Caloric value defines 8 units as follows:
units (beef, 3)
units (fruit, 1)
units (cake, 5)
units (pâté, 6)
units (pork, 7)
units (radishes, 1)
units (sole, 2)
units (tuna, 4)

4 1- Constants (also called 'atoms') are numerical data like integers (like 33456), floating point numbers (like 12.0), and negative integers (like -13), a sign or an operator (like +, >, ?, $ etc.) or alphanumerical (like a12). A constant working as an operator '+' in the case of X + 12 is written in PROLOG +(X, 12).
2- Structures (also called 'complex terms' or 'compound terms') are made of constants data (called 'functor') and one or more components (the number of elements called in mathematical term 'arity'). A structure is written in PROLOG: go(N,L) which means search L proprieties (to be defined) with the N properties (to be defined).
3- Variables are other terms which are constituted of an upper-case character or an underscored character followed by a string of numbers.

(2) Define the light meal:
PROLOG computes sums of 2 numbers (corresponding to 2 units). So the light meal, who is composed of 3 courses (appetizer, main and dessert) is calculated in two steps called 'little sum' and 'little successor' (intermediary sums) which, cumulated, stays at under 10 calories.

(3) Perform the arithmetic:
The first intermediary sum (little sum) calculates the appetizer and the main such as x + y = z with z < 10.

Light-meal (a, m, d) -> meal (a, m, d)

- Meal is a triplet a, m, d.
- The number of units of appetizers is x.
- The number of units of main is y.
- X + y = z. with z < 10.

The second intermediary sum is to calculate v = z + u.

- The number of units of dessert is z.
- Z + u = v. v < 10.

(4) Give the result of the composition and number (7) of light meals:

Light-meal (a, m, d) ?
(a = radishes, m = sole, d = cake)
(a = radishes, m = sole, d = fruit)
(a = radishes, m = tuna, d = fruit)
(a = radishes, m = pork, d = fruit)
(a = radishes, m = beef, d = cake)
(a = radishes, m = beef, d = fruit)
(a = pate, m = sole, d = fruit).

Putting aside, the way PROLOG developed, which reflects the natural history of Colmerauer and his colleagues' ideas, PROLOG open up for two important areas of inquiry marking its development: (1) implementation of other logical systems and (2) implementation of different proof procedures. Logic programming defines an epistemic field of research in computing formal logic whereby every computer scientists develop a sub-area of formal logic tested against the constraints imposed by the machine.

2.3. Liberal interpretation of syntax and innovation in algorithms

Clocksin (1984a) is certainly an active PROLOG promoter. He has reviewed areas where PROLOG has been put into use, namely (1) relational data bases and expert systems, (2) mathematical logic, theorem proving and semantics, (3) abstract problem solving and plan formation, (4) natural language understanding, (5) architectural design, site planning and logistics, (6) symbolic equation solving, compiler writing, (7) biochemical analysis and drug design. I propose to review some aspects that made PROLOG disseminations a reality. I propose to review 5 domains distinguishing PROLOG: (a) the technical advantages of PROLOG vis-à-vis LISP. This comparison underlines the viable alternative for programming more complex sets of data by, what I call, structural rather than sequential programming. (b) Those advantages of PROLOG software architecture show in the possibilities for other computer scientists to develop logical and parallel architecture. (c) The constitution of hybrid systems (like POPLOG) by combining PROLOG with sequential processing like LISP or object orientated processing like SMALLTALK. (d)The combination of other kind of hybrid version of PROLOG by using other language and using non-logical augmentation of the program and (e) the creation of micro-PROLOG as the development of PROLOG for education and end-users purposes (see 3.6).

3. SOME EXAMPLES OF DISSEMINATIONS OF PROLOG IN COMPUTER SCIENCES

In 1982, the debate between declarative versus procedural programming tends to confirm that PROLOG was a language that people were taking up. In the field of programming, there is a lot of invention but few long term survivors due to the change of hardware capacity and the corresponding software. PROLOG offers a software architecture that is a sort of tautological system. The program states facts and rules which define a problem and, in the same time, are programming instructions to solve the problem.

Kowalski (1988: 38) calls them declarative and procedural functions. Logical schemes representing data combines information along a linear procedure of first order logic. This way of combining syntax and semantics constituted the programming language of which LISP is the paradigmatic reference. The advantage of sequential logical programming is its definition in terms of inference rules, a clear formal semantics (linkage between set of words) and simple notation to create the knowledge base. The drawback of the sequential scheme of programming is the lack of organisational principle for the data constituting the knowledge base. Colmerauer (1985) says of PROLOG that it is 'a formalism for defining knowledge independent of the method of computation.'

PROLOG and the diffusion of logical programming took place around the 1980's. Researchers established the field with (1) textbooks on logic

programming, mathematical reasoning and PROLOG programming (Clocksin & Mellish, 1981; Clark & Tärnlund, 1982), (2) the organisation of workshop and conferences and (3) the creation of a journal of logical programming. It is noticeable that concise reference manual are essential for other practitioners to take up the task, try it and eventually expend it and apply it to its own set of problem. Computer scientists work with the manuals, the available published research papers on the topic and word of mouth between colleagues.

The essential point of the following section is that formal logical possibilities as well as practical one can largely extend original PROLOG capabilities. R Ennals J Briggs and D. Brough (1984) rightly observe that one can modify PROLOG logical machine by dealing explicitly with the Horn Clause subset of predicate logic. This is the application to specific domains and issues in the real world that specifies which sub-set of predicate logic computer scientists should use. The applications provide the occasion to suggest amendment to the formal logical and the basic Edinburgh syntax. At this stage, I rely on Clocksin's review (1984b) of PROLOG applications presented in 4 main domains: expert systems, interface of databases, design automation, and scientific tools. Research teams worked to develop special application with different contractors (industrial production, biological research, agricultural planning and information databases).

(1) In scientific tool, Sterling et al. (1982) came up with a tool of computer-aided algebra called 'Press system', derived from earlier research on another system called 'Mecho' (Bundy et al., 1982). The system aims at finding solutions of simultaneous transcendental equations. Darvas et al. (1983) have created a program that calculates derivations of regression models used in biochemical problems (un-synthesised drugs, pesticides and pharmacological compounds). (2) In expert system, Pereira et al. (1982) created an expert system for environmental resource evaluation called ORBI. It is a database system which provides information about intensive agriculture exploitation. It informs the user on climate, soil characteristics, planning permissions and geological data. (3) In design automation, Forrest and Edwards (1983) used PROLOG to translate one automation system (AFSM: Algorithmic Finite-State Machines) into another kind of automation (PLA: Programmable Logic Array). Barrow (1983) and Horstmann (1983) have developed different PROLOG programs to verify the correctness of digital hardware circuits and rules of circuit design. (4) In database interfaces, Warren and Pereira (1982) created a system called CHAT-80 which answers queries about geography.

Historically speaking, the implementations above are still domain specific programs. Those applications exploit the immediate advantage of logic programming. It is important to understand that those applications are possible since individual computer scientists started to approach PROLOG as a viable alternative to issues they experienced in other language such as power, efficiency and procedural approaches.

3.1 Computer Scientists' Interest in PROLOG: Power, Efficiency and Procedural Approach

PROLOG possesses advantage of manipulation, referred in computer scientists' jargon as 'powerful'. For example, this programming language permits flexible stipulation of symbols in the data structure. The logical variables articulate the symbols without specifying them further than calling them 'holes'. This is a considerable advantage when building up a program structure. It allows a division of labour whereby the computer scientist can specify the details of the data content and its structure afterward the details of the program structure. The logical variable provides parameters to construct the data structure in terms of selection and construction of objects.

PROLOG symbol manipulation performs better than LISP regarding data. It is simpler to perform recursive procedures and surface syntax. PROLOG can control its sequential flow of operation by backtracking when encountering a logical dead-end in the program. This permits to localise and perform test on a specific sequence of code (called 'generate-and-test', Clocksin, 1984: 93). LISP by contrast is unidirectional in its sequencing. If dead-end happen, one has to reconsider the logic of the whole string of code from the beginning.

PROLOG can be considered as a relational database with rules and facts. Its structure allows the programmer to change either the structure of the program, or the structure of the data by adding or removing clauses and data. Queries can be more sophisticated than in a sequential language precisely because they combine clause and data structures. This relational form of procedures allows multi-purpose use of a procedure. The programmer has to stipulate if a procedure is mono or multi-purpose.

3.2 Remarks on elegant solution

Clocksin (1984b) provides a beautiful example of the practical need for elegant solutions in programming. This is a topic of passionate debate among programmers precisely because it is at the

core of programmers' definition of professional competence. Clocksin shows an example where a programmer has written all the variables of a PROLOG clause for getting information in a database in FORTRAN. Although this is possible, he demonstrates how one can shorten dramatically the code by using less FORTRAN features that are extra-logical to PROLOG features. The result is a better readability and it runs faster on the machine to compute the result.

One can understand that seeing practical and elegant solutions corresponds to the level of fluency and experience one has in a given language. Obviously, the non-elegant solution, Clocksin is showing suggests that the programmer in question was more comfortable with FORTRAN coding and may not knew the alternative offered by PROLOG. The ability to use a language with skills affects the professional definition of the good programmers within the community. The implication of this professional definition leads to opportunities in terms of job placement, career mobility, research opportunities, contracts, etc.

3.3 The Development of Logical and Parallel Architecture

The maturation of the field offers new ways to approach PROLOG programming. Early development involves the emergence of individuals developing a line of inquiry. But subsequent developments of PROLOG, especially in building larger systems, shift the focus from personal dedication to the bureaucratic demands made upon organising teams of people and projects. Skills are implied but not the possession of a single individual. In the early 1990's work started on parallel processing whereby one computer can use more than one processing unit to execute a program[5]. This possibility of using processing power is key to the development of local area network (LANs). A single program can run simultaneously in various places and distributed databases render available data stored in more than one computer system. The database system keeps tracks of the place where data are stocked in order to be able to retrieve them at any time for any given user.

In this configuration, PROLOG solutions tend to integrate more bureaucratically defined sets of issues such as (1) databases, (2) parallel programming in distributed systems. In the domain of semantics research, effort concerns gathering ideas and means of investigation of other kinds of logic (not only based on first order predicate calculus but on temporal logic, modal logic, unifying logic, equational logic). Developers of PROLOG in parallel architecture focus on efficiency and seek to develop tools answering those demands such as module systems, polymorphic type systems, automatic debugging, software engineering development.

Gupta et al (2001: 474) indicates that PROLOG has two main advantages for the development of parallel processing: (1) PROLOG has a clean semantics that makes compiling easier and the system run-time efficient, i.e. rapid and correct. It offers a clear articulation of variables. This is an important advantage since it is difficult to divide a program without interference of one part of the program with the other(s).

For the programmer who works with parallel computing, PROLOG algorithm permits to treat every specification of a given problem as a variable unit coherent with the rest of the programming logic. In parallel computing, there is a potential issue of articulation of specific variables. This may induce a lot of irregular computation (due to problem generated by symbolism in algorithms and the complexity of data structure due to independence of some variables). In other words, there is an inflation of code procedures that are difficult to unify under the same logic (and therefore to keep track and eventually modify) in reason of the diversity of processes they accomplish. (2) It is faster to run a program since several parts of a single program are handled by different processors. This profits PROLOG as well as any other programming language.

There are three approaches to parallel execution of logic programs: (1) the implicit exploitation of parallelism where parallel execution is done without the explicit intervention of the programmer and (2) the explicit construction of parallel solution with concurrent semantics or modification of PROLOG semantics. (3) The hybrid solution which consists in an extension of PROLOG language with possibility of detailed manual parallelisation (for example, &-PROLOG system (Hermenegildo and Greene (1991), CGE and ACE languages (Pontelli et al. 1995, 1996), DASWAM (shen, 1992).

There is a tremendous amount of PROLOG implementation of explicit parallel solution. I will review some in the next section 3.4. For the sake of simplicity, I will only mention, below, implicit parallelisation, i.e. PROLOG parallel processing to implicit methods.

The idea of implicit parallelisation is to treat indeterminism in PROLOG operational semantic to perform parallel operation without modifying its

[5] Parallel processing is not to be confused with multitasking which is a partition of the processor itself in order to perform several programs at once.

overall semantics. To achieve this goal, the programmer translates non determined choices into parallel computation clauses. Gupta et al. (2001: 482-3) present the three classical forms of parallelism (Conery and Kibler, 1981):

1- And-parallelism[6]: when more than one sub-goal[7] solves a query, the system chooses some of those sub-goals to be executed in parallel:

While (Query ≠ ø) do
Begin
$Select_{literal}$ B from Query;

This is useful in list-processing applications to launch multiple searches, various constraint solving problems to take into account several variables at once (let say speed and aerodynamic) and system application.

2- Or-parallelism: when more than one clause solves a query, the system chooses between several clauses to be executed in parallel:

While (Query ≠ ø) do
Begin
$Select_{clause}$ (H:- Body) from Program;

This option is useful in applications exploring large search spaces via backtracking in expert systems, optimisation problem, certain kind of language parsing, and deductive database systems.

3- Unification parallelism: when the argument of a goal can be associated with the argument of a clause:

Argument 2 can be associated with argument 1.

Unify(Arg1, Arg2)
If (arg1 is a complex term f(t1, …, tn) and
Arg 2 is a complex term g(s1, …, sm)) then
If (f is equal to g and n is eaul to m) then unify (t1, s1), unify (t2, s2), …, unifiy (tn, sn)
Else
Fail

Unification parallelism has not been a major research focus in parallel logic programming. Nevertheless, those techniques of parallelism are key features for the exploitation of parallel processing systems. This does not go without its problems of exploitation, which I will not review. For the sake of this paper, suffice it to say that the idea of exploiting parallelism is to achieve greater performance for programs in terms of speed of execution. A lot of work has been done in the domain of shared memory allowing a single storage of data to be shared among users.

During the mid-80's, the development of interest in PROLOG explicit parallel solution has brought the development of new implementations such as PARLOG, GHC, KL1 and concurrent PROLOG. In 1982, the Japanese fifth generation computer system project became one of the most prominent body of developers of those solutions.

3.4 The Japanese Fifth Generation Computing Systems (FGCS)

PROLOG was given new impetus when Japanese computer scientists and government research agencies launched their Fifth Generation Computing Systems (FGCS). This program (Moto-aka, 1981; Furukawa, 1992; Fuchi et al. 1993; Englemore & Feigenbaum, 2000; Feigenbaum and McCorduck, 1979) brought over 100 researchers in a committee to investigate the development of new information processing technologies. The focus was on both knowledge information processing technology and improved parallel computer technology. In term of knowledge information processing, the idea was to consider PROLOG as a base to design programming language that will contribute to build a new software culture.

The FGCS wanted to develop a completely new information system taking into account software compatibility with existing systems. Compatibility becomes an issue when one changes programming language. The FGCS idea is to consider logical programming as a foundation for different information processing including programming itself, software engineering, database and knowledge information processing. In this context, PROLOG's importance is contained in FGCS working hypothesis: how to use PROLOG to bridge the gap between knowledge information processing and parallel processing.

I will not enter into any technical description of FGCS due to the fact that this integrated project covers a large amount of new software technology. This implies that PROLOG has to be approached along the line of the detailed organisational aspects of its project development implying the coordination of human resources and monitoring meetings with corrected technical guidelines. Ueda (1993) has provided the best account available of FGCS development reporting the technical feature of the project as well as the managerial decision and constraints that emerge out of it.

[6] There are several more kind of and-parallelism such as independent and-parallelism (IAP) and dependent and-parallelism.

[7] A sub-goal is also called an atomic formula and, in PROLOG, a literal.

To get a sense of the complex integration FGCS project suppose, let us remind that computer scientists research were creating a completely new computing environment whereby logical programming played the role of integrator between (1) a language in concurrent logic called Guarded Horn Clauses (GHC), (2) a parallel computer called the Parallel Inference Machine (PIM) and (3) programming methods as well as applications in GHC. In this work environment, decision has been taken that PROLOG will serve to start the first Kernel Language (KL0). It appeared quickly that another language had to be designed (KL1) shifting toward concurrent logic to handle the original task of parallel computer architecture, programming and applications. This is during the conception of KL1 that, in 1984, that Ueda proposed the Guarded Horn Clause (GHC) as a solution to computer scientists' frustration to re-develop a parallel programming language. He proposed (1993: 65) Guarded Horn Clause as a solution to support GHC for their kernel language[8] (KL1). The importance of GHC is to furnish the basic framework of concurrent computation where kernel languages are attached.

The KL1 started as a language which dealt with issues of parallel processing in knowledge information (such as knowledge representation, knowledge base management, cooperative problem solving). The FGCS embraces concern on how to reconcile logic programming and object-orientated programming. The research leads toward an alternative to PROLOG which is concurrent logic programming. In 1982, works on concurrent logic programming came out (Takeuchi, Shapiro) and influenced their decision to proceed further. In 1983, FGCS assembled a task group establishing that KL1 will be based on concurrent PROLOG with the following characteristics: (a) general purpose language accepting concurrent algorithm, (b) two syntactic constructs are added from the original logical programming framework, (c) coherent as the original logical programming and (d) possible adaptation of logic programming into concurrent logic programs.

The amount of work that goes into the establishment of a new platform is phenomenal. This kind of work demanded, besides building up the experience and finding technical solution for its feasibility, demanded a careful management of the personal. It is obvious that any change or decision for another technical solution reduce to nothingness months or years of development. It also demands from the programmer themselves, even if offering a good technical solution, to get a committee agreement to integrate it into the guideline. It reminds the weight carried by the intermeshing between technical and managerial decision in large projects over the development itself.

The FGCS interrogates the sustainability of research for long period of time. One can say that the Japanese Institute for New Generation Computer Technology (ICOT) where the FGCS took place has created an environment increasing the technical know-how of computer scientists, it also put constraints on the personal interests of many of them who did not aim at working on concurrent logic programming or which research activities were not linked to concurrency and parallelism. Technically speaking, the design of the Kernel language was a history of simplification passing from concurrent PROLOG to GHC and from GHC to flat GHC.

Due to the acceleration of computing R&D, it suggest that political decision to launch such project such as FGCS comprises a fine understanding of the implication of large research endeavour. For example, FGCS has to face two problems: the adoption of FGCS solutions and systems by other constituencies and (2) its economical viability. Retrospectively, the integration of new technology is better secured from already existing and commercially successful products. It demands the coordination of international corporations and industry leading to agreements between each other. Finally, it suggests that individual realisations are subsumed under the technico-commercial umbrella.

3.5 The Use of Other Language and Non-logical Augmentation

During the early 80's, there was a number of attempts to use other languages in conjunction with PROLOG (Santane-Toth 1982, Szeredi, 1982). From the hardware point of view developers could use the DEC VAX computer series with VMX or UNIX operating system which support compilers for three languages POP-11, PROLOG and small LISP. Those developers were engaged in the exploration of hardware possibilities thanks to the software. In this particular case, Mellish and Hardy (1984: 117) wanted to develop a model for a hybrid compiler. They indicated that PROLOG was not the best program to write screen editor or network interface controllers but conventional application would gain by adopting PROLOG for CAD systems, statistics packages or relational databases.

The development of multi-language environment envisaged to develop logic programs with procedural language such as LISP or POP-2. Robinson and Sibert (1982) have developed

[8] The kernel language is a language that permits to link parallel hardware and application software. Three versions of Kernel Language have been issued (K0, K1 and K2).

LOGLISP, Genesereth and Lenat (1980) MRS, and Komorowski (1982) QLOG have integrated logic programming with LISP. The motivation for those developments was to offer the programmer with multi-language programming systems. Developers have different objectives when programming. For Mellish and Hardy, the development of a hybrid version of PROLOG demands the definition of a clear model on how PROLOG data structure and control mesh with procedural language.

They critically advance that providing PROLOG with convenient connection to LISP is a complete solution. To justify their research endeavour, Mellish and Hardy indicate several advantages to adopt their position: (1) creation of backtracking point in the procedural language (2) flexible control of PROLOG solution by procedural language (3) remove the asymmetry between PROLOG and POP-11 where both languages are compiled by the same virtual machine. (4) The avoidance of building syntax and built-in predicates for PROLOG.

The early stage of Mellish and Hardy's researches has brought the implementation of POPLOG virtual machine, i.e. the formal logic of a common compiler between POP-11 and PROLOG. The development of a semantic called 'continuation-passing' is not simply the implementation of 'subroutine calling'. It is exploiting the Warren (1977) principle whereby one can represent the head of a clause as in-line instruction. For example (Mellish & Hardy, 1984) in PROLOG the head of a clause is:

(Here we are dealing with connecting (unify) elements (x and m with y) in a database structure.)

```
Unify (y, conspair (x, m), continuation)
```

In POPLOG, the unification of elements is done in-line[9] (rather than creating a control stack frame for UNIFY in PROLOG).

In classic PROLOG, the unification of data demands the building of concrete data structure. As we have seen in the section 2.1 if a clause mentions a list structure in its head, the list has to be constructed in order to match an item, a list of items. In contrast, Mellish and Hardy's POPLOG is an implementation of the Warren (1980) principle of 'task recursion optimising'. This model represents explicitly the structure of the data (how data relate to each other). The data structures are not mentioned unless it is necessary to specify a variable that is not instantiated. In this sense, the head of a clause plays a different role than PROLOG's declarative procedure. Instead, it performs a test on the type of data that can be failed or accepted.

3.6 Micro-PROLOG: the development of PROLOG in education

In AI, there was a large movement of computer scientists emulated by the lead of Seymour Papert who saw in programming a way to address issues of education. For example, in his 1980's book *Mindstorm*, he defends the idea that micro-worlds (well defined environment that could be subject to computing through the use of programming interfaces) are incubators of knowledge. His educational view is a fairly straightforward extension of Newell's heuristics for problem solving. He wants to find areas of computation applied to principles that he sees as educational. In 1982 (Ennals, 1982) and 1984, Ennals, Briggs & Brought, (1984) have identified similar ideas of using PROLOG programming as a micro-world without an explicit educational theory. It is at the occasion of works on PARLOG (parallel implementation of PROLOG) and other work on databases and parallel architectures that their interest arose to provide relatively simple interfaces to be able to use PROLOG. Ennals, Briggs and Broughts (1984: 380) raise three concerns when it comes to educating people to use PROLOG. (1) One has to understand the problem of learning PROLOG logic, (2) the logical extension from the original PROLOG are necessary for pedagogical investigations and (3) users' tools must be developed to match users' development of logical programs.

(1) Ennals, Briggs & Brought, (1984) worked with children as well as adults. They composed their pool of amateurs in PROLOG. The implementation of an educative version was done in micro-PROLOG (McCabe, 1980; Clark, Ennals and MCCabe, 1981, Ennals, 1982). One must remember that the hardware, therefore the availability of Micro-PROLOG is an issue to consider. It has to be available across a large range of micro-computers. An interface program was developed permitting easy usage. Ennals (1984: 378) gives an example of simplified notation used to program with it:

```
Cow eats grass
Grass is –a plant
X is herbivore if x eats y
 And y is-a plant
```

The simple program is running on a different machine and different implementation of PROLOG (either micro-PROLOG or PARLOG). Ennals reports that he uses this program either on microcomputer or mainframe.

[9] See annex for Mellish and Hardy's example of hybrid program created in POPLOG.

Ennals et al. (1984) reports, unsurprisingly from my point of view, that the different representation of operator and variables may become a difficulty for the user. In fact, the amateur is unaware of the syntax that presides to PROLOG. But programming in PROLOG is an exercise in formal syntax. The problem is obviously one of translation from computer scientists' practice and jargon into a pedagogic teaching of syntax. Ennals (1984: 379) indicates that issues for amateur in PROLOG are:

- The problem of PROLOG rules in terms of 'informal specification to executable specification in Horn clause logic'.
- Starting to learn the operation on database, because the amateur can be a user of databases.
- Ennals, Briggs & Brought offered a simplified notation system dealing with adding information to the database (rather than creating new assertions.) It is the beginning of teaching the amateur to program.
- Rather than use an instruction to find missing information, the amateur learns to use symmetric dialogue technique. For example:

X citizen-of y if x born-in y
User: where John citizen-of x?
Machine: where john born-in x?
User: England
Machine: John citizen-of England.

(2) The objective of educating people in using PROLOG is to show the different option of logic (in the jargon 'the power of logic') by extending PROLOG facilities. It is done by building other predicates than the standard full predicate logic on standard PROLOG (Kowalski, 1974, 1979, 1981, 1982a&b). Micro-PROLOG adds three predicates: negation by failure, Is-All and For-All primitives.

The idea of adding primitives is to provide the user with the ability to deal with reading and printing facilities, to access the internal logic of the program, to produce queries at all level of the program (object or meta-level), non-textual option (graphic, interactive graphics, sound, iconographic programming and logical spreadsheet).

(3) In 1984, there was already a concern for matching the advance of other systems (for example PC/MS DOS (Disk Operating System) in terms of screen and line editor and modules for the development of other structures. As we have seen in the sections before, flexibility of implementation is (as a result in interest in the program) necessary for implementing facilities. It seems to me Ennals, Briggs & Brough go beyond the simple amateur when they suggest that work can be done on the operating system and other parts of the host machine. For example, they suggest the user should be able to engage in printing, word processing, directory & memory checking. It supposes that functional knowledge developed the proper identification of how to act upon them. I suggest that Ennals, Briggs and Brought move beyond the simple amateur skills toward those of a tester. In their view, the user is somebody who would search for solutions for flexible error messages, varied conversational exchange with PROLOG modules and other user-interface systems (Hammond, 1983). In early human computing interaction research domain, Agre (1995) recalled that for programmers, the user is concerned with issue of access, improvement or exploration of system solution. It suggests that users can directly engage in object or meta-level customisation in PROLOG without resorting to PASCAL or machine code. Researches developed in those areas but the development of micro-PROLOG remained an affair for full time specialists. It is interesting to note that PROLOG specialists have used amateurs as 'statistical indicators' for their system test. It demonstrated to PROLOG developers that its improvements could be done more fruitfully by developing other strings of declarative sentences rather than concentrating on the debugging of anomalous cases. Ennals, Briggs and Brought suggests that parallel implementation in logic will solve most of those problems.

4. CONCLUSION

4.1. Logical Programming, Artificial Intelligence critiques and its implications for Social Studies of Technology.

I have made no attempt in this paper to investigate how, for example, the Horn's theory translates algebraic formulae into sentences. At the ideological level where professional practice becomes professional psychosis (E.C Hughes, 1984), one is allowed to warn against the danger to see computer scientists treating formal grammar of syntax or semantics as a description of a language in working order. This philosophical confusion between formal grammar and ordinary language (Button et al. 1995: 247) applies in the same way to PROLOG specialists dealing with its arithmetic and syntax. I have a tendency to treat computer scientists' primary cognitivism as a side effect of their profession rather than the core of their practices. Of course, cognitivism across discipline brings insidious and long lasting effects upon the understanding of social phenomenon such as technological development. It may be possible that PROLOG has introduced in the mind of many computer scientists, Gottlob Frege's idea that language logic is in effect co-extensive of arithmetical forms. To a certain degree, the confusion between what computer scientists take

language to be and the logical options offered by arithmetical hypotheses are constitutive of their research object and way of working out solutions. In other words, one deals with professional practices and not with confusion held by an outsider of their work scene. The implication of these remarks are threefold: (1) the demands of knowledge on those who engage in social studies of science and technology (next paragraph) (2) the achievement of a fair grasp of the epistemic practices of the professionals (section 4.2) and (3) the grasp of practitioners' practices within the right environment of understanding reporting the dynamics of the profession rather than the effect of their mental attitude (section 4.3).

For social scientists, the difficulty in studying computer programs is their own basic knowledge about programming and its limitated experience in applied settings. Clocksin and Mellish's (1981) good advice is to program. In doing this, one can de-code its technical vocabulary from its appearance of everyday expression into its technical domain of belonging. For the non-specialist, the technical jargon may certainly be unclear but more importantly, the technical work itself remains incomprehensible. It demands some training, learning and acquaintance with the machine functions. For computer scientists, the use of language is metaphoric by default not by choice as talking *about and in terms* of the machine function is the same thing for the good practical reason to be able to communicate the function to each other. As it is difficult to communicate what the machine does with symbols or equations in our everyday communications, one has to rely on the jargon which is metaphorical by definition. 'Language parsing' reflects different arithmetic methods of association between words or set of words, 'declarative sentence' is a statement, 'problem solving' is a deductive procedures. In PROLOG, 'statement', 'clause' and 'variables' are based on formal syntax and arithmetic. As a high level language, its meaning can be either practice or reported from the function to which practitioners have put them to use. In this sense, the survey is trying to reflect some of the concerns that animate the professional community. The first noticing is that PROLOG is not simply a tool in view of achieving a result but also an object of research in itself. This help to define the boundaries of the PROLOG research communities according to their work products: the implementations and the context that make their research activities real.

4.2. Computer sciences' practical epistemologies.

One can take the view that PROLOG is a paradigm (Kuhn, 1962) of its own, considering that arithmetic logic is a form of programming. From the point of view of the studies of science and technology, this does not assume an internal versus external (Pinch, 1986: 14) picture of technology whereby the epistemological decision would take place inside the brains of computer scientists and eventually, in a distributed way, among their colleagues assuming a similar mind set. There is a commonality in knowledge and PROLOG communities are formed, as many other academic communities, on a competence, i.e. a series of knowledge one acquires through education and professional practices.

The paradigm of PROLOG programming contains certainly a view of language and intelligence that are criticisable. I suggest that those views also belong to the computer scientists' communities. In this sense, they define professional hope and expectation in future development. It is beside the point to decide if those hopes have foundation in technical possibilities. Those hopes are also part of the rhetoric of science and technology which plays its parts in the formulation of research grant applications, the public interest in projects and the political importance attached to it within or outside the community of practitioners.

4.3. Professional Dynamics.

In the 1970's, at the early stage of the shaping of PROLOG, the developers had to prove the worth of their endeavour to themselves and other colleagues established in the field. Taking a view on PROLOG from the perspective of early Artificial Intelligence is informative in this respect. It shows how PROLOG as a new trend of thought has to be defined vis-à-vis already established research programs (LISP, PLANNER…) and compare to professionals having or establishing their own status in computing. During the 1980's, the number of professionals working on PROLOG has expended. PROLOG has offered numerous opportunities for computer scientists to reconsider its language as a resourceful domain of experimentation. It has been approached in many different ways, from the arithmetical point of view of theorem proving (Kowalski), the mechanisation of logic (Robinson) or address issues of development (Colmerauer & Roussel, 1992) and implementations (Clark and Tärnlund, 1982; Campbell, 1984) all of which shape research communities and their projects.

Clocksin and Mellish's (1981) first textbook is a good indication of a coming generation of international students and developers. The take off of PROLOG in the Europe, Israel, Canada, Japan and Australia (but not the US) suggests high professional mobility which corresponds to the rapid development of the profession in general.

PROLOG has served different purposes in academic research, large industrial projects, state funded programs, personal related computing and education. In the UK, it seems that an active academic environment supported the expansion of PROLOG computing through research projects, development and implementations.

In the 1990s, one can perceive professional trend traversing PROLOG and programmers' interests in implementations. They reflect the socio-economical forces shaping their coming project. The last phases of development of parallel computing indicate the beginning of the unification of different techniques. Effective bureaucratisation of the profession demands a tight management well informed of R&D economic and market value. For example, the FGCS suggests that basing decision making on technical prospective is risky for technological development itself. Today, unified technologies show that large R&D endeavour demands diversified financial back-up. The use of private bank loan or public support is explored along the line of using revenues of intermediary products available in the market.

The dynamism of computing shows that thinking in terms of large explanatory framework cannot cover the complexity of development which acts under financial, competitive and organisational constraints. In the social studies of computer science, specific software developments have been largely under-studied. It remains that computer scientists themselves have done most of the reference works in the history of computing. It is certainly the best source of information one may expect, although not exempt of critics. It demands from social studies of technology to formulate accordingly the epistemic frameworks within which computer scientists' current practices develop. In this paper, I proposed a preliminary survey sensitising scholars to the infra-level of change within programming practices seen as a knowledge domain.

References

P. Agre (1995) 'Conceptions of the user in computer systems design', in Peter J. Thomas (ed.) *The Social and Interactional Dimensions of Human-Computer Interfaces*, Cambridge University Press, 67-106.

K. Bowen (1979) 'Prolog', ACM/CSR-ER: Proceedings of the 1979 annual conference, ACM, NY: 14-23.

G. Button, J. Coulter, J. Lee and W. Sharrock (1995) *Computer, Mind and Conduct*, Polity Press, Oxford.

K. Clark and Gregory (1986) 'PARLOG: A parallel implementation of PROLOG, *ACM Trans. On Programming Languages Systems*, vol. 8, no. 1: 1-49.

K. Clark et al. (1982) K.L. Clark FG McCabe & P. Hammond 'PROLOG: A language for Implementing Expert Systems' in Hayes et al. (1982) PJ Hayes D Michie and YH Pao (eds.) *Machine Intelligence 10: Intelligent Systems Practice and Perspective*, Chichester, Ellis Horwood.

W. F. Clocksin and C. Mellish ([1981]2003) *Programming in PROLOG*, Springer-Verlag, New York.

W. F. Clocksin (1984a) 'An Introduction to PROLOG', in Tim O'Shea & Marc Eisenstadt (ed.), *Artificial Intelligence: Tools, Techniques and Applications*, Harper and Row Publishers, New York: 1-21.

W. F. Clocksin (1984b) 'Logic Programming and PROLOG' in Fred. B. Chambers (ed.) *Distributed Computing,* Academic Press: 79-110.

J. Chassin de Kergommeaux, Philippe Codognet (1994) 'Parallel Logic Programming Systems', *ACM Computing Surveys*, vol. 26, no. 3, September 1994: 295-336.

A. Colmerauer (1982) 'An interesting subset of natural language' in K. L Clark & S-A Tärnlund (eds.) Logic Programming, Academic Press, London: 45-66.

A. Colmerauer (1985) 'PROLOG in 10 figures', *Communications of the ACM*, December, volume 28, number 12: 1296-1310.

A. Colmerauer & P. Roussel (1992) 'The Birth of PROLOG' *ACM SIGPLAN Notices*, volume 28, no. 3, March 1993 and In Thomas J. Bergin and Richard G. Gibson, (eds.) (1996) *History of Programming Languages*, ACM Press/Addison-Wesley: 331-367.

J. Cohen (1988) 'A View of the Origins and Development of PROLOG', *Communications of the ACM*, vol. 31, no. 1, 26-36.

J. Cohen (2001) 'A Tribute to Alain Colmerauer', *TLP* 1, no. 6: 637-646.

R. Ennals (1982) *Beginning micro-PROLOG*, Ellis Horwood and Heinemann, Chichester and London.

R. Ennals ([1982], 1984) 'Teaching logic as a computer language in school' in M. Yazdani (ed.) New Horizons in Educational Computing. Ellis Horwood.

R. Ennals, J. Briggs & D. Brough (1984) 'What the naïve user wants from PROLOG' in J.A. Campbell (ed.) *Implementations of PROLOG*, Ellis Horwood Limited, Chichester: 376-386.

R. Ennals and J. Briggs (1984) 'Logic and Programming', Steve Torrance (ed.) *The Mind and the machine: philosophical aspects of artificial intelligence*, Ellis Horwood Limited publishers, Chichester: 133-144.

R. Fogelholm (1984) 'Exeter PROLOG – some thoughts on PROLOG design by LISP user' in J.A. Campbell (ed.) *Implementations of PROLOG*, Ellis Horwood Limited, Chichester: 111-116.

K. Fuchi, R. Kowalski, K. Furukawa, K. Ueda, K. Kahn, T. Chikayama, E. Tick (1993) 'Launching New Era', Communications of the ACM, vol. 36, no. 3, March 1993: 49-100.

K. Furukawa (1992) 'Logic Programming as the integrator of the firth generation computer systems project', *Communications of the ACM,* March 1992, vol. 34, no.3: 82-92.

G. Gupta, E. Pontelli, K. A.M. Ali, M. Carlsson, M. V. Hermenegildo (2001) 'Parallel execution of PROLOG programs: a survey', *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 4: 472-602.

C. Hewitt. *Planner: A Language for Proving Theorems in Robots IJCAI* 1969.

C. Hewitt. *Procedural Embedding of Knowledge In Planner IJCAI* 1971.

C. Hewitt. 'The Challenge of Open Systems', *Byte Magazine*, April 1985.

C. Hewitt and G. Agha (1988) 'Guarded Horn clause languages: are they deductive and Logical?' International Conference on Fifth Generation Computer Systems, Ohmsha 1988. Tokyo. Also in *Artificial Intelligence at MIT*, Vol. 2. MIT Press 1991.

E. C. Hughes (1984) *The Sociological Eye*, Transaction Books, New Brunswick.

B. Kornfeld and C. Hewitt. 'The Scientific Community Metaphor' *IEEE Transactions on Systems, Man, and Cybernetics*. January 1981.

R. Kowalski (1974) 'Predicate Logic as a Programming Language', Proceedings IFIP Congress, 569-574.

R. Kowalski (1979) *Logic for Problem Solving*, North Holland, NY.

R. Kowalski (1979) 'Algorithm = Logic + control', *Communications of the ACM,* volume 22, number 7: 424-436.

R. Kowalski (1982) 'Logic Programming and the 5th generation' in *State of the art of 5th generation computing*, Infotech, Pergamon.

R. Kowalski (1988) 'The early years of Logic Programming', *Communications of the ACM*, January.

H. Roth and McDermott (1978) 'An interface matching technique for inducing abstractions, Communications of ACM, 21.

J. McCarthy. (1958) 'Programs with common sense', *Symposium on Mechanization of Thought Processes*, National Physical Laboratory, Teddington, England.

C. Mellish and S. Hardy (1984) 'Integrating PROLOG in the POPLOG environment' in J.A. Campbell (ed.) *Implementations of PROLOG*, Ellis Horwood Limited, Chichester: 147-162.

T. Moto-oka, (1982) *Fifth Generation Computer Systems: Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, North-Holland Publishing Company, Amsterdam.

T. Pinch (1986) *Confronting Nature: the Sociology of Solar-Neutrino Detection*, Reidel Publishing Company, Dordrecht.

J. A. Robinson (1965) 'A Machine-Oriented logic based on the resolution principle', *Journal of the ACM*, vol. 12, no. 1, January 1965: 23-41.

J. A. Robinson (1969) 'A Note on Mechanizing Higher Order Logic', in B. Meltzer, D. Michie (eds.) Machine Intelligence 5, Edinburgh University Press, Edinburgh: 123-133.

J. A. Robinson and E. E. Silbert (1982a)'LOGLISP: An Alternative to PROLOG', in Hayes, P. J., Michie, D., Pao, Y-H. (eds.) (1982) Machine Intelligence 10: Intelligent Systems – Practice and Perspective, Ellis Horwood, Chichester.

J. A. Robinson and E. E. Silbert (1982b) 'LOGLISP: Motivation, Design and

Implementation' in K. L. Clark & S-A. Tarnlund (eds.) (1982) *Logic Programming*, Academic Press, New York.

P. Roussel (1975) *PROLOG: Manuel d'utilisation*, Groupe d'intelligence Artificielle, Université d'Aix-Marseille, Luminy, France.

E. Shapiro (Ed.) (1987) *Concurrent Prolog: Collected Papers, vol. 1 & 2,* MIT Press, Cambridge, MA.

E. Shapiro (1989) 'the family of concurrent logical programming languages', *ACM Computer Survey*, vol. 21, no. 3, September 1989: 413-510.

S. Uchida and K. Fuchi (1992) *Proceedings of the FGCS Project Evaluation Workshop* Institute for New Generation Computer Technology (ICOT). 1992.

J. Slagle. 'Experiments with a Deductive Question-Answering Program' *CACM*. December, 1965.

Takeuchi et al. (1982) *New Unified Environment*, Nippon Telegram and Telephone Public Corporation.

K. Ueda (1986) 'Guarded Horn Clauses', ICOT Technical Report TR-103, Institute for New Generation Computer Technology (ICOT), Tokyo, 1985.
Revised version in *Proc. Logic Programming '85*, Wada, E.(ed.), Lecture Notes in Computer Science 221, Springer-Verlag, Berlin Heidelberg New York Tokyo, 1986, pp.168-179.
Also in *Concurrent Prolog: Collected Papers*, Shapiro, E.Y. (ed.), The MIT Press, Cambridge, 1987, pp.140-156.

T. Winograd (1971) Natural Language Processing, IJCAI 71, September,

T. Winograd (1972) *Understanding Natural Language*. Academic Press, New York.

D. H. D. Warren (1977) Implementing PROLOG, *Research report 39 & 40,* Department of AI, University of Edinburgh.

D. H. D. Warren (1980) 'An improved PROLOG Implementation which optimises tail recursion', in Tärnlund, S-Å, *Proceedings of the Logic Programming Workshop*, Debrecen, Hungary.

ANNEX:

This is Mellish and Hardy (1984) example of the unification of elements in POPLOG by combining PROLOG notation with task recursion notation:

```
Define member (x, y, continuation)
Vars t;
Deref (x) -> x;
Deref (y) -> y;

(;;;deref gets the value of a PROLOG variable if it has one ;;; if
not, it gets the last 'ref' in the chain)

if isref (y) then
conspair (x, consref ('undef')) -> cont (y);
continuation ( );
'undef' -> cont (y)
Elseif ispair (y) then
Deref (front (y)) -> t;
If isref (x) then
t-> cont (x);
continuation ( );
'undef' ->cont (x)
Elseif isref (t) then
X -> cont (t);
Continuation ( );
'undef' -> cont (t)
Else
Unify (x, t, continuation)
Endif
Endif
;;; code for second clause of definition enddefine;
```