From: AAAI-87 Proceedings. Copyright ©1987, AAAI (www.aaai.org). All rights reserved.

PROLEARN: Towards A Prolog Interpreter that Learns[1]

Armand E. Prieditis and **Jack Mostow**
Department of Computer Science
Rutgers University, New Brunswick, NJ 08903

Abstract

An *adaptive interpreter* for a programming language adapts to particular applications by learning from execution experience. This paper describes PROLEARN, a prototype adaptive interpreter for a subset of Prolog. It uses two methods to speed up a given program: explanation-based generalization and partial evaluation. The generalization of computed results differentiates PROLEARN from programs that cache and reuse specific values. We illustrate PROLEARN on several simple programs and evaluate its capabilities and limitations. The effects of adding a learning component to Prolog can be summarized as follows: the more search and subroutine calling in the original query, the more speedup after learning; a learned subroutine may slow down queries that match its head but fail its body.

I Introduction

How could an interpreter adapt to its execution environment by learning from execution experience and thereby customize itself toward particular applications? This paper describes PROLEARN, a prototype *adaptive interpreter* for a subset of Prolog. PROLEARN handles all of Prolog except for the cut symbol (used to control backtracking) and side-effects (primitives that cause input or output or change the database). PROLEARN learns new subroutines that represent justifiable generalizations of example executions. (We will use "subroutine" to mean a user-defined Prolog rule, as opposed to a primitive or a fact.)

Search reduction is an important way to increase the performance of search-based problem-solving system [Mitchell *et al.*, 1986, Mitchell *et al.*, 1983, Minton, 1985, Langley, 1983, Laird *et al.*, 1986b, Fikes *et al.*, 1972, Mahadevan, 1985, Korf, 1985]. Because Prolog uses search as a basic mechanism and has built-in unification, it is a natural vehicle for developing adaptive interpreters.

PROLEARN combines explanation-based generalization (EBG) [Mitchell *et al.*, 1986] with partial evaluation [Kahn and Carlsson, 1984, Kahn, 1984] to learn from execution experience and thereby reduce future search. As PROLEARN interprets each subroutine call, it uses EBG to compute the general class of queries solved by the same execution trace. Partial evaluation techniques are used to simplify the generalized execution trace for more efficient execution.

The rest of this paper is organized as follows. Section II describes EBG in PROLEARN and Section III describes partial evaluation in PROLEARN. PROLEARN is only a prototype for a practical adaptive interpreter; Section IV discusses some of its shortcomings and suggests possible improvements. Section V summarizes the empirical results. Section VI discusses related work. Finally, Section VII summarizes the research contributions of this work.

II EBG in PROLEARN

As it executes a program, PROLEARN treats every subroutine call as a goal concept for EBG to operationalize in terms of primitives and facts. Using the specific execution trace as a template, it constructs a customized version of the subroutine, as follows. PROLEARN records all calls to primitive subroutines and user-defined facts performed in the course of executing the subroutine, generalizes them to remove the dependence on the particular arguments passed to the subroutine, and conjoins the generalized calls. The resulting conjunction is used to define a new special case version of the original subroutine.

To illustrate, consider the following database (from [Mitchell *et al.*, 1986]):

```
on(box1,table1).
volume(box1,10).
isa(box1,box).
isa(table1,endtable).
color(box1,red).
color(table1,blue).
density(box1,10).

safe_to_stack(X,Y) :-
   lighter(X,Y);not(fragile(Y)).

lighter(X,Y) :-
   weight(X,W1),
   weight(Y,W2),
   W1 < W2.
```

[1]This work is supported by NSF under Grant Number DMC-8610507, and by the Rutgers Center for Computer Aids to Industrial Productivity, as well as by DARPA under Contract Number N00014-85-K-0116

```
weight(X,Y) :-
   volume(X,V),
   density(X,D),
   Y is V * D.

weight(X,500) :- isa(X,endtable).
```

Given the query `safe_to_stack(box1,table1)`, PROLEARN learns the subroutine definition

```
safe_to_stack(X,Y) :-
   clause(volume(X,V),true),
   clause(density(X,D),true),
   M is V * D,
   clause(isa(Y,endtable),true),
   M < 500.
```

That is, object X is safe to stack on object Y if the product of X's volume and density is less than 500 and Y is an endtable. The learned subroutine uses `clause` to ensure that user-defined predicates like `volume`, `density`, and `isa` are evaluated solely by matching stored facts; otherwise the learned subroutine might invoke arbitrarily expensive subroutines for these predicates, thereby defeating its efficiency. PROLEARN also learns similar specialized versions of each subroutine invoked in the course of executing `safe_to_stack` (e.g., `lighter`).

EBG in PROLEARN amounts to subroutine unfolding with generalization. Unfolding a query into its primitive calls effectively caches the result of searching for the intermediate subroutines needed to answer the query. For example, when the query `weight(table1,W2)` is evaluated in the course of executing `safe_to_stack(box1,table1)`, PROLEARN must determine which definition of `weight` to use. As it turns out, the first definition fails and it must use the second one instead, namely `isa(obj2,endtable)`. The learned subroutine, which is unfolded in terms of primitive calls like `isa(Y,endtable)`, eliminates this search. While subroutine unfolding provides some speedup in procedural languages, in Prolog the speedup can be much greater because of this search reduction.

While the learned subroutine is a special case of the original definition, it is a generalization of the execution trace generated for the specific query. Therefore it will apply to queries other than the one that led to its creation, and should speed up interpretation for the entire class of such queries. This class is restricted to queries whose execution would have followed the same execution path as the original query. Relaxing this restriction would require allowing learned subroutines to call user-defined subroutines, which would make learned subroutines more general but possibly less efficient.

PROLEARN relies on EBG to guarantee that this generalization is justifiable. An execution trace constitutes a proof of a query in the "theory" defined by the subroutines and facts in the program. EBG generalizes away only those details of the trace ignored by this "proof." The learned subroutine should therefore work for any query it matches, that is, produce the same behavior as the original program. (Section IV discusses the problem of preserving correctness of the original program.)

III Partial Evaluation

EBG sometimes produces a conjunction of terms that can be simplified by exploiting constraints about the learned subroutine. This simplification is similar to partial evaluation without execution as described in [Kahn and Carlsson, 1984] and [Bloch, 1984].

The need for partial evaluation of the generalization becomes evident in the following example. The program below solves the Towers of Hanoi puzzle—where $N \geq 0$ disks are `move`'d from the `From` pole to the `To` pole via the `Using` pole. The program uses a standard recursive formulation of the solution: move N-1 disks from the `From` pole to the `Using` pole, move one disk from the `From` pole to the `To` pole, and finally move N-1 disks from the `Using` pole to the `To` pole. The resulting plan is bound to the variable `Plan`.

```
move(0,_,_,_,[]).
move(N,From,To,Using,Plan) :-
   N > 0, M is N-1,
   move(M,From,Using,To,Subplan1),
   move(M,Using,To,From,Subplan2),
   append(Subplan1,[[From,To]],Frontplan),
   append(Frontplan,Subplan2,Plan).

append([],L,L).
append([H|T],L,[H|U]) :- append(T,L,U).
```

Given the query `move(3,left,right,center,Plan)`, `Plan` is bound to:

```
[[left,right],[left,center],
[right,center],[left,right],[center,left],
[center,right],[left,right]]
```

Since `move` contains recursive calls to `move`, PROLEARN learns three `move` subroutines. In particular it learns the following subroutine:

```
move(N,From,To,Using,[[From,To],[From,Using],
   [To,Using],[From,To],[Using,From],
   [Using,To],[From,To]]) :-
   N>0, M is N-1, M>0, L is M-1,
   L>0, 0 is L-1, L>0, 0 is L-1,
   M>0, K is M-1, K>0, 0 is K-1,
   K>0, 0 is K-1.
```

The `move` definition learned by EBG represents a specialized subroutine for moving 3 disks from the `From` pole to the `To` pole via the `Using` pole. The newly learned `move`'s for two disks and one disk are similarly verbose. Worse still, future queries with `move` of $N > 3$ disks will match the learned rule and not fail until they reach the conjunct `0 is L-1`.

The term `0 is K-1` (in the above conjunction) could be eliminated by binding `K` to `1` instead. In general, if two of the unknowns in an expression like `X is Y - Z` are known then the third can be deduced. PROLEARN uses

such rules to simplify primitive subroutine calls. PROLEARN actually learns the above move as:

```
move(3,From,To,Using,[[From,To],[From,Using],
    [To,Using],[From,To],[Using,From],
    [Using,To],[From,To]]).
```

PROLEARN partially evaluates the following kinds of terms: terms like `3 < 5` (that are always true) are eliminated, terms like `X == Y` (a test on whether X and Y are the same variable) and `X = Y` (a test on whether X has the same value as Y) are eliminated by binding X to Y, and terms like `12 is X * 4` are eliminated by binding X to 3. Terms like `4 is Y/3` are simplified to conjunctions like `((Y > 11), (Y < 15))` to exploit PROLEARN's integer division. Early cutoff occurs when $Y \leq 11$.

After partial evaluation, the learned subroutine is added to the database. To ensure that it is tried before the less efficient original version from which it was derived, PROLEARN inserts it above existing subroutines.

IV Limitations

PROLEARN exhibits a number of shortcomings common to other problem-solving architectures that learn.

A. The Search Bottleneck Problem

As PROLEARN learns more subroutines, search time gradually increases. As an example, consider **member**, a commonly-used Prolog subroutine that tests for membership of an item in a list:

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

Given the query `member(a,[b,c,d,a])`, PROLEARN learns the following three **member** definitions (since **member** is called recursively three times). They represent membership of an item at the fourth, third and second positions in an arbitrary length list.

```
member(X,[T1,T2,T3,X|_]).
member(X,[T1,T2,X|_]).
member(X,[T1,X|_]).
```

Given the query `member(a,[b,c,d,e,a])`, PROLEARN must search through all the learned **member** definitions to get to the general case for **member**. A query testing the membership of an object in the list of a hundred items may generate ninety-nine new **member** definitions—clogging up the interpreter if most of them are executed again.

A number of researchers have observed that uncontrolled learning of macro-operators can cause search bottlenecks [Minton, 1985, Iba, 1985, Fikes *et al.*, 1972]. Minton's program used the frequency of use and the heuristic usefulness of macro-operators as a filter to control learning. Iba's program learned only those operator sequences leading from one state with peak heuristic value to another.

B. The Learning Worth Problem

Given the tradeoffs between storing and computing, between the cost of learning and the resulting speedup, and between speeding up some queries at the cost of slowing down others, what is worth learning and remembering? As an example of possibly *worthless* learning, consider the following database (after [Mahadevan, 1985]):

```
equiv(X,X).
equiv(¬¬X,Y) :- equiv(X,Y).
equiv(¬(X ∨ Y),M) :- equiv(¬X ∧¬Y,M).
equiv(¬(X ∧ Y),M) :- equiv(¬X ∨¬Y,M).
equiv(X ∧ Y,M ∧ N) :- equiv(X,M), equiv(Y,N).
equiv(X ∨ Y,M ∨ N) :- equiv(X,M),equiv(Y,N).
```

The subroutine **equiv** is used to test whether its two arguments are equivalent boolean expressions (with the standard logical operators of ¬, ∧ and ∨). Consider the following query:

```
equiv(¬¬(a ∨ b) ∧ ¬¬(c ∨ d),(a ∨ b) ∧ (c ∨ d))
```

Prolog interpreters typically index on the subroutine's name (**equiv**) and the first argument's principal functor (∧) to find relevant subroutines. Here the only match is the subroutine whose head is `equiv(X ∧ Y,M ∧ N)`. In fact, there is only one relevant subroutine at each node of the entire search tree for this query, so its *branching factor* is 1. When branching factor is this low, there is no search to eliminate. Here, speedup derives only from elimination of subroutine calls in the learned rules, namely:

```
equiv(¬¬X ∧¬¬Y,X ∧ Y).
equiv(¬¬X,X ).
```

Whether a query is worth learning from depends on its execution cost (search and subroutine calling) without learning, the cost of the learning process, and the distribution of subsequent queries. PROLEARN simply learns from *every* execution without considering these factors.

C. The Correctness and Redundancy Problem

Is the set of programs still correct after PROLEARN learns new subroutines? That is, does the new program preserve the user-intended behavior? Two problems, redundancy and over-generalization, make this question difficult to answer.

PROLEARN disallows subroutines with side-effects to avoid redundancy. Since every learned subroutine in PROLEARN represents another way to execute a particular query, the database of original subroutines already describes how to execute the query (perhaps in a less efficient manner). The danger lies in side-effects that may be repeated and thus change program behavior in a way that violates the intent of the original program.

PROLEARN also disallows the cut symbol (used to control backtracking and define negation-as-failure). If PROLEARN allowed the cut symbol, learning new subroutines would, in general, be impossible without considering

the context of other subroutines (e.g., cut and `fail` combinations that were executed during a particular query).

While applying PROLEARN to a program with side effects is not guaranteed to preserve its behavior, neither does it necessarily lead to disaster. Demanding that adaptive interpreters preserve program behavior is unnecessarily restrictive, since not all behavior changes are unacceptable [Mostow and Cohen, 1985, Cohen, 1986]. Further work is needed to characterize and expand the class of programs PROLEARN can interpret without changing program behavior in unacceptable ways.

PROLEARN is also theoretically subject to the same over-generalization problem as SOAR [Laird *et al.*, 1986a], in which a learned rule masks a pre-existing special-case rule that didn't apply when the rule was learned. It appears possible (though unwieldy in Prolog) to overcome this problem by placing each learned subroutine immediately above the subroutine from which it was derived rather than at the top of the database.

V Summary of Empirical Results

Query	EBG	EBG + Part. Eval.	Avg. Brchg Factor	User Subr. Calls
`safe_to_stack`	23	23	1.3	6
`move`	11	94	1	36
`member`	4	4	1	3
`equiv`	2	2	1	6

Table 1: Speedup Factors Over Original Query in Prolog

Table 1 lists the speedup factors in CPU time over the original query in Prolog for the examples presented above. For each example, the table also lists the average branching factor and number of calls to user-defined subroutines for the original query. The results can be summarized as follows:

- For EBG alone, the example with the largest average branching factor (`safe_to_stack`) produced the greatest speedup. The next greatest speedup resulted from eliminating the user subroutine calls (`move`).
- The two examples with branching factor 1 and few user subroutine calls (`member` and `equiv`) had the least speedup. Any speedup from EBG resulted from subroutine call elimination, not search reduction.
- The one example (`move`) where partial evaluation helped was sped up by a factor of 8.5 over EBG because costly arithmetic operations were eliminated.

Since each learned subroutine applies to a whole class of queries, the speedup results apply to the entire class. However, the results presented above are incomplete, since they do not show the slowdown for queries outside this class. Such slowdown occurs when a query matches a learned subroutine and then fails after executing some of the terms in its definition.

We measured speedup by comparing Prolog execution time before and after learning. This comparison measures how much speedup would be achieved if PROLEARN's techniques were *efficiently* implemented in the Prolog interpreter itself. PROLEARN is *actually* implemented as a simple but inefficient Prolog meta-interpreter. The extra level of interpretation imposes a considerable performance penalty, largely because it loses the efficiency of Prolog's indexing. Although learning speeds up PROLEARN's execution of the example queries by factors ranging from 4 to 32, this speedup is outweighed by the interpretation penalty, which renders PROLEARN 54 to 156 times slower than Prolog.

VI Related Work

Like Soar [Laird *et al.*, 1986b], PROLEARN is an *incidental* learning system because it learns as a side-effect of problem-solving. While PROLEARN's cached subroutines resemble Soar's chunks, Soar assumes that multiple occurrences of a constant are instances of the same variable, which can cause it to learn chunks that are more specific than necessary. PROLEARN uses EBG, which avoids this assumption. Also, Soar's chunking mechanism does not use partial evaluation to simplify learned chunks.

Unlike systems that cache specific values [Mostow and Cohen, 1985, Lenat *et al.*, 1979], PROLEARN stores generalized procedures. Both approaches pay for decreased execution time with increased space costs and lookup time.

PROLEARN adapts programs to their execution environments more dynamically than typical program optimizers [Aho *et al.*, 1986, Kahn and Carlsson, 1984, Bloch, 1984]. While some optimizers use data about the execution environment [Cohen, 1986] or collect statistics about the execution frequency of different control paths in a program, they do not generalize as PROLEARN does.

VII Conclusion

This paper has shown how an interpreter can adapt to its execution environment and thus customize itself to a particular application. PROLEARN, an implemented prototype of an adaptive Prolog interpreter, uses two methods to increase its performance: explanation-based generalization and partial evaluation. The generalization of computed results differentiates PROLEARN from programs that cache and reuse specific values.

The effects of adding a learning component to Prolog can be summarized as follows:

- The more search and subroutine calls in the original query, the more speedup after learning: the indexing and backtracking caused by search are eliminated, as well as the overhead of subroutine calls.
- The same speedup applies to the class of queries that match the learned subroutine, not just the query from which the subroutine was learned. This class includes queries that would have followed the same execution path as the original query.
- A learned subroutine may slow down queries that match its head but fail its body.

Acknowledgments

Many thanks go to Tony Bonner, Sridhar Mahadevan, Prasad Tadepalli, Steve Minton, Tom Fawcett, and Smadar Kedar-Cabelli for their comments on this paper. Thanks also go to Chun Liew and Milind Deshpande for their help with LaTeX.

References

[Aho *et al.*, 1986] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[Bloch, 1984] C. Bloch. *Source-to-Source Transformations of Logic Programs.* Technical Report CS84-22, Weizmann Institute of Science, November 1984.

[Cohen, 1986] D. Cohen. Automatic compilation of logical specifications into efficient programs. In *Proceedings AAAI-87*, American Association for Artificial Intelligence, Seattle, WA, August 1986.

[Fikes *et al.*, 1972] R. Fikes, P. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972. Also in *Readings in Artificial Intelligence*, Webber, B. L. and Nilsson, N. J., (Eds.).

[Iba, 1985] G. Iba. Learning by discovering macros in problem-solving. In *Proceedings IJCAI-9*, International Joint Conferences on Artificial Intelligence, Los Angeles, CA, August 1985.

[Kahn, 1984] K. Kahn. Partial evaluation, programming methodology and artificial intelligence. *AI Magazine*, 5(1):53–57, Spring 1984.

[Kahn and Carlsson, 1984] K. Kahn and M. Carlsson. The compilation of prolog programs without the use of a prolog compiler. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, Japan, 1984.

[Kedar-Cabelli and McCarty, 1987] S. T. Kedar-Cabelli and L. T. McCarty. Explanation-based generalization as resolution theorem proving. In *Proceedings of the Fourth International Machine Learning Workshop*, Irvine, CA, June 1987.

[Korf, 1985] R. Korf. *Learning to Solve Problems by Searching for Macro-Operators.* Pitman, Marshfield, MA, 1985.

[Laird *et al.*, 1986a] J.E. Laird, P.S. Rosenbloom, and A. Newell. Overgeneralization during knowledge compilation in soar. In *Workshop on Knowledge Compilation*, Oregon State University, Corvallis, OR, September 1986.

[Laird *et al.*, 1986b] J. E. Laird, P. S. Rosenbloom, and A. Newell. Soar: the architecture of a general learning mechanism. *Machine Learning*, 1(1):11–46, 1986.

[Langley, 1983] P. Langley. Learning effective search heuristics. In *Proceedings IJCAI-8*, International Joint Conferences on Artificial Intelligence, Karlsruhe, West Germany, August 1983.

[Lenat *et al.*, 1979] D.B. Lenat, F. Hayes-Roth, and P. Klahr. Cognitive economy in artificial intelligence systems. In *Proceedings IJCAI-6*, International Joint Conferences on Artificial Intelligence, Tokyo, Japan, August 1979.

[Mahadevan, 1985] S. Mahadevan. Verification-based learning: a generalization strategy for inferring problem-decomposition methods. In *Proceedings IJCAI-9*, International Joint Conferences on Artificial Intelligence, Los Angeles, CA, August 1985.

[Minton, 1985] S. Minton. Selectively generalizing plans for problem-solving. In *Proceedings IJCAI-9*, International Joint Conferences on Artificial Intelligence, Los Angeles, CA, August 1985.

[Mitchell *et al.*, 1986] T. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-based generalization: a unifying view. *Machine Learning*, 1(1):47–80, 1986.

[Mitchell *et al.*, 1983] T. M. Mitchell, P. E. Utgoff, and R. B. Banerji. Learning by experimentation: acquiring and refining problem-solving heuristics. In *Machine Learning*, Tioga, Palo Alto, CA, 1983.

[Mostow and Cohen, 1985] J. Mostow and D. Cohen. Automating program speedup by deciding what to cache. In *Proceedings IJCAI-9*, International Joint Conferences on Artificial Intelligence, Los Angeles, CA, August 1985.