Prolog without tears: An evaluation of the effectiveness of a non Byrd Box model for students

Paul Mulholland
Human Cognition Research Laboratory
Open University
Milton Keynes
MK7 6AA
p.mulholland@open.ac.uk

Acknowledgements: This work was supported by an EPSRC postgraduate research studentship.

1. Introduction

Prolog is a difficult language to learn (Taylor, 1988). A large number of Prolog tracers or debuggers have been developed to aid understanding. A central aim of these is to provide a clear and consistent account of Prolog execution (du Boulay *et al.*, 1981). An earlier study evaluated the efficacy of four different tracers for Prolog novices. The study found major differences in their performance. which was related to how well students were able to access information from the display, develop comprehension strategies and avoid misconceptions (Mulholland, 1994). Some misconceptions were very difficult to avoid within the conventional Byrd Box model of Prolog which underpinned all of the tracers.

Following on from this work a choice-point execution metaphor was adopted to provide a less confusing model of execution. Two tracers were developed using the model. The design aim was to provide a clear representation of the model in a way which would facilitate access to data flow and control flow information.

Two versions of the tracer were compared against the Prolog Trace Package (PTP) (Eisenstadt, 1984) which had been found to be reasonably successful in the earlier evaluation study. The comparison identified information access from the display, an inventory of comprehension strategies employed, and misunderstandings of the notation as well as completion rates on the task. The findings raise issues as to how the various representations of Prolog can best be understood by the HCI community and what role the tracer should play within the learning environment.

2. Earlier work

This study builds on earlier work comparing the suitability of four Prolog tracers for novice Prolog programmers (Mulholland, 1994). These were Spy (Byrd, 1980); the Prolog Trace Package (PTP) (Eisenstadt, 1984); the Transparent Prolog Machine (TPM) (Eisenstadt & Brayshaw, 1990; Brayshaw & Eisenstadt, 1991) and the Textual Tree Tracer (Taylor *et al.*, 1991).

Spy provides a linear textual trace of execution using the Byrd Box model of execution incorporating four status codes: call, exit, fail and redo. The basic model of execution shown in Spy underpins the other more complex representations of Prolog. PTP is a linear textual trace providing a richer account of Prolog execution. PTP uses a greater range of symbols to differentiate types of goal failure and explicitly distinguish between the

goals of the execution and the clauses within the program. TPM attempts to provide the same richness of information in a more accessible form by representing Prolog as an AND/OR tree. Fine-grained views of individual nodes of the tree provide a detailed account of variable bindings. TTT uses a non-linear textual notation to provide a hierarchical representation of Prolog in a form closer to the source code.

TPM faired less well than the three textual tracers. This was most likely due to the larger learning curve of a more complex graphical notation. As a result subjects using TPM were less able to access data flow and control flow information and frequently failed to appreciate the position within the execution being presented. Subjects using Spy performed less well than those using TTT and PTP. This seemed largely due to the confusing way unification is represented leading to a number of misunderstandings of control flow, data flow and the relation between goals in the execution and clauses in the program. Although PTP and TTT performed similarly well the protocols highlighted important differences in the way the two tracers were used. The non-linear execution of TTT allowed a greater focus on data flow though as in Spy the clause and goal were often confused. PTP provided a clear representation of control flow though some problems were still identified, particularly during backtracking.

This work demonstrated the benefits of using a fine-grained protocol-based account of the user rather than relying solely on timing data in order to gain a fuller understanding of how the tracer performs and why.

3. The choice-point model

All existing tracers adopt the Byrd Box style model of execution which underpins the Spy tracer (Byrd, 1980). The results gained from the previous study suggest that certain principles of Prolog execution are difficult to represent within the Byrd box model, in particular backtracking. In order to combat these problems a choice-point model of execution was adopted (Dodd, 1993). The key feature of the choice-point model is that as each clause is entered, the interpreter "looks ahead" to see if any later clauses could match should the present route fail. If so, this is marked as a choice-point. Backtracking can then be shown as jumping back to the nearest choice-point.

This model was combined with promising notational techniques found in existing tracers to develop two Prolog choice-point tracers: the Prolog Linear Tracer (Plater) and the Prolog Non-linear Tracer (Pinter). These also shared a new textual representation of binding, loosely based on the lozenge notation used in TPM (Eisenstadt & Brayshaw, 1990).

For example, given the query `fun(What)` and the program below:

```
fun(X) :-
      car(X),
      gold(X).
fun(X) :-
      bike(X),
      silver(X).
car(mini).
bike(honda).
silver(honda).
```

Plater would begin by showing the query:

```
• ?  fun(What)
```

A further step shows entry into the first rule:

```
• ?  fun(What)
• »1 fun({What=X})
```

The » symbol indicates that at least one more matching clause is available should the present path fail. Bindings between clause and goal are shown in braces. The current path fails on the second subgoal of the rule. The symbol -d indicates that failure was due to the goal not being defined in the database:

```
• ?  fun(What)
• »1 fun({What=X})
•    ?  car(What)
•    +1 car({What=mini})
•    ?  gold(mini)
•    -d gold(mini)
```

Backtracking is then shown as a jump to the nearest choice-point. Bindings occurring within the failed path are lost (shown by = being replaced by ≠) .

```
• ?  fun(What)
 «»1 fun({What≠X})
     ?  car(What)
     +1 car({What≠mini})
     ?  gold(mini)
     -d gold(mini)
     ^^^^^^^^^^^^^^^
```

The execution then continues using the next available path:

```
• ?  fun(What)
  «1 fun({What≠X})
     ?  car(What)
     +1 car({What≠mini})
     ?  gold(mini)
     -d gold(mini)
     ^^^^^^^^^^^^^^^
• >2 fun({What=X})
```

Pinter executes in a similar way except that goal lines are updated rather than rewritten to preserve the goal hierarchy.

4. Methodology

The motivation behind the methodology is to allow the gross performance measures to be explained in terms of the information types, strategies and misunderstandings which occur as a result of the notation and its navigation. The motivation behind the methodology is explained in detail elsewhere (Mulholland, 1993) and has been employed in a similar study (Mulholland, 1994).

The assumption underlying the analysis of information types is that when trying to gain a detailed understanding of the code the subjects will derive certain types of information more readily depending on how clearly they are represented in the display. The information taxonomy also classifies referrals to the source code and utterances connected with understanding the features of the trace. The information types are outlined in table 1.

Information code	Description
CFI	Derive control flow information from the trace
DFI	Derive data flow information from the trace
ETO	Compare to an earlier trace output
GOAL	Comment on the goal of all or part of the program
PRED	Predict future behaviour of the trace
READ	Read the trace output
SOURCE	Refer to or reconstruct source code
TRACE	Comment on navigation or notation of trace

Table 1: Protocol coding scheme for information types.

A number of identified comprehension strategies that subjects develop in order to understand the information the tracer presents to them are explained in table 2.

Strategy code	Description
REVIEW CF	Review previous execution steps
REVIEW DF	Review previous data flow
TEST CF	Predict and test future steps of the trace
TEST DF	Predict and test future bindings of variables
EXPERIENCE	Compare against previous experience of the tracer
SOURCEMAP	Map successive steps of the trace against the code
OVERVIEW	Comment on the overall trace output at some point

Table 2: Strategies identified in the protocols.

The four types of misunderstanding of the trace are outlined in table 3.

Misunderstanding code	Description
CGM	Confusing the clause and its associated goal
CFM	Deriving an incorrect model of control flow from the trace
DFM	Deriving an incorrect model of data flow from the trace
TM	Time misunderstanding: failing to appreciate the point in the execution currently being represented

Table 3: Misunderstandings of the trace identified in the protocols.

5. Outline of the study

The study was carried out involving 48 Open University summer school cognitive psychology students taking the Artificial Intelligence project. Students taking the project are required to model a simple cognitive theory in Prolog. Each summer school project lasts approximately 2.5 days. Each tracer was used as the main teaching focus and sole debugging aid for one week (i.e. two AI project groups). Prior to the summer school the students had completed assessed work using Prolog to model a simple AI problem. The level of exposure to the trace was approximately treble that of the previous study.

A three level between subjects design was used with 16 subjects per cell working in pairs. Each pair of subjects were given five minutes to familiarise themselves with a program presented on a printed sheet. They each retained a copy of this program throughout the experiment. The program was an isomorphic variant of the one used by Coombs and Stell (1985) to investigate backtracking misconceptions. They were then asked to work through the traces of four versions of the program which had been modified in some way. Their task was to identify the difference between the program on the sheet and the one they were tracing. They had no access to the source code of the modified versions. Subjects were asked whether they wished to move onto the next task if the end of the trace had been reached without identifying the change. Verbal protocols were taken throughout.

Program modifications were selected which required the novice to focus on different types of information in order to correctly identify the change. The four problems given were a change in a relation name, a changed atom name, a data flow change and a control flow change. The data flow change was either passing the wrong variable from a rule or changing a variable within a rule to an atom. The control flow change was either a swap in the subgoal order of a rule or the fact order within the database.

A post-test questionnaire was administered to derive feedback on the tracer and its role within the course.

5. Results

The mean completion rates are shown in table 4. A one factor ANOVA revealed a main effect for tracer, $F(2, 21) = 3.627$, $p < 0.05$. A pairwise comparison revealed significant differences between Plater and Pinter ($p <$ 0,05) and Plater and PTP ($p < 0.05$).

Tracer	Plater	Pinter	PTP
Solutions	3.875	3.000	2.875

Table 4: Mean number of problems completed.

An analysis of the level of information access revealed no significant differences between the three tracers. A two factor ANOVA of the comprehension strategies identified in the protocols revealed main effects for strategy, $F(6, 126) = 23.853$, $p < 0.01$; and tracer, $F(2, 21) = 4.244$, $p <$ 0.05. A pairwise comparison revealed a significant difference between PTP and Plater ($p < 0.05$). Simple effects were found for REVIEW DF ($p < 0.05$) and SOURCEMAP ($p < 0.01$). Table 5 shows the mean number of comprehension strategies identified for each tracer.

Tracer / Strategy	Plater	Pinter	PTP
REVIEW CF	2.00	1.13	1.00
REVIEW DF	4.38	3.00	1.50
TEST CF	4.88	4.87	3.50
TEST DF	2.38	1.00	1.50
EXPERIENCE	0.63	0.13	0.38
SOURCEMAP	6.00	7.13	3.75
OVERVIEW	0.00	0.00	0.00

Table 5: Mean number of comprehension strategies for each subject pair.

The mean number of misunderstandings of the tracer are shown in table 6. A two-factor ANOVA revealed main effects for misunderstanding, $F(3, 63) = 3.597$, $p < 0.05$; and tracer, $F(2, 21) = 4.324$, $p < 0.05$. A pairwise comparison revealed a significant difference between Pinter and Plater ($p < 0.05$).

Tracer / Misunderstanding	Plater	Pinter	PTP
CGM	0.25	0.63	0.75
CFM	0.00	0.00	0.25
DFM	0.00	0.13	0.25
TM	0.00	0.75	0.00
Total	0.25	1.48	1.22

Table 6: Mean number of misunderstandings of the trace per subject pair.

PTP did not perform significantly differently from the previous study on problems completed, information access, comprehension strategies or misunderstandings.

The post-test questionnaire revealed most Pinter subjects found the non-linear development of the trace confusing. Subjects from each tracer suggested an on-line symbol key would be very useful.

6. Discussion

No improvement was found in the performance of PTP though the level of exposure to the trace had increased approximately threefold. This suggests that once the students are relatively familiar with the tracer, an increase in use alone will not improve their performance. This can be contrasted with the preliminary results of an ongoing study of Prolog experts where the subjects were able to use a new tracer competently after only a few minutes. It therefore appears that knowledge of the programming domain is a far more important determiner of performance than familiarity with the software and its notation. Stasko *et al.* (1993) argued the main reason an algorithm animation had not been found to improve understanding was because the amount of useful information that can be derived is bounded by the students' knowledge of what the features of the animation map to in the programming domain.

These findings raise an important educational issue. If the ability to comprehend a visualization of a program is related to the understanding of the constructs found within the program then the educational role of such software should be limited to helping the student to consolidate lessons learnt in the classroom rather than as a direct method of teaching in itself.

No differences in information access rates were found though differences were found in the use of comprehension strategies and the number of misunderstandings. There are two likely reasons why there was not a direct relation between information access and comprehension strategies. Firstly, though the amount of information accessed may have been similar, the amount of cognitive effort required to derive that information may have varied leaving less resources for the application of strategies. Secondly, strategies rely more heavily on the overall gestalt of the trace rather than on a single line. These therefore draw on different qualities of the notation including its dynamics. In particular, the textual lozenge notation in the choice-point tracers helped students to review the flow of data through the execution. The explicit representation of bindings to variables in the code also encouraged subjects to check the mapping between the execution and the source code.

The most notable finding from the analysis of misunderstandings is the large number of timing misunderstandings in Pinter, more than were found previously in the textual non-linear tracer TTT. It appears that the richness of the information found in Pinter hindered the clear gestalt necessary for the dynamics of a non-linear tracer to be followed.

Plater was found to be a very useful tracing tool for novices both in the encouragement of useful comprehension strategies and the minimising of misunderstandings. An improved choice-point tracer called Theseus has been built based closely on Plater. A supporting analogy of Prolog execution has also been used, comparing Prolog execution to the mythological story of Theseus finding his way through the labyrinth (Rose, 1965). Many students reported finding the analogy useful when comprehending the working of their own programs.

7. Conclusion

The development of a choice-point model of Prolog raises the question as to how the various representations of Prolog can best be understood. Pain and Bundy (1987) outlined a number of Prolog stories that can be used to represent Prolog execution such as a linear textual notation (e.g. Spy) or an AND/OR tree notation such as TPM. All of these stories represented the Byrd box model of execution in different ways. The choice-point tracers differ in a new way. The notational constructs employed are not particularly novel though the underlying execution model is. It therefore seems unsatisfactory to describe them simply as other stories.

I argue that a two tier description would be more appropriate where the story referred to the underlying execution model (e.g. Byrd box, choice-point) and the notational constructs employed provided differing accounts of each story. Much of the work in visualization focuses on telling the truth about a particular language, though this is only a relative truth based on the currently accepted notion of how the language works. The aim of research into teaching programming should not only be concerned with how best to represent an accepted story but also to devise new stories consistent with the behaviour of the language which may better describe its functioning.

References

Brayshaw, M. & Eisenstadt, M. (1991). A Practical Tracer for Prolog. *International Journal of Man-Machine Studies*, 42, 597-631.

Byrd, L. (1980). Understanding the control flow of Prolog programs. *Proceedings of the Logic Programming Workshop.* Debrecen, Hungary.

Coombs, M.J. and Stell, J.G. (1985). *A model for debugging Prolog by symbolic execution: the separation of specification and procedure.* Research Report MMIGR137. Department of Computer Science, University of Strathclyde.

Dodd, T. (1993). A choice-point model of Prolog execution. *ALP-UK Workshop on Logic Programming Support Environments*, Edinburgh.

du Boulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, 14, 237-249.

Eisenstadt, M. (1984). A Powerful Prolog Trace Package. *Sixth European Conference on Artificial Intelligence*, Pisa, Italy.

Eisenstadt, M. & Brayshaw, M. (1990). A fine grained account of Prolog execution for teaching and debugging. *Instructional Science*, 19(4/5), 407-436.

Mulholland, P. (1993). *Evaluating Program Visualization Systems: An information-based methodology.* Technical Report 107. Human Cognition Research Laboratory. Open University.

Mulholland, P. (1994). The effect of graphical and textual visualization on the comprehension of Prolog execution by novices: an empirical analysis. In *Proceedings of the Psychology of Programming Interest Group.*

Pain, H. & Bundy, A. (1987). What stories should we tell novice PROLOG programmers? In *R. Hawley (Ed.) Artificial Intelligence Programming Environments* , pp 119-130.

Rose, H.J. (1965). *A Handbook of Greek Mythology.* Frome: Tanner.

Stasko, J., Badre, A., & Lewis, C. (1993). Do algorithm animations assist learning? An empirical study and analysis. In *Proceedings of INTERCHI '93.*

Taylor, C., du Boulay, B., & Patel, M. (1991). *Outline proposal for a Prolog 'Textual Tree Tracer' (TTT)* . CSRP No. 177, University of Sussex.

Taylor, J. A. (1988). *PROGRAMMING IN PROLOG: An In-Depth Study of the Problems for Beginners Learning to Program in Prolog.* PhD Thesis, University of Sussex.