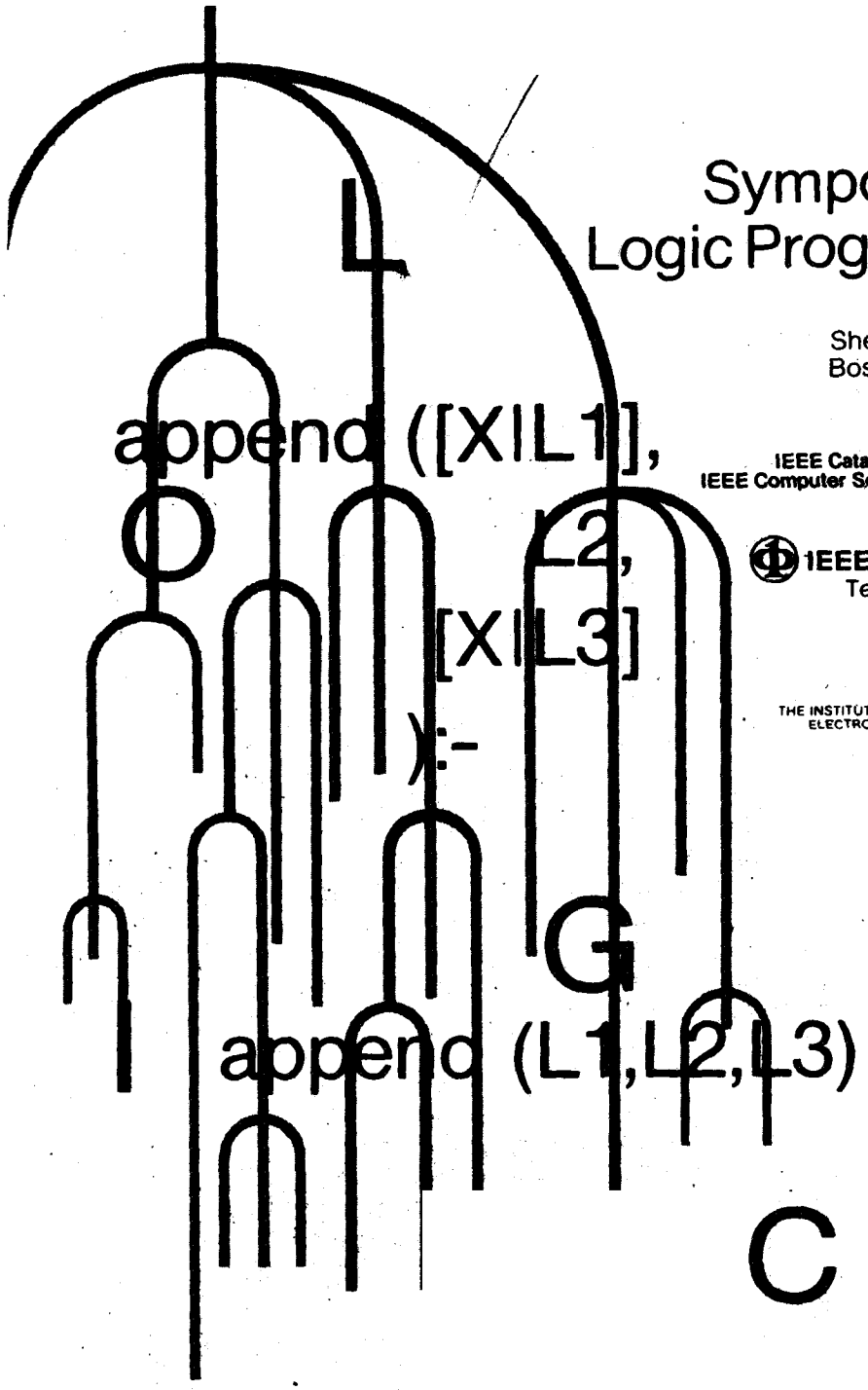
1985 SYMPOSIUM ON LOGIC PROGRAMMING



1985 Symposium on Logic Programming

July 15-18, 1985
Sheraton Boston Hotel
Boston, Massachusetts

ISBN 0-8186-0636-3
IEEE Catalog Number 85CH2205-3
IEEE Computer Society Order Number 636

Sponsored by:
IEEE Computer Society
Technical Committee on
Computer Languages

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC
IEEE

IEEE COMPUTER SOCIETY PRESS

Published by IEEE Computer Society Press
1730 Massachusetts Avenue, N.W.
Washington, D.C. 20036-1903

ISBN 0-8186-0636-3 (Paper)
ISBN 0-8186-4636-5 (Microfiche)
ISBN 0-8186-8636-7 (Casebound)
IEEE Catalog Number 85CH2205-3
IEEE Computer Society Order Number 636

Order from: IEEE Computer Society
Post Office Box 80452
Worldway Postal Center
Los Angeles, CA 90080

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

Preface

This second symposium on logic programming demonstrates the continued interest by researchers in the United States as well as by researchers abroad in this new area of computer science. About 80 papers were submitted to the **1985 Symposium on Logic Programming.** From these papers, we selected less than one-third. For the selection we had the invaluable help of the referees, whose names are listed on page vii. In addition, at least one member of our technical committee refereed each of the submitted papers. This screening process reassured us that only the highest quality papers would be accepted. The breadth of scope of the papers is impressive. Topics covered include theory, semantics, extensions, parallelism, language issues, implementation, and software engineering.

We undoubtedly benefited from the experience gained in organizing the first symposium, held in Atlantic City in 1984. Nevertheless, it is always a considerable task to manage a conference of this size. This effort was made easier by the outstanding and friendly cooperation among the members of the technical committee and the existence of the CSNET. Furthermore, we believe in the advantages of logic programming: a data base written in Prolog by John Conery enabled us to do an efficient and fair matching of papers to referees and also helped us considerably in the selection process.

Finally, it should be pointed out that a substantial percentage of the accepted papers were authored by researchers from abroad. This gives our symposium a truly international flair. We hope that this trend will continue in future symposia.

Jacques Cohen
John Conery
Program Cochairmen
1985 Symposium on Logic Programming

Conference Committee

General Chairman

Doug DeGroot
IBM T.J. Watson Research Center

Program Cochairmen

Jacques Cohen
Computer Science Department
Brandeis University

John Conery
Department of Computer and Information Science
University of Oregon

Technical Program Committee

Jacques Cohen *(Brandeis)*
John Conery *(Oregon)*
Doug DeGroot *(IBM Yorktown)*
Seif Haridi *(IBM Yorktown)*
Bob Keller *(Utah)*
Gary Lindstrom *(Utah)*
Jack Minker *(Maryland)*
Fernando Pereira *(SRI)*
Alan Robinson *(Syracuse)*
Sten-Ake Tarnlund *(IBM Yorktown)*
David S. Warren *(Stony Brook)*
Jim Weiner *(New Hampshire)*

Commercial Exhibits Chairman

Yuriy Tarnawski *(IBM Stamford)*

Referee List

Harvey Abramson
Daniel Bobrow
Peter Borgwardt
Ken Bowen
Robert A. Boyer
Ron Brachman
Maurice Bruynooghe
Mats Carlsson
Takashi Chikayama
Jacques Cohen
Shimon Cohen
John Conery
Philip T. Cox
Veronica Dahl
Doug DeGroot
Al Despain
Sally Douglas
Michael Dyer
Stephen Fickas
Herve Gallaire
Jean Gallier
Susan Gerhart
Joseph Goguen
Ralph Griswold
Seif Haridi
Lisa Hellerstein
Tim Hickey
Bharadwaj Jayaraman
Ken Kahn
James Kajiya
Tadashi Kanamori
Simon Kasif
Robert Keller
Dennis Kibler
Jan Komorowski
William Kornfeld
Robert Kowalski
Tony Kusalik
Guy Lapalme
Rosanna Lee
Wm Leler
Hector Levesque
Georgio Levi
Gary Lindstrom
David Maier
Umberto Martelli
Jack Minker
Naftaly Minsky
Prateek Mishra
Dan Moldovan
Lee Naish
Sanjai Narain
Mike Newton
Kamran Parsaye
R. Pattel
Fernando Pereira
Barak Perlmutter
Uday Reddy
Naphtali Rishe
Hajime Sawamura
Andreas Schuelke
Ehud Shapiro
Aditya Srivastava
Amitabh Srivastava
Mark Stickel
Stan Szpakowicz
Chun Tam
Sten-Ake Tarnlund
Evan Tick
Peter van Emde Bóas
H. van Emde Boas-Lubsen
Maarten van Emden
Mitchell Wand
David H. D. Warren
David S. Warren
Jim Weiner
Walter Wilson
Michael Wise

Table of Contents

Semantics

Chairman: Jan Komorowski, Harvard University

Implementation Issues

Chairman: William Kornfield, Quintus Computer Systems, Inc.

Theory

Chairman: Maarten H. Van Emden, University of Waterloo

Special Topics

Chairman: Susan Gerhart, Wang Institute of Graduate Studies

Keynote Address

Robert Kowalski
Imperial College, University of London
London, England

DIRECTIONS FOR LOGIC PROGRAMMING

Robert Kowalski

Department of Computing,
Imperial College. University of London

To evaluate possible future directions for logic programming, we need first to determine what logic programming is, and we need to determine its relationship both with PROLOG and with logic.

As a first (and most conservative) approximation we can identify logic programming with the observation that

procedures = Horn clauses + backward reasoning

elsewhere expressed more generally as

algorithm = logic + control [1,2].

Like a relation in a relational database or in an ideal logic program, this relationship can be used in different ways. Two uses are of special significance:

1. It can be used to support **declarative** programming. **Given** sentences formulated by a user in Horn clause logic, the **application of backward reasoning** by the computer gives rise to procedures, which render such sentences **executable.**

2. It can also be used to support **procedural** programming. **Given** procedures formulated by a programmer in problem-reduction terms, the **removal of backward reasoning,** gives rise to sentences of logic. In this way, the syntax of logic serves as an instrument to control the behaviour of the machine.

The **declarative use** is associated with such applications of logic as program specification and database query, where knowledge and problems are formulated without concern for the problem-solving process. The computer uses backward reasoning to convert such knowledge into procedures. But the form of reasoning used by the computer is generally of no interest to the user, whose main concern is that the computer uses its knowledge correctly and efficiently to solve the problems it is posed.

The **procedural use** is associated with programmer-controlled problem-solving. The programmer conceives of procedures and then uses the conclusion-conditions form of Horn clause logic to capture the problem-subproblems structure of those procedures. The "logical form" is more abstract than the procedures from which they are obtained.

Although both uses of logic programming are necessary, the declarative use is primary. Even when the procedural use is required for the sake of efficiency, the declarative use is necessary to establish the criteria of correctness for the procedures.

Although the two modes of use are conceptually quite distinct, they need to interact. Good software engineering principles dictate that declarative systems analysis and problem specification precede procedural implementation. (Extensions of) Horn clause logic can be used for both the declarative and the procedural stages of software development. Moreover, procedures expressed in logical form can also be read declaratively. This gives the programmer a "double check" on his intuitions. It also facilitates the verification of procedures by showing that the logical form of the procedures is a consequence of the program specification [3,4,5,6,7,8,9,10].

Relationship with functional programming

Horn clause programming shares many characteristics with functional programming. Both can be regarded as special cases of logic programming understood in the broad sense of

using logic to represent programs and
using deduction to perform computations.

Horn clause programming uses a restricted subset of predicate logic, whereas functional programming uses an equational form of logic, in which programs are represented by sets of equations and computation is performed by reasoning with equations used as rewrite rules. Both Horn clause programming and functional programming share the possibility of declarative and procedural modes of use. The gap between the two modes however, is greater for Horn clause programming. This is both a potential strength, as well as a source of potential weakness. It is a strength because Horn clauses are more expressive than functional programs and therefore have greater potential for declarative programming. It is a source of weakness because users have more trouble making the transition from declarative to procedural use. (We shall return to this below.)

Certain research areas in functional programming have been explored more thoroughly than

their analogues in Horn clause programming. Chief among these are higher-order functions, data types and parallel computer architectures. The attempt to synthesize systems which combine functional programming and Horn clause programming is probably the most active area of research in logic programming today. This work certainly needs to be furthered, but there is a danger that it may distract attention from the need to improve and extend Horn clause programming in other ways.

Many of these other improvements are suggested by a comparison of Horn clause logic with more powerful forms of logic in general. Others (to do with destructive assignment in particular) are suggested by a comparison with conventional procedural programming languages. To a certain extent, both types of improvement are also suggested by a comparison with relational databases.

In the remainder of this paper I shall use the term "logic programming" rather loosely to refer to extensions of Horn clause programming which stay within the broader notion of logic programming discussed above.

Relationship with relational databases

Logic programming (in this loose sense) can be regarded as combining characteristics of functional programming and relational databases. It shares with relational databases, the use of relations to represent information and the use of deduction to derive conclusions from assumptions.

Data in relational databases is defined by simple Horn clauses without conditions. However, queries can be expressed in full, unrestricted first-order logic. Thus data is more restricted than Horn clause programs, but queries are more powerful.

The more powerful form of queries in relational databases can be extended to logic programming by extending Horn clauses with negation as failure [11]. To the extent that negation as failure approximates full negation, this allows full unrestricted logic in the conditions of rules in general [12].

The extension of the simple Horn clause model of logic programming to include negation as failure applies to both the declarative and procedural modes of use. For example the rule

x is ordered if for all i and j
(u ≤ v if x_i is u and x_j is v)

which can be re-expressed in terms of negation as failure

x is ordered if for all i and j
not[x_i is u and x_j is v and not u ≤ v]

has an obvious **declarative** reading. But it can also be used to represent the **procedure** which shows x is ordered by searching through all pairs of positions i and j, succeeding if it cannot find any pair whose contents are out of order.

This extension of Horn clause logic as well as other features of database query languages, such as set abstraction, are essential features of logic programming today. Approximations to these features can and have been implemented in PROLOG [13,14].

Relational database query languages are purely declarative - the user specifies the form of the data to be found without identifying the method of access. Efficient access is the responsibility of query evaluators and optimizers.

The development of more efficient and more intelligent, system-controlled problem-solving strategies, extending those which have been developed for relational database systems, is an important area of research. This area has been neglected by comparison with the work relating Horn clause programming and functional programming.

Perhaps the most important problem-solving improvement which is needed is better subproblem selection. PROLOG's strategy of solving subproblems in the order in which they are written is appropriate for procedural uses of logic programming, but it is inadequate to support declarative modes of use. Other improvements which have already been investigated include selective backtracking and loop detection. These strategies may be essential for declarative uses which extend relational databases. However, they may also be too sophisticated for most programmers to control.

Logic programming shares with functional programming and relational databases the lack of a destructive assignment statement. However, destructive assignment is a feature of both conventional programming languages and database updates. Whether the efficiencies possible with programmer-controlled use of destructive assignment can equally be obtained by sophisticated implementations of assignment-free programs is perhaps the single. most important problem of logic programming. We shall return to the destructive assignment problem later.

PROLOG. For many people, logic programming has become synonymous with PROLOG. However, PROLOG is a concrete language with a specific problem-solving strategy and a number of extralogical features. Logic programming is uncommitted with respect to many aspects of problem-solving strategy and is not comfortable with the use of extralogical features.

PROLOG's problem-solving strategy is well-known: subproblems are tackled in the order in which they are written; and clauses are tried in the order in which they occur. PROLOG's most important extralogical primitives are

(a) "/" which controls backtracking,
(b) "add" and "delete" which dynamically add and delete clauses, and
(c) "evaluable predicates" for input-output.

PROLOG contains a "pure PROLOG" sublanguage consisting of Horn clauses without extralogical features. Consequently pure PROLOG admits both declarative and procedural modes of use. However, for the most part, its depth-first, sequential problem-solving strategy makes it better suited for procedural programming than for declarative knowledge representation and problem-solving. Moreover there is a discontinuity between learning to use PROLOG declaratively and learning to use it procedurally.

As a declarative language, pure PROLOG is simpler than most procedural languages because the user can ignore issues of program execution and can concentrate instead on the problem definition. However, as a procedural language, it is both simpler and more complicated than conventional programming languages. On the one hand, because it (pure PROLOG) lacks the assignment statement and other side effects, interaction between clauses is more transparent and therefore incremental development of programs is easier. On the other hand, because execution involves unification and backtracking, it is more complicated for the user to understand and more difficult to control.

Because of PROLOG's simple, relatively unsophisticated problem-solving strategy, declarative use can easily give rise to loops and fail to give an answer. To avoid the loop, the user has to abandon the declarative mode of use and learn to use PROLOG procedurally. This creates a discontinuity in the learning process. Many users never graduate from declarative to procedural modes of use.

The same discontinuity exists to a lesser extent with functional programming languages such as pure LISP. Because functional programs are determinate, order of equations doesn't matter, whereas order of clauses in PROLOG is critical; and because functions are not executed "backwards", order of evaluation of subterms is less important than order of subgoals in PROLOG.

The discontinuity problem will become more acute with logic programming systems which employ more sophisticated problem-solving strategies. Programming languages such as IC-PROLOG [14,15,16,], PARLOG [17,18,28], Concurrent PROLOG [19,20,21,] and GHC [29]. for example, seem to anticipate and address this problem by expecting its users to be sophisticated programmers. The problem will become even more severe with the advent of highly parallel non-von Neumann implementations of logic programming.

Extralogical features of PROLOG

In addition to the discontinuity problem, which arises when using PROLOG as a pure logic programming language, there are problems which arise with the use of extralogical features. The use of extralogical features can be classified into three categories:

1) uses which are inherited from previous exposure to conventional procedural languages or which in some other way fail to exploit what is possible with pure (declarative and/or procedural) logic programming style;

2) uses which extend the logic of Horn clause programming, either by extending expressive power or by extending problem-solving strategies;

3) uses which do not seem to fall into either of the first two categories. This applies especially to the addition and deletion of clauses to obtain the effect of destructive assignment.

The ideal is to do away with extralogical features completely (or at least to isolate and identify those which cannot be eliminated). Those uses, in category 2. whose purpose is to implement extensions of Horn clause programming should be replaced by providing such extensions as primitives in an improved PROLOG environment [22]. This has already happened with the provision of primitives for negation as failure and set abstraction. However, the form of negation as failure which is generally provided is still incorrect [23]. Its correction should be considered a priority.

Replacing extralogical features by extensions of Horn clause programming would also help to eliminate those uses, in category 1, which are the result of poor programming style.

The problem is with uses of extralogical features in category 3, which cannot readily be replaced by clearly identifiable extensions of Horn clause programming. In some cases the problem is that there seems to be more than one candidate extension which solves the problem. The elimination of extralogical primitives for input-output is an example. One possibility is to incorporate all input into input streams and all output into output streams. The input can be read when needed and the output can be written when available. Such behaviour can be obtained by producer-consumer execution [15,16,28].

Another possibility is to arrange all input to take the form of answers supplied by the user to queries posed by the system. The system poses such queries when it tries to tackle subproblems which it does not have the knowledge to solve [24]. Other than the output which is implicit in the system's questions to the user, all output takes the form of bindings to variables in the top-level goal.

The first solution requires the programmer to control input-output explicitly using sophisticated consumer-producer execution. The second solution is better suited to declarative uses of logic programming. Moreover, in theory at least, it allows input to be supplied by several users in parallel. The problem is how to reconcile these two solutions.

Destructive assignment

Most discussion of PROLOG's extralogical features concentrates on the use of cut "/" . Some of these uses can be replaced by negation as failure. Others can be avoided by conditionals or case expressions. PROLOG'S primitives for adding and deleting clauses are more troublesome.

However, many instances of adding and deleting clauses are logically innocuous. Addition of clauses, for example, is often used simply to improve efficiency by adding lemmas to store intermediate results for later reuse. The logical character of such additions can be made more apparent by providing lemma generation as an explicit facility.

Similarly, deletion of clauses can be used in a logically justifiable manner to reclaim storage occupied by clauses which are no longer accessible to the rest of the program. The logical character of such deletions would be more secure if they were replaced by garbage collections automatially performed by the system.

It is the combined use of addition and deletion to transform objects without changing their names, as in the destructive assignment operation of conventional programming languages, which is logically most troublesome.

Novice PROLOG programmers often use deletion of clauses to simulate destructive assignment, even when this is unnecessary. Often the same effect can be obtained more efficiently by writing assignment-free tail-recursive procedures which call themselves with new values as parameters. The following program which defines multiplication as repeated addition illustrates the point:

```
Times(x y z) if TimesAux (x y x z)
TimesAux(x 1 z z)
TimesAux(x y z' z) if  Plus(y' 1 y)
                   and Plus(x z' z")
                   and TimesAux(x y' z" z)
```

Here, exploiting the use of destructive assignment in the implementation of tail recursion, successive calls to TimeAux

> destructively decrement the second parameter by 1, and destructively add x to the third parameter, until the second parameter equals 1, in which case the result is the value held in the third parameter.

It might be preferable if similar efficiency could be obtained by more intelligent execution of the "obvious" recursive definition:

```
Times(x 1 x)
Times(x y z) if  Plus(y' 1 y)
             and Times(x y' z')
             and Plus(x z' z)
```

In fact this might be possible with some appropriately goal-oriented form of forward reasoning.

Procedural interpretations of forward reasoning seem to be worth exploring more generally for several reasons. Many rule-based expert systems languages, for example, work by forward reasoning. Moreover, the implementation of forward reasoning in the connection graph proof procedure [25,26,] provides further opportunities for deletion of clauses and therefore for system-introduced destructive assignment.

The problem of destructive assignment becomes more acute with the manipulation of more elaborate data structures. This has been studied extensively for recursive data structures. In this case, the effect of destructive assignment can often be obtained by some form of incremental garbage collection. Thus, for example. it is possible to devise implementation schemes which use destructive assignment to construct the list x by overwriting A in the call

```
Append(A B x) ?
```

if there are no other references to A elsewhere.

The use of conventional destructive assignment to overwrite A in the example above not only conserves space, but it also conserves names by reusing A to name the result of appending B to A. This is logically dangerous. The same name is used to refer to two quite different lists.

The problem of destructive assignment becomes most severe when we consider the use of databases as data structures. In such cases, straightforward use of logic without destructive assignment gives rise to the frame problem [25].

Databases as data structures. Many uses of addition and deletion of clauses in PROLOG can be interpreted as manipulating databases as data structures. PROLOG programs which use addition and deletion to update databases are a special case.

Databases defined by sets of clauses can perform many of the functions of conventional data structures such as arrays. An array A, for example, which contains the first 100 even positive integers:

$A_i = 2i$ for $1 \leq i \leq 100$

can be defined either by clauses which enumerate its elements:

```
Item(A 1 2)
Item(A 2 4)
   .
   .
   .
Item(A 100 200)
```

or by a general rule which computes them:

```
Item(A i x) if  1 < i
            and i ≤ 100
            and Times(2 i x).
```

It is also useful to have a predicate which

determines the length of arrays. This can be defined by an assertion

```
Length(A 100)
```

or by a general rule

```
Length(x i) if  Item(x i u)
            and Plus(i 1 j)
            and not Item(x j v).
```

Like arrays and relational databases, data defined by clauses has several advantages over recursive data structures. Perhaps chief among these is that clauses provide direct access to the data in contrast to the recursively programmed access obtained with recursive data structures. Moreover, such conceptually direct access can be implemented by the system equally well by means of sequential or parallel search.

The problem with such data structures arises when they are transformed into other data structures. Consider, for example, the problem of appending two sequences A and B defined by means of a three argument predicate "Item" as above:

```
Append(A B z) ?
```

The goal can be solved by a single clause! Namely:

```
Append(x y append(x y))
```

However, we need other clauses to determine the elements of the resulting sequence append(A B). The following clauses do this:

```
Item(append(x y) i u) if  Item(x i u)          (App1)
Item(append(x y) j u) if  Length(x k)          (App2)
                      and Plus(i k j)
                      and Item(y i u)

Length(append(x y) m) if  Length(x k)          (App3)
                      and Length(y l)
                      and Plus(k l m)
```

In this case, if there are no other calls of the form

```
Item(A i u) ?
```

the effect of destructive assignment to A can be obtained in part by forward reasoning, resolving the clauses which define A with the condition of App1 above. The clauses which define A can then be deleted and replaced by the new clauses which together with App2 and the definition of B define append(A B). Certain proof procedures, such as the connection graph proof procedure, perform this deletion automatically. Compared with conventional destructive assignment, however, there is still a major inefficiency involved in using App1 over and over again to derive the clauses in the definition of append(A B) which replace the clauses in the definition of A. This inefficiency is equivalent to the **frame problem**, which is illustrated again in the following example.

Consider, the problem of interchanging the second and third items in the sequence A.

```
Interchange(A 2 3 z) ?
```

This too can be solved by a single clause:

```
Interchange(x i j inter(x i j))
```

Here the clauses

```
Item(inter(x i j) i u) if  Item(x j u)      (Int1)
Item(inter(x i j) j u) if  Item(x i u)      (Int2)
Item(inter(x i j) k u) if  Item(x k u)      (Int3)
                       and k ≠ i
                       and k ≠ j

Length(inter(x i j) k) if Length(x k)       (Int4)
```

determine the elements of the resulting sequence inter(A 2 3).

Clause Int3 is a **frame axiom**, which expresses that most positions k have the same value in inter(x i j) as they have in x. The **frame problem** is the inefficiency which arises from reasoning with the frame axiom. The inefficiency is independent of whether the axiom is used backwards or forwards, with or without automatic deletion.

Conclusion

I have concentrated in this paper on three problems:

(1) the discontinuity between declarative and procedural uses of logic programming,

(2) the elimination of extralogical features from PROLOG, and

(3) destructive assignment.

The problem of destructive assignment is possibly the single most important problem of logic programming. Elsewhere I and my colleagues have sketched possible solutions to the problem, including the related frame problem. These solutions include the use of reflection principles in metalogic programming [23] and the use of time periods as parameters [27] of time-varying relationships. These and other possible solutions need to be studied in greater detail.

The problem of destructive assignment is highlighted by an analysis of the use of extralogical features in PROLOG. Such an analysis also motivates extensions of the simple notion of logic programming as Horn clauses with backward reasoning. Extensions are needed both of expressive power and problem-solving strategies.

Just as important, however, are the methodological problems which arise with both PROLOG and "pure" logic programming. The main problem here is the discontinuity between declarative and procedural modes of use. Some of these problems might be alleviated by program transformations.

Much of the past success of logic programming is based upon the success of PROLOG. Careful study of the problems of PROLOG is one of our best guides, therefore, both to the more general problems of logic programming and to the future direcxtions logic programming might take.

Acknowledgements

Research in Logic Programming at Imperial College has been supported by the Science and Engineering Research Council and International Computers Limited. I am indebted to Steve Gregory, Chris Hogger, Frank Kriwaczek and Fariba Sadri for useful comments on an earlier draft of this paper.

REFERENCES

[1] Hayes P.J. (1973), Computation and Deduction. Proc. 2nd MFCS Symp. Czechoslovak Academy of Sciences, pp 559-565.

[2] Kowalski R.A. (1979) Algorithm = Logic + Control. CACM, August 1979. Vol. 22 pp.424-436.

[3] Clark K.L., and Sickel S.,(Aug.1977) Predicate Logic: A Calculus for Deriving Programs, Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, pp. 419-420.

[4] Clark K.L., Darlington J., Algorithm Classification Through Synthesis. The Computer Journal 1980. Vol. 23, No. 1, pp. 61-65.

[5] Clark K.L., (1981) The Synthesis and Verification of Logic Programs. Technical Report 81-36, Imperial College, London, 1981.

[6] Clark K.L., McKeeman W.M., Sickel S., (1982), Logic Program Specification of Numerical Integration. In Logic Programming (eds. Clark K.L. and Tarnlund S-A.), Academic Press, pp. 123-140.

[7] Hogger C.J., (1978) Goal-Oriented Derivation of Logic Programs, Proceedings of Mathematical Foundations of Computer Science, Lecture Notes in Computer Science No. 64.

[8] Hogger C.J., (April 1981) Derivation of Logic Programs, Journal of the ACM Vol.28,pp. 372-392.

[9] Hogger C.J., (1984) Introduction to Logic Programming. Academic Press.

[10] Kowalski R.A., (1984) The Relationship Between Logic Programming and Logic Specification. Proc. Royal Society Discussion Meeting, February 1984. In Mathematical Logic and Programming Languages (eds. C.A.R. Hoare and J.C. Shepherdson) Prentice Hall International.

[11] Clark K.L., (1978) Negation as failure. In Logic and Databases. (eds. Gallaire H. and Minker J.), Plenum Press, New York, pp 293-322.

[12] Lloyd J.W., Topor R.W., (1984) Making Prolog More Expressive. Journal of Logic Programming Vol 1. No. 3, pp 225-240.

[13] Warren D.H.D., (1982) Higher-Order Extensions to Prolog: Are They Needed? Machine Intelligence, 1982.

[4] Clark K.L., McCabe F.G., (1980) IC-PROLOG Aspects of its Implementation. In Proceedings of Logic Programming workshop, Debrecen, Hungary.

[15] Clark K.L., McCabe F.G., (1979) The Control Facilities of IC-PROLOG. In Expert Systems in the Micro-Electronic Age (ed. Michie D), Edinburgh University Press, pp. 153-167.

[16] Clark K.L., McCabe F.G., Gregory S., (1982) IC-PROLOG Language Features. In Logic Programming, Clark and Tarnlund, Academic Press, 1982, pp. 253-266.

[17] Clark K.L., Gregory S., (Oct.1981) A relational language for parallel programming. Proceedings of ACM Conference on Functional Programming Languages and Computer Architecture, New York, pp. 171-178.

[18] Clark K.L., Gregory S., (Nov.1984) Notes on Systems Programming in Parlog. Proceedings of International Conference on 5th Generation Computer Systems, Tokyo, Nov. 1984. North-Holland.

[19] Shapiro E.Y., (Jan.1984) Systems Programming in Concurrent Prolog, POPL. Salt Lake City, Utah.

[20] Shapiro E.Y., Takeuchi A.,(1983) Object Oriented Programming in Concurrent Prolog, New Generation Computing 1, Springer Verlag.

[21] Shapiro E.Y. (Jan.1983) A Subset of Concurrent Prolog and Its Interpreter, Technical Report TR-003, ICOT - Institute for New Generation Computer Technology, Tokyo, Japan.

[22] Kowalski R.A., (1981) Prolog as a Logic Programming Lanuage. Proceedings of AICA Congress, Pavia, Italy.

[23] Kowalski R.A., (1983) Logic Programming, Proceedings of the IFIP-83 Congress, North-Holland, Amsterdam, pp. 133-145.

[24] Sergot M.J., (1983) A Query-The-User Facility for Logic Programs, in Integrated Interactive Computer Systems, Degano P., and Sandwell E., (eds.), North-Holland, 1983. Also published in New Horizons in Educational Computing, Yazdani M. (ed.), Ellis Horwood, 1984, pp. 145-163.

[25] Kowalski R.A., (1979) Logic for Problem Solving. North-Holland/Elsevier, New York.

[26] Kowalski R.A., (1974) A Proof Procedure Using Connection Graphs. In Journal of ACM Vol 22, pp. 572-595.

[27] Kowalski R.A., Sergot M.J., (March 1985) A Logic Based Calculus of Events. Department of Computing, Imperial College, London.

[28] Clark K.L., Gregory S., (1985) PARLOG: parallel programming in logic. To appear in ACM Trans. on Programming Languages and Systems.

[29] Ueda K., (1985) Guarded Horn Clauses. Technical Report TR-103, ICOT.

Parallelism

Chairman
John Conery
University of Oregon