Visual Metaphors for Understanding Logic Program Execution

Eric Neufeld, Anthony J. Kusalik, and Michael Dobrohoczki
Department of Computational Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada, S7N 0W0
{neufeld,kusalik,dobro}@cs.usask.ca

Abstract

A classic notion in logic programming is the separation of logic and control. Logic is for problem solving; control is for directing inference. However, practical experience in the classroom suggests that problem-solving students nonetheless devote much effort to understanding control issues such as eliminating looping behaviours and improving program efficiency.

In the case of Prolog, this requires a clear understanding of the operation of both unification and backtracking. Students often try to get this understand by tracing executions, but the common four-port debugger used in Prolog is not as helpful as it could be. In particular, it provides information in low bandwidth textual form.

This paper describes a new visualization system for logic programming that uses *colour tagging* to trace unification through the Prolog proof tree. A user can dynamically "tag" a term or subterm with a colour that is immediately propagrated through the displayed tree. The colour is also propagated through the proof tree on subsequent execution steps. This "colour unification" has an interesting relationship with usual Prolog unification. Initial experiences reveal several interesting visual metaphors that assist understanding of logic program execution. Experience to date also suggests new directions for visualization of logic programs.

Keywords: (program) execution visualization, programming tools, colour

1 Introduction

Advocates of logic programming sometimes sum up their view of computing with Kowalski's [Kow79a] phrase:

$$\text{ALGORITHM} = \text{LOGIC} + \text{CONTROL}\ .$$

This is interpreted to mean that programmers are freed from worrying about control issues. Instead, they become problem solvers, succinctly stating premises and queries in the first-order predicate calculus, after which an independent control module or program executor derives proofs of queries from the premises by appropriately controlling the direction of inference. Kowalski argues [Kow79b] that, among other things, this conceptual separation allows inexperienced programmers to focus on the logic component, leaving the control component to the computer.

Although logical languages such as Prolog give the problem solver a powerful language in which to express problems and indeed free the problem solver from much of the tedium of computer programming, the authors find that even in introductory treatments of Prolog, students spend a lot of time understanding programming techniques dictated by Prolog's procedural semantics. For example, the backtracking and unification algorithms exhibit many subtleties, especially in the context of the 'cut' [OK90]. As well, clauses often must be reordered to eliminate looping behaviours. As a final example, difference lists and accumulator variables, used to improve efficiency, seem to be better understood by some students when given a procedural interpretation.

A useful exercise for Prolog learners, both for gaining an understanding of how Prolog works and for understanding or debugging their own programs, is tracing execution. One possible tool for tracing is the common four-port debugger [Byrd80]. However, this debugger has certain limitations because it provides information in low bandwidth textual form; roughly speaking, it prints the procedure call showing bound variables, as well as other information describing the state of the computation. Novice users find this unsatisfying. They also find the surfeit of information reported baffling or bewildering.

To help learners understand the execution model of Prolog, we have built a protypical system that animates the construction of the Prolog proof tree and uses *colour tagging* to help visualize unification. Most systems for graphically visualizing Prolog [Boja89, DeCl86, EiBr88] focus on portraying success or failure of clauses and backtracking, and use an AND/OR tree or some subset thereof to show unification between a subgoal and a clause head. However, these systems provide little visual information on the effects of unification on the rest of the tree. The system described here dynamically displays the Prolog proof tree, but also allows a user to tag a term or subterm with a colour that is immediately propagated backwards and forwards through the proof tree, and continues to be propagated as the computation continues. Like-coloured terms are joined by coloured lines at the point of unification. The user may subsequently "clear" the view by "uncolouring" terms. Another interesting feature of this system is that it provides a real-time animation of the proof tree, and not just a static display of the final proof. (Some of these animations are interesting in their own right; for instance, nondeterministic programs constructing a list of unbound variables of the right length. This topic is not pursued further here.)

The system was built with the belief that certain visual metaphors would emerge as users worked with the system, and in turn these metaphors would be useful in pedagogy. The speculation was that certain patterns of colour might characterize frequently-used operations (e.g., decomposition of data structures) and movements of colour might characterize others (e.g., returning an answer through an accumulator variable). Initial experience with this system confirms that such metaphors arise and that they seem to aid understanding of program structure and also assist in trouble-shooting incorrect programs. In fact, it seemed easier to debug the examples provided herein because of the availability of colour. Some of the anticipated metaphors turned out to be interesting, and some new metaphors appeared. They are consequences of the movement of colour during program execution, but also of the relationships of coloured figures.

The following section explains how colour unification works. The rest of the paper describes some of the visual metaphors found, and their use for debugging or understanding programs. As programming languages evolve, and more and more computation takes place "behind the scenes" (for example, constructors and destructors in object-oriented languages), it seems reasonable to believe that these ideas will find wider applicability in the visualization of high level programming languages.

2 How colour unification works

This paper presumes an understanding of Prolog's backtracking and unification algorithms [ClMe94, OK90]. (For the reader unfamiliar with Prolog, the following approximation may help. Prolog allows multiple definitions of procedures, in the same way object-oriented languages allow overloading of function definitions, except that Prolog allows multiple definitions of procedures with the same type signature. The execution model allows backtracking through such multiply-defined procedures until a successful execution path is found. Unification can be thought of as Prolog's parameter-passing mechanism. However, unification is much more than that since, for instance, given two different calls of the same procedure, a particular variable could be an input variable in one case and an output variable in the other.)

To track the attachment of bindings to particular terms, a *colour attribute* is associated internally with each term by the Prolog interpreter. Via a graphical user-interface, a user of the system may paint any term, including constant terms, with a colour. The unification algorithm is extended to unify colour attributes whenever two terms successfully unify. Three cases arise:

1. Neither term has a colour; i.e. the colour attribute is not set for either term. In this case, the terms are made to share the same colour attribute. This means that if one of the terms is subsequently given a colour by explicit user action (i.e. "tagging") or via unification, the setting of the colour attribute will also affect the other term. Shared colour attributes let the user colour and uncolour terms "on the fly" to view relationships among different variables.

2. Only one term has colour. The uncoloured term takes on the colour of the coloured term.

3. Both terms have a colour. If the colours are the same, there is nothing to consider. Otherwise there is a *colour conflict.*

The effects of colour unification are undone on backtracking, just as other effects of normal unification.

Several approaches to handling colour conflicts have been investigated. As a technical detail, it is necessary that colour unification succeed whenever normal unification would, even when conflicts do not result in colour matching. The simplest approach is to not change the colour of either term. This makes it easy to locate a point where two constants are matched. Alternately, the colour unification algorithm can choose the dominant colour from the term "higher" in the stack or heap, or use a predefined colour hierarchy. As well, colours could be made to mix in various ways. Initially, the simple approach of not changing either colour when a colour conflict occurs was implemented. However, the system is being extended to allow users to dynamically select the course of action followed by system. Such an extension will allow gathering of feedback on which approaches to the colour conflict problem are most appealing and useful.

A related issue is whether it should be possible to allow identical terms (not made this way by unification) to be coloured differently or whether it should be possible to automatically colour all identical terms identically. Which is ideal will probably be decided empirically by preferences of users.

3 The Colour Prolog System

A Prolog interpreter implementing colour unification and colour tagging has been implemented using C and standard X-Windows libraries. The system provides both a text-oriented interface (for input of queries, and overall control of the system) and a visual interface. The visual interface consists of a large pane in which the execution tree is drawn, and well as buttons for controlling aspects of the execution and scroll bars for positioning large images. Mouse clicks while the cursor is over a term in the drawn tree are used to tag terms with colour. Each mouse button corresponds to a different colour. Mouse clicks are interpreted as toggles, in that tagging an already-coloured term will "untag" (i.e. uncolour) that term.

The visual interface is exemplified in the screen dump in Figure 1 [1]. Since details of that interface are secondary to this discussion, the buttons and scroll bars are eliminated from the screen dumps in remaining figures.

4 Visual metaphors in logic program execution

Colour tagging of certain terms results in interesting characteristic patterns. All images in this section were produced by our system.

[1]The figures can be seen in full-colour in the electronic version of the conference proceedings, or obtained via `http://www.cs.usask.ca/projects/envlop/Colour_Prolog /GI97`.

4.1 Building up answers

The following logic program

```
find_vowel( [], [] ).
find_vowel( [H|T1], [H|T2] ) :-
  vowel(H),
  find_vowel( T1, T2 ).
findvowel( [H|T1], T2 ) :-
  findvowel( T1, T2 ).
vowel( a ).  vowel( e ).  vowel( i ).
vowel( o ).  vowel( u ).
```

when called with a query such as
`?- find_vowel( [a,b,e,i,p,o], Answer )` ,
gradually "builds up" the answer variable, passing additions upward as it recurses through the input list, as Figure 1 shows.

This program is more interesting as an animation, where the user literally sees the answer variable "built up".

4.2 Accumulator variables

More spectacular is the behaviour of "accumulator variables", where answers are accumulated, but passed *downward* (toward the leaves of the tree). When the program encounters its base case, the accumulated result is usually unified with an answer variable and returned upwards (towards the root of the tree). This propagation from the bottom to the top of the tree is generally a significant event during an animation, especially if an answer is passed back after a deep computation. A classic example of a program with accumulator variables is the "list reverse" predicate *reverse/3*:

```
reverse( [], L, L ).
reverse( [Hd|Tl], AccumL, FinalL ) :-
  reverse( Tl, [Hd|AccumL], FinalL ).
```

For reasons of efficiency (see discussion of "naive reverse" below), the answer is constructed by *cons*-ing the current head of the input list to the head of an answer list that must be passed downwards. Thus, in the program above the first argument is the list being reversed, the second is the accumulating reversed list, and the last is the final result (the reversed list). The first two arguments are sometimes merged into a single argument, and the predicate known as "difference-list reverse".

Figure 2 shows an initial query where the user has tagged the two elements of the input list red and green. Figures 3 and 4 illustrate the colours propagating as the program recurses. The "crossovers" show the answer list being built up. Figure 5 gives a

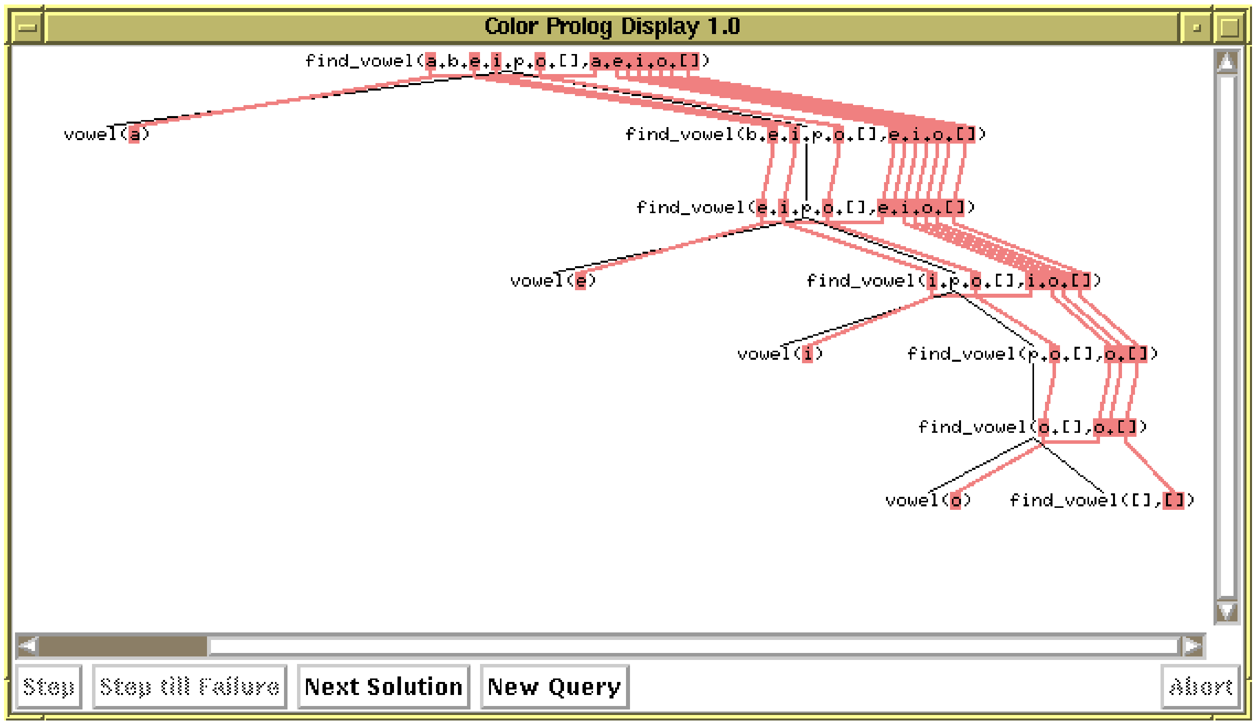

Figure 1: *find_vowel/2* example

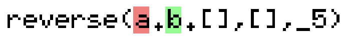

Figure 2: Initial tagging of terms in *reverse/3* query

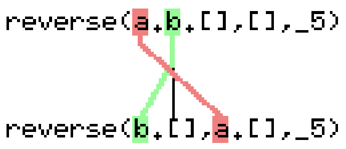

Figure 3: Second step in *reverse/3* query

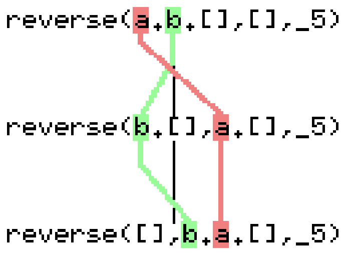

Figure 4: Third step in *reverse/3* query

sense of what happens when the base case is reached and all the colours of the answer list immediately

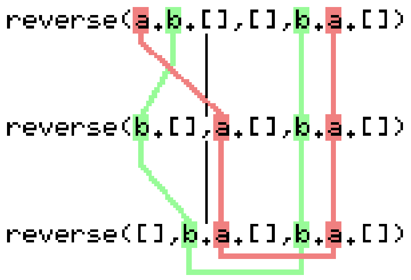
reverse(a.b.[],[],b.a.[])
reverse(b.[],a.[],b.a.[])
reverse([],b.a.[],b.a.[])

Figure 5: Final step in *reverse/3* query

propagate from the bottom to the top of the tree. Such bursts of colour signal completion.

Figure 6 illustrates tagging “on the fly”. The answer variable was coloured blue after resolution was complete. This confirms the action of the difference list accumulator; the portion of the output that comes after the reverse input list is none other than the second argument of the original query.

To give another example of the impression conveyed by completion of accumulation, consider an alternate situation. The initial query is similar to the one before, but the colour tagging scheme is different. In this case, green marks the elements of the input list, red marks the base case of the list, and

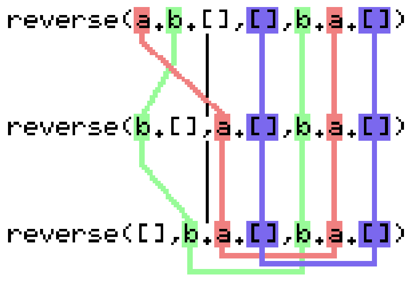
reverse([a,b,[],[],b,a,[])
reverse(b,[],a,[],b,a,[])
reverse([],b,a,[],b,a,[])

Figure 6: On-the-fly tagging in *reverse/3* example

the answer variable is tagged blue. Figure 7 shows execution nearly complete, with green crossovers illustrating the *cons*-ed output list formation. The dramatic movement of blue and green arcs to the initial query in Figure 8 indicates that a solution

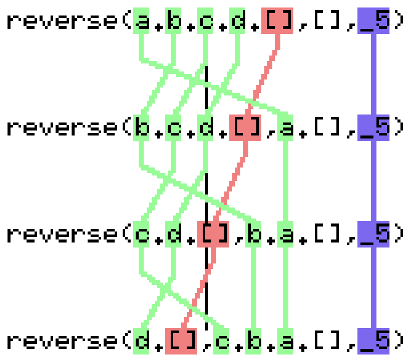
reverse(a,b,c,d,[],[],_5)
reverse(b,c,d,[],a,[],_5)
reverse(c,d,[],b,a,[],_5)
reverse(d,[],c,b,a,[],_5)

Figure 7: Nearing completion of alternate *reverse/3* query

has been found and returned.

4.3 Program efficiency

The logic program of the previous subsection is a classic example of using accumulator variables to make an inefficient program efficient. The so-called "naive reverse"

```
nrev( [], [] ).
nrev( [Hd|Tl], RevL ) :-
  nrev( Tl, RevTl ),
  append( RevTl, [Hd], RevL ).

append( [], L, L ).
append( [Hd|Tl], L2, [Hd|Res]) :-
  append( Tl, L2, Res ).
```

is $O(n^2/2)$ in the size of its input list, since it must on average execute an *append/3* of $n/2$ elements on each of n steps.

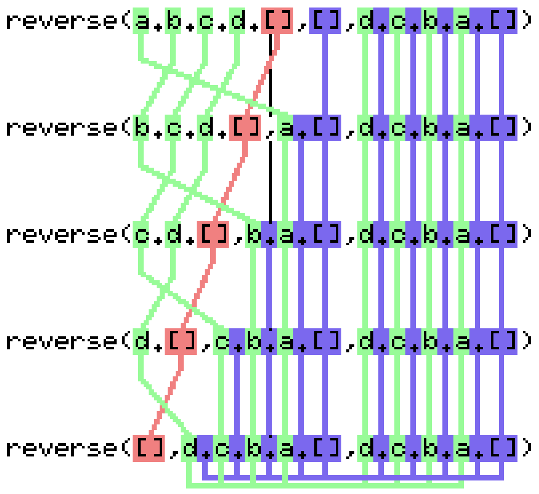
reverse(a,b,c,d,[],[],d,c,b,a,[])
reverse(b,c,d,[],a,[],d,c,b,a,[])
reverse(c,d,[],b,a,[],d,c,b,a,[])
reverse(d,[],c,b,a,[],d,c,b,a,[])
reverse([],d,c,b,a,[],d,c,b,a,[])

Figure 8: Completion of alternate *reverse/3* query

Figure 9 depicts correct execution of the *naive_reverse/2* predicate. In this example, the three different elements of the input list are coloured differently. Note that the proof tree contains three coloured *append* "clusters". Interestingly, each cluster contains a copy of the coloured *append* cluster to its left, suggestive of the execution complexity of the program. This proof is "wide", and contains copies of portions of itself, whereas the proof tree for accumulating list reverse (Figures 2–8) is narrow and doesn't contain this kind of visual redundancy. Hence, the operational behaviour and efficiency for both formulations of list reverse are evident from the output provided by the system.

4.4 Program Errors

Visual metaphors provided by the system have also proven useful for debugging programs: they can portray errors resulting from problems in program logic or from misunderstandings of the procedural semantics of Prolog. For example, the display in Figure 10 shows the final step in a query using an erroneous formulation of the accumulating reverse predicate. From the display, it is evident that the result in the accumulating variable is not being returned (through the last variable). A problem with predicate arguments is suspected. Since terms are being propogated correctly in preceding steps (as evidenced by the colour bands), the error is most likely in the base case of the predicate. This, in fact, is the correct diagnosis, as the erroneous program used to create the display was

```
reverse( [], X, Y ).  % error here
```

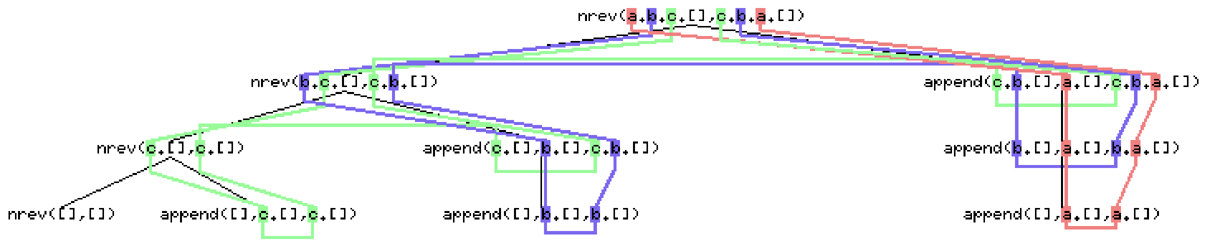

Figure 9: Correctly executed "naive reverse"

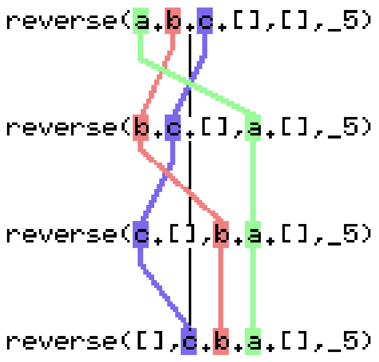

Figure 10: Completion of erroneous *reverse/3* program

```
reverse([H|T],X,Y) :-
  reverse(T,[H|X],Y).
```

Logical errors also frequently express themselves with recurrent visual motifs. A beginning Prolog programmer might write the following formulation for "naive reverse":

```
nrev( [], [] ).
nrev( [Hd|Tl], RevL ) :-
  append( [Hd], Tl, NewRev ),
  nrev( NewRev, RevL ),
```

Figure 11 shows an image given by the system for this example. What should be evident to the user is that the problem (at each level of recursion) is getting no smaller. In fact, the problem at each level is simply a new instance of the previous one. Thus, the user can not only see that there is an error, but what the nature of that error might be.

5 Conclusions and Future Directions

The Prolog visualization system described here provides and supports visual metaphors that may help learners of logic programming more quickly get a better understanding of logic program execution. This paper has presented a sampling of these metaphors, and illustrated how they might be useful.

Other possibilities for assisting users learn and understand logic programs are being implemented and explored. For instance, a peculiarity of Prolog is that when unification fails, everything is lost. However, errors commonly occur when using negation as failure, which works precisely when unification fails. It might be useful for failed branches to leave a faint trace, rather than disappear altogether.

Another idea, also under implementation, is *input program colouring*. The need for such a facility arises when using the system to debug a program with many predicates of the same name and arity. Trying to trace execution of such predicates, especially with the four-port debugger, is confusing. However, tracing is simpler if different input clauses of the same name and arity can be coloured differently; the location of a bug can be pinpointed when it occurs under an instance of a clause of a particular colour.

A problem with the present system is that even modest programs can quickly use up significant screen real estate. Although a Prolog novice can learn a lot by carefully studying a modest program, a wider view for program execution is essential for debugging more practical programs. We are investigating simple perspective transformations that will allow the user to look closely at certain parts of programs, while maintaining contact with relevant data in distant parts of the program through colour traces.

Acknowledgements
We would like to thank the reviewers for some useful constructive criticism on an earlier draft of this paper, and NSERC for research funding.

References

[Byrd80] L. Byrd, "Understanding the Control Flow of PROLOG Programs", Proceedings of the Logic Programming

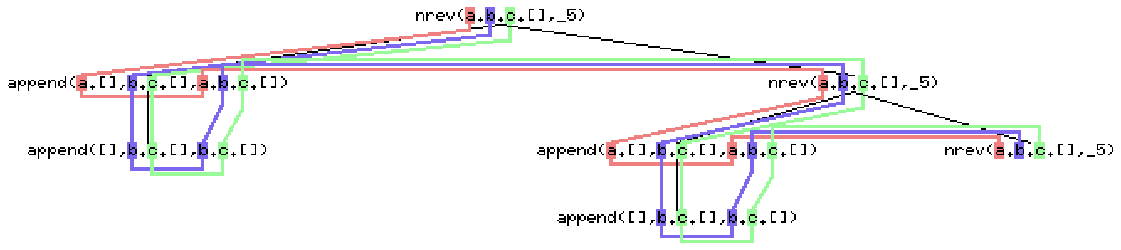

Figure 11: Erroneous “naive reverse”

Workshop, edited by S.-Å. Tärnlund, pp. 127–138, 1980.

[Boja89] D. Bojantchev, “XPGT User’s Guide”, Computer Science and Engineering Department, Case Western Reserve University, Cleveland, Ohio, 1989.

[ClMe94] W. Clocksin and C. Mellish, *Programming in Prolog*, 4th edition, Springer-Verlag, 1994.

[DeCl86] A. Dewar and J. Cleary, “Graphical display of complex information within a Prolog debugger”, in *International Journal of Man-Machine Studies*, Vol. 25, 1986, pp. 503–521.

[EiBr88] M. Eisenstadt and M. Brayshaw, “The Transparent Prolog Machine (TPM): An execution model and graphical debugger for logic programming”, *Journal of Logic Programming*, Vol. 5, No. 4, 1988, pp. 277-342.

[HS96] J. D. Horton and B. Spencer, “Clause trees: a tool for understanding and implementing resolution in automated reasoning”, *Artificial Intelligence*, accepted for publication.

[Kow79a] R. Kowalski. “Algorithms = Logic + Control”, *Communications of the ACM*, Vol. 22, No. 7, pp. 424–436, 1979.

[Kow79b] R. Kowalski, em Logic for Problem Solving, Elsevier Science Publishing, 1979.

[OK90] R. O’Keefe, em The Craft of Prolog, MIT Press, 1990.