Non-Procedural Logic Programming

Seppo Keronen

Wilhelm-Schickard-Institut, Universität Tübingen, Sand 13
D-72076 Tübingen, Germany
seppo@logik.informatik.uni-tuebingen.de

Abstract. We present a logic programming language where both problem domain and computational knowledge are expressed in logic. A logic program in this language consists of an object-program and a number of meta-programs. The object program, a collection of formulae, is a description of the problem domain of interest. The meta-programs, also just collections of formulae, specify desired computational behaviours. The object-program and meta-programs are compiled together to produce a single, efficient procedural logic (Prolog) program.

1 Motivation

In 1979 Robert Kowalski [10] used the slogan

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

to express the idea that a computer program should consist of two quite separate components:

- **Problem Description:** A description of the problem domain is supplied as a collection of predicate logic assertions. The problem to be solved is represented by a query, again a predicate logic formula. A solution to the problem is computed by a process of inference. The process aims to construct a proof for the query from a subset of the given assertions.
- **Search Strategy:** An uninformed inference process cannot find solutions for any but the most trivial problems. That is, finding a proof (or demonstrating failure) for a given problem description is itself a difficult problem. The knowledge about how the search for a proof is to be carried out needs also to be supplied by the programmer.

The current generation of logic programming languages, such as Prolog, are based on the procedural reading of the problem domain assertions. This simple, elegant idea has made the practical implementation of logic programming languages possible. Technology developed for the interpretation and compilation of procedural languages has been applied, resulting in very efficient implementations. The desired proof search strategy is expressed in terms of procedural constructs embedded in the problem domain assertions. Such a control language is easily and efficiently implementable. A control construct appears as just another procedure call in the sequential execution flow of the program.

We now want to expand and refine Kowalski's schema. Firstly, a proof search strategy is not the only kind of computational knowledge that needs to be supplied by the programmer. Again taking Prolog as a point of reference:

- **Input/Output:** Communication is an essential part of any program. A large number of constructs, 'read', 'write', 'consult' for example, are embedded in the program clauses to specify this component.
- **Resource Use:** Limited resources, such as time, space and processors need to be allocated wisely. A 'time-out' construct as well as some form of exception mechanism is usually provided.
- **Learning:** Revision of the problem domain theory, as more is learned from external agents or simply to avoid expensive recomputation, is currently specified using 'assert', 'retract' and so on.
- **Computational Models:** Computer hardware and low level software can compute answers faster than deduction from axioms. Commonly arithmetic functions and an external language interface are made available.
- **Optimizations:** Implementation options and parameters, for instance choosing between interpretation, compilation or partial evaluation, need to be specified.

Secondly, the specification of a proof search strategy may be factored into a number of separate concerns. Prolog distinguishes two components:

- **Order of Search:** Textual order of clauses and subgoals is traditionally used to specify the order in which inferences are to be made. More recently, co-routining constructs, such as 'wait' and 'block' provide a limited escape mechanism from the procedural execution model.
- **Search Space Pruning:** A search space often contains dead-ends, infinite branches and duplicate solutions. Constructs such as '!' (cut) and 'once' can be used to prune some of these unwanted portions of the search space.

We argue that the procedural specification of computational knowledge, as described above, falls far short of the ideal. The following symptoms are familiar to any Prolog programmer:

- **Readability of Programs:** Unlike the problem domain language, the computation domain language is not declarative. The rearrangement and embedded constructs even destroy the readability of the problem domain assertions. These factors lead to programs that are difficult to understand and verify.
- **Reusability of Programs:** In principle a given problem domain description can be used in many different ways. For example, the specification of an electronic circuit could be used in design, fabrication, fault diagnosis, teaching etc.. In practice the embedded computational language inhibits such reuse.

- **Expressive Power of Problem Domain Language:** Only a small sublanguage of predicate calculus is available. When extending the expressive power of the language, it can be difficult to extend the procedural semantics to cover the new constructs.
- **Expressive Power of Computation Domain Language:** The procedural language lacks the expressive power for many necessary control constructs. Partly to escape this problem many programs are written as meta-interpreters, in effect forcing the programmer to invent a new language for expressing the problem. The result is that program development effort is higher and programs are less efficient than need be.
- **Parallel Execution:** A sequential procedural language is not well suited for parallel execution.

Motivated by the above observations we are developing a new logic programming langauge based on the following refinement of Kowalski's schema:

$$\begin{aligned}\text{Algorithm} = {} & \text{ProblemDomainTheory} + \\ & \text{ComputationDomainTheory}\end{aligned}$$

In other words we demand that computational knowledge as well as problem domain knowledge be expressed declaratively as a logical theory. Further, we argue that computational behaviour should be specified as a set of separate subtheories.

$$\begin{aligned}\text{ComputationDomainTheory} = {} & \text{SearchOrderTheory} + \\ & \text{SearchPruningTheory} + \\ & \text{Input/OutputTheory} + \\ & \text{ResourceUseTheory} + \\ & \text{LearningTheory} + \\ & \text{ConnectiontoComputationalModelsTheory} + \\ & \text{OptimizationsTheory} + \\ & \ldots\end{aligned}$$

This approach provides decisive advantages over a procedural, non-separable language:

- **Readability of Programs:** Both domain knowledge and control knowledge are expressed declaratively. As a first approximation, the various theories can be read and understood separately. It is only when a computational theory refers to the particulars of the domain theory, that the domain theory needs to be understood first.
- **Reusability of Programs:** The one domain theory may be paired with different computational theories implementing different uses of the domain knowledge. Computational subtheories can be plugged in for purposes such as different search strategies, debugging, execution on different machines etc.

- **Expressive Power of Problem Domain Language:** It is no longer necessary to assign a procedural reading for each problem domain language construct. This relaxation makes the extension of the problem domain language easier.
- **Expressive Power of Computation Domain Language:** Potentially the full power of the predicate calculus may be available. The expressiveness of this language, however, can have a serious impact on execution speed. This point is taken up later.
- **Parallel Execution:** The language is no longer tied to a procedural execution model. Only the relevant computational subtheories need be rewritten for parallel execution. This paper, however, focuses on the implementation of a sequential language only.

In order to obtain these gains we do not need to sacrifice the advantages, most notably execution speed, of procedural logic programming. The implementation put forward in this paper compiles the problem domain and computation theories together into a single efficient procedural logic (Prolog) program.

Our motivations and methods are related to existing work in the area of multi-level inference systems and meta-interpreters. The next section examines these relationships. Section 3 provides a brief introduction to the language with examples. Section 4 outlines a semantics for the language based on multiple, communicating reasoners. Section 5 describes briefly the compilation of the language into Prolog. We conclude with some examples and comments on future directions.

2 Multi-level Inference Systems

Let us assume that predicate logic is an appropriate language for representing and reasoning about complex problem domains. Let us also recognise that the control of reasoning is a complex task which in itself requires reasoning. It follows that we should employ predicate logic to represent and reason about control. This simple argument leads us to consider systems where reasoning takes place at two or more levels, namely *object-level* reasoning about the original problem domain, *meta-level* reasoning about object-level inference, *meta-meta-level* reasoning about meta-level inference and so on. Such *multi-level inference systems* have been studied by a number of investigators during the last 20 years, including [2], [5], [6], [12] in the logic programming community, as well as [4], [8], [15], [16] and others working on automated theorem proving, proof editors and expert systems. A survey of this work, up to 1989, can be found in [15].

A subclass of multi-level inference systems, called *meta-interpreters* has seen vigorous development within the main-stream of logic programming, see for example [1]. A meta-interpreter typically implements just two levels of reasoning. The two levels are collapsed into a single meta-level by having a specification of the object-level simulated at the meta-level. At first sight a meta-interpreter would seem to be an ideal vehicle for representing computational knowledge.

After all isn't the object level computation fully and declaratively specified by its meta-interpreter? On closer examination, however, this approach suffers from serious problems:

- **Separation:** The meta-interpreter is called on to carry knowledge about a number of distinct computation domains, such as control, input/output etc. as listed earlier. So again we have the problems of inadequate separation. How do we plug in different proof search strategies, for example, without affecting the other functions?

  In contrast we advocate separate theories for search order, search space pruning, input/output, learning etc.. The separate theories may be independently developed, maintained and reused.
- **Control of Meta-Level:** How is the computational knowledge for the meta-interpreter to be specified? The usual solution seems to be to fall back on a procedural semantics for this language. This means that the computational specification is obscured by control annotations. The parallel execution of the meta-interpreter is also not possible.

  In contrast each of our meta-level theories is small (in any case finite), and so we can assume that the deductive closure of these finite theories is always computed. For the parallel execution of the problem domain computation, the allocation of processing units is just another meta-theory to be supplied.
- **Expressive Power:** The specification of the object-language is in no way constrained. A meta-interpreter can implement an arbitrary object-language either intentionally or due to a bug in the specification.

  In our approach knowledge about each of the domains is expressed uniformly as a collection of predicate logic assertions.
- **Efficiency:** Implementations based on meta-interpretation are inefficient. Partial evaluation techniques no longer appear as promising as they once did [15].

  Our implementation is based on compiling the object-level and meta-level together into a single efficient Prolog program.

Leon Sterling's group [13] [14] has pioneered the idea of a computational specification consisting of a number of separate meta-theories to be composed into a single efficient meta-interpreter. Our main departures from this work are that our meta-language is not Prolog and that our implementation is based on compilation instead of meta-interpretation.

3 The Language

A logic program for us consists of a number of separate (sub)programs, being a single *problem domain (sub)program* (or *object-program*) and multiple *computation domain (sub)programs* (or *meta-programs*). The problem domain subprogram is a collection of predicate logic formulae interpreted as a description of the problem domain of interest. The deductive closure of this subprogram is typically very large and its search space will often contain infinite branches, and multiple paths to solutions. As a small example of such a badly behaved program see figure 1. What is important here is the clarity and completeness of the description not computational behaviour.

```
edge(a,b).
edge(c,b).
edge(a,c).
edge(X,Y) :- edge(Y,X).

path(X,Y) :- edge(X,Y).
path(X,Z) :- path(X,Y), path(Y,Z).
```

Fig. 1. example object-program

In what follows we assume a simple proof theoretic view of such a subprogram. We regard the facts (atomic formulae) of the program as assumptions, and the clauses (implications) as rules of inference that may be applied to derive new atomic formulae, see [9] and [7]. For example the program of figure 1 entails the formula path(a,c), since we can construct the proof

$$\dfrac{\dfrac{\text{edge(a,b)}}{\text{path(a,b)}} \quad \dfrac{\dfrac{\text{edge(c,b)}}{\text{edge(b,c)}}}{\text{path(b,c)}}}{\text{path(a,c)}}$$

When our inference engine constructs such a proof it applies rules by backward chaining, starting from the desired conclusion as the root of the proof tree moving towards the facts that are the leaves of the proof tree.

Like the problem domain subprogram, each computation domain subprogram is also just a collection of predicate logic formulae. A computation subprogram is interpreted as specifying part of the behaviour of the object-level inference engine. For example the program shown in figure 2 is a search space pruning strategy that may be applied when searching for proofs given the problem domain program of figure 1. Program 2 is a loop detector consisting of a single implication: The antecedent tests for a goal formula that is more general (or equivalent) to one of its ancestor goals. The conclusion demands the failure of such a goal.

```
open(Formula1,Goal),
ancestor(Goal,Formula2),
subsumes(Formula1,Formula2)
-: fail(Goal).
```

Fig. 2. example meta-program

Notice that contrary to the object-level, meta-level rules are applied in the forward direction, starting from the facts of the current computation state and moving towards their consequences. The meta-level operator "`-:`" in contrast to the object-level "`:-`" is used to indicate this difference. Notice also that control programs, such as the program of figure 2, specify proof theoretic moves that may (and should) be formally justified. In this case:

$$\begin{array}{c} \dfrac{\Pi_2}{(G)} \\ \dfrac{\Pi_1}{G\Theta} \end{array} \qquad \longrightarrow \qquad \dfrac{\Pi_2\Theta}{G\Theta}$$

That is, any proof of a goal $G\Theta$ relying on the more general subgoal G may be pruned, since the simpler proof $\Pi_2\Theta$ for $G\Theta$ will always exist.

As a more complex example, the program shown in figure 3 specifies an iterative deepening search strategy. Like program 2, program 3 is domain independent, and may therefore be combined with any problem domain program. Also, program 2 and program 3 specify two quite independent strategies that may be combined separately or together with a given object-program.

```
open_query(_,_)                             % clause 1
-: write('iterative deepening parameter? ')
   -: read(Depth)
      -: remember(limit(Depth)),
         remember(parameter(Depth)).

open(_,Goal),                               % clause 2
limit(Depth),
depth(Goal,Depth)
-: fail(Goal),
   remember(limit_exceeded).

fail_query(QueryFormula,Query),             % clause 3
limit_exceeded,
parameter(Increment)
-: forget(limit_exceeded),
   forget(limit(Depth)),
   plus(Depth,Increment,NewDepth)
   -: remember(limit(NewDepth))
      -: open_query(QueryFormula).
```

Fig. 3. meta-program for iterative deepening search

Program 3 consists of three clauses:

- **Clause 1:** When a query is opened, ask the user to supply a value for the depth parameter, and then add two facts, recording the value of the parameter and search limit, to the meta-program.
- **Clause 2:** When a goal whose depth matches the search limit is opened, fail the goal and record the fact that the depth limit was exceeded.
- **Clause 3:** When the query fails due to the search limit having been exceeded, reopen the query with a deeper search limit.

This program illustrates a number of important points discussed in the remaining paragraphs of this section.

The language contains a fixed vocabulary of introspection and command predicates:

- **Inspection Predicates:** The current state of object-level proof search is available for inspection by the meta-level. The execution state of individual goals (open, succeeded, failed, blocked) and the location of goals in the proof tree relative to the root of the proof tree (the query) and other goals can be examined.
- **Command Predicates:** The conclusions inferred at the meta-level are interpreted as commands by the object-level inference engine. This vocabulary includes input/output and theory change commands, as well as commands that affect the state of the current proof tree.

The specification of the order in which computations are to be performed is often necessary. Recall that in the Prolog language all operators have a sequential operational semantics. Here just the meta-level implication operator '-:' is assigned a procedural meaning. The expression

$$A\ -:\ C$$

is read, *when A then C.* The nesting of implication is allowed in the conclusion. For example in clause 1 we specify that a write operation (writing a prompt) be executed succesfully before the parameter value (requested by the prompt) is read, and that the value is remembered only once it has been read. The implementation is free to evaluate all operators, other than the '-:', in any order or in parallel.

4 Introspection

Our approach is an evolutionary advance on, rather than a radical departure from, current logic programming practice. Consider, for instance, the construct

$$\texttt{when}(\mathit{Condition}, \mathit{Goal})$$

found in modern Prolog implementations [3]. This is best thought of as a piece of meta-language that instructs the implementation to suspend execution of *Goal* until the given *Condition* is satisfied. In other words, the computation state is available for *introspection* and constraints on computational actions derived from this knowledge are obeyed by the implementation. This is exactly the relationship between our meta-level and object-level subprograms.

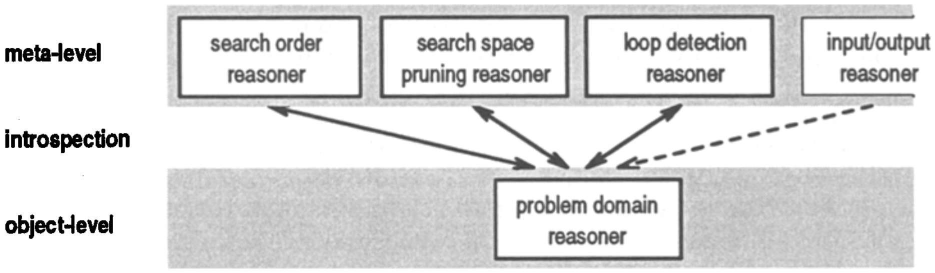

Fig. 4.

Consider a set of reasoners, one each for the subprograms making up the logic program. This arrangement is illustrated in figure 4. Each meta-level reasoner is coupled to the object-level reasoner by an introspection relationship [15]. Figure 5 focuses on the relationship between any one meta-level reasoner and the object-level reasoner. We distinguish the two directions of this relationship:

- The *upward reflection* of a part of the proof search state as a logical theory (collection of atomic formulae) accessible to inspection by a meta-level reasoner.
- The *downward reflection* of a theory (collection of atomic formulae) specifying inference engine actions as computational behaviour by the object-level reasoner.

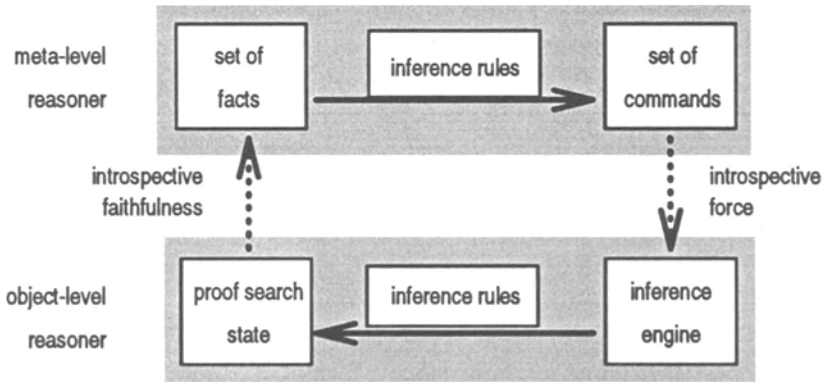

Fig. 5.

Each meta-level reasoner is associated with a theory generated by three knowledge sources:

- **Proof Search State:** An extensional (collection of atomic formulae) description of the current state of proof search.
- **Subprogram:** An intensional (requiring inference), partial specification of computational behaviour is supplied by the meta-level subprogram.
- **Default Behaviour:** In the case that no commands are derivable from meta-programs, the object-level actions may be formalized as a default theory.

For a given proof search state a meta-level reasoner has the task of computing the complete extension (collection of atomic formulae) of its associated theory. In other words, control of the inferences carried out by meta-level reasoners is not required. The most natural computational mechanism would of course be parallel forward chaining from the proof search state to the commands implied by that state.

The expressive adequacy of the meta-language depends crucially on how much of the computation state, and what computational actions, are made available for reasoning. One usually pays for increased expressive power by suffering reduced execution speed. We will study this tradeoff in the next section. The meta-level constructs used in current procedural languages provide a starting point for a compromise between expressive power and performance. From this baseline extensions, such as controlled loop checking for example, can be readily made.

5 Implementation

A direct implementation of the language, based on the communicating multiple reasoners semantics presented in the preceding section, would be extremely inefficient on current hardware. In this section we demonstrate that we can instead compile the meta-programs and object-program together into a single efficient procedural logic (Prolog) program.

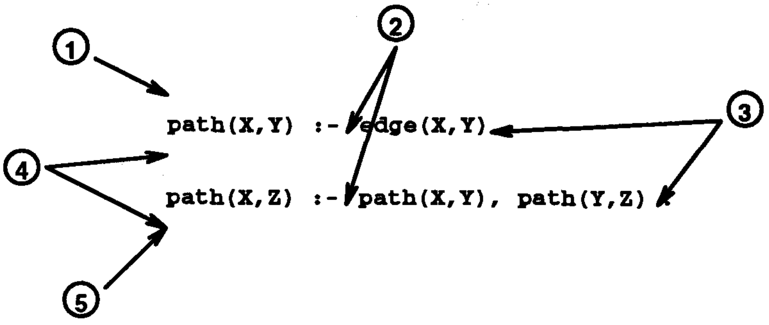

Fig. 6.

Let us adopt, as the default search strategy for the object level inference engine, the depth first, left to right search strategy of Prolog. This implies that in the absence of any meta-programs the object-program is correctly executed by Prolog. This leaves the problem of what to do with the meta-language statements. We take the view that these statements specify control constructs to be inserted into the object-program.

Consider the procedure *path/2* shown in figure 6. When executed by a Prolog inference engine, then at the points indicated in the figure, we know the following about the proof search state:

1. a *path/2* goal is opened
2. a clause is selected
3. a proof has been found
4. a clause is rejected
5. the goal has failed

We can engineer meta-language inspection predicates to target these points. For example, the predicate

```
open(path(A,B),Ref)
```

is such a *targeting predicate*. It becomes true as the flow of execution moves through point 1 in figure 6 as a *path/2* goal is opened. The parameter **Ref** here is a unique reference to a goal node in the proof tree, required as a parameter for other predicates in the same meta-language statement.

Once we have a target, the proof search state may be further inspected. For example we may write:

```
depth(Ref,Depth)
```

to find the depth of the referred to goal in the proof tree; Or

```
var(A)
```

to find out whether the first argument of the goal is a variable. The compiler inserts these tests into the object program at locations indicated by targeting predicates.

Any commands specified in meta-language statements are also inserted into the points indicated by the targeting predicates. For example the statement

```
open(path(A,B),Goal),
found(path(C,D),Goal),
ground(A), ground(B)
-: commit(Goal).
```

prunes the search space preventing multiple solutions for ground *path/2* goals. Together with the object program of figure 6 it compiles into the Prolog program shown in figure 7. Notice how code has been placed into multiple points and additional clauses have been generated by this single meta-language statement.

```
path(X,Y) :- ground(X), ground(Y),
             edge(X,Y),
             !.
path(X,Y) :- (nonground(X) -> true ; nonground(Y)),
             edge(X,Y).
path(X,Z) :- ground(X), ground(Z),
             path(X,Y), path(Y,Z),
             !.
path(X,Z) :- (nonground(X) -> true ; nonground(Z)),
             path(X,Y), path(Y,Z).
```

Fig. 7. compiler output

Our implementation scheme should now be clear: We have separated the amalgamated Prolog language into an object-language and a meta-language. The object program and meta-programs are pure logic programs. All the Prolog meta-predicates are available in the new separate meta-language. The task of the compiler is to re-amalgamate the object and meta-programs resulting in a single executable Prolog program.

The compilation process becomes a little more complicated than indicated above, as multiple targeting predicates and constructs such as goal suspension are included in the meta-language. These issues will be addressed in a future paper.

6 Conclusion

We have outlined a logic programming language where both problem domain and computational knowledge are expressed in logic. A logic program consists of a number of separate theories, a single problem domain theory and a number of interchangeable computation domain theories. The object-program (problem domain description) and meta-programs (computational behaviours) can be compiled together to produce a single, efficient procedural logic (Prolog) program.

A prototype compiler for the language described in this paper has been developed using SICStus Prolog as both the implementation and target language. Experience with this implementation indicates that the main practical advantages of our approach are the following:

- The problem domain description can be more easily developed, verified and understood, as it is not loaded up with procedural constructs.
- Computational behaviours are easier to specify, verify and understand, as a collection of rules rather than procedural constructs hidden in the problem domain description.
- A problem domain description may be combined with various search strategies, input/output behaviours etc. without modifying the description.

– Any efficiency problems can be diagnosed by reading the Prolog program generated by the compiler.

The main limitation is that we are confined to a proof search mechanism that represents just a single object proof at a time. In this framework we cannot express control strategies, such as breadth first search for example, that explore multiple partial proofs concurrently. Also, only a very restricted meta-language is accepted by our current implementation. Future work will focus on better compilation techniques and on increasing the expressive power of the meta-language.

References

1. Harvey Abramson and M.H. Rogers (eds). *Meta-Programming in Logic Programming.* MIT Press, 1989.
2. Kenneth A. Bowen and Robert A. Kowalski. 'Amalgamating Language and Metalanguage in Logic Programming'. in K. L. Clark and S-Å. Tärnlund (eds). *Logic Programming.* Academic Press, 1982.
3. Mats Carlsson et al. *SICStus Prolog User's Manual (version 2.1).* Swedish Institute of Computer Science, 1991.
4. Randall Davis. 'Applications of Meta Level Knowledge in the Construction, Maintenance and Use of Large Knowledge Bases'. in R. Davis and D. Lenat. *Knowledge-Based Systems in Artificial Intelligence.* McGraw-Hill, 1980.
5. K. Eshghi. *Meta-Language in Logic Programming.* PhD thesis, Department of Computing, Imperial College, 1986.
6. Herve Gallaire and Claudine Lasserre. 'Metalevel Control for Logic Programs'. in K. L. Clark and S-Å. Tärnlund (eds). *Logic Programming.* Academic Press, 1982.
7. Lars Hallnäs and Peter Schroeder-Heister. *A Proof Theoretic Approach to Logic Programming. I, Clauses as Rules.* Journal of Logic and Computation 1(2) 261–283, 1991.
8. Patrick J. Hayes. *Computation and Deduction.* Mathematical Foundations of Computer Science – 2nd Symposium. Czechoslovakian Academy of Sciences, 1973.
9. Seppo Keronen. *Computational Natural Deduction.* PhD Thesis, Department of Computer Science, Australian National University, Canberra, 1991.
10. Robert Kowalski. *Algorithm = Logic + Control.* Communications of the ACM 22 424–436, July 1979.
11. Pattie Maes and Daniele Nardi (eds). *Meta-Level Architectures and Reflection.* Elsevier, 1988.
12. L. M. Pereira. 'Logic Control with Logic'. in J.A. Campbell (ed). *Implementations of Prolog.* Ellis Horwood, 1984.
13. Leon Sterling and Arun Lakhotia. 'Composing Prolog Meta-Interpreters.' in Bowen and Kowalski (eds). *Logic Programming: Proceedings of the Fifth International Conference and Symposium.* MIT Press, 1988.
14. Leon Sterling and L.U. Yalcinalp. *Explaining Prolog-Based Expert Systems Using a Layered Meta-Interpreter.* Proceedings 11th International Joint Conference on Artificial Intelligence. Morgan-Kaufmann, 1989.
15. Frank van Harmelen. *Meta-level Inference Systems.* Morgan Kaufmann, 1991.
16. Richard Weybrauch. *Prolegomena to a Theory of Mechanised Formal Reasoning.* Artificial Intelligence 13 133–170, 1980.