Operationality Criteria for Recursive Predicates

Stanley Letovsky
School of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213
letovsky@cs.cmu.edu

Abstract

Current explanation-based generalization (EBG) techniques can perform badly when the problem being solved involves recursion. Often an infinite series of learned concepts are generated that correspond to the expansion of recursive solutions over every finite depth. Previous attempts to address the problem, such as Shavlik's *generalization-to-N* EBG method, are overly reluctant to expand recursions; this reluctance can lead to inefficient rules. In this paper EBG is viewed as a program transformation technique on logic programs. Within that framework an improved operationality criterion for controlling the expansion of recursions is presented. This criterion prevents certain infinite and combinatorially explosive rule classes from being generated, yet permits expansion in some useful circumstances, allowing more efficient rules to be learned.

Introduction

Despite the promise of machine learning as a technique for making it easier to get knowledge into machines, the current status of EBG [Mitchell *et al.*, 1986; Mooney and Bennett, 1986] learning techniques is such that it is often much harder to write a program which an EBG-based learning system will transform into something reasonable than it is to write that "reasonable" program directly. It is not the case that EBG systems can take any, or even most, inefficient encodings of a task and turn out an efficient version; rather it is *sometimes* the case that there exists *some* encoding of a task for which an EBG system can do something reasonable. If EBG systems are to become a useful technology they must transcend this limitation. Recursions in the initial program or rule set supplied to an EBG system are a common cause of undesirable learned rules. For example Minton's PRODIGY system [Minton, 1988], operating in the blocks world domain, can learn rules for constructing towers 2 blocks high, 3 blocks, high, and so on. Etzioni [Etzioni, 1990] makes this same observation, but does not provide a solution. The problem of learning rules which are applicable to similar problems regardless of their size has been called the *generalization-to-N* problem. [Shavlik, 1989] has presented a modification of EBG that permits such rules to be learned. The change involves making the EBG algorithm more reluctant to unfold definitions. This paper presents a more general solution to the problem of controlling the unfolding of recursion, which was implemented in an EBG-engine called RECEBG.

The issue of recursion control extends beyond the problem of simply generalizing to N, however; the work described here was motivated by a different problem involving recursion and EBG. An important technique for specifying search control knowledge to a problem solver involves providing some sort of progress metric to be used for evaluation of intermediate states of problem solving. Heuristic search algorithms such as A* are based on this principle. Frequently such metrics work by counting some aspect of the state, which means they are recursively defined on the state representation. For example, in simplifying algebraic expressions, the metric of expression size can be used to impose directionality on the application of a set of undirected equality rules. When search control knowledge is provided in this manner, it is often possible to statically evaluate the effects of the available operators on the value of the metric in order to learn search control for the operators. For example, one could statically determine that the equality rule `X*1=X` reduces expression size when used as a rewrite rule in the left-to-right direction, but not in the reverse direction. If this fact can be established in general for all applications of the rule, then it is not necessary to evaluate the size of every expression before and after application of the rule in order to see if progress is being made – in effect, the evaluation of the metric is performed once 'at compile time' (or 'learning time') rather than repeatedly at run-time. [1] In order to perform such compile-time evaluation of recursive metrics, it is necessary to unfold recursions. In the example, one would

[1] Such precomputation of parts of a program at compile-time is known as partial evaluation of a program. Several authors have recently pointed out the close relationship between partial evaluation and EBG.[van Harmelen and Bundy, 1988]

need to count the expression sizes on the left and right hand sides of the rewrite rule. This in turn requires unfolding the definition of the *expression size* metric over the terms on each side of the equation. Since these terms contain variables, the unfolding of the metric is only partial, and the compile time value of the metric is not a number but an algebraic expression. The effects of the rule on the metric can nontheless be assessed by reasoning about alegebraic inequalities, such as whether `size(X)+2 > size(X)`, must be true.

In summary we have the observation that unfolding of recursions must be limited in order to prevent the generation of infinite rule sets, yet it must be permitted in order to allow static evaluation of recursive properties where possible. This paper presents a strategy for controlling the unfolding of recursions which attempts to satisfy both of these goals. We will see that there are limits on solving this problem in general – a strong version of it is equivalent to the halting problem – but that there are incomplete solutions of demonstrable practical utility. This paper first describes a model of EBG as transformation of logic programs, then presents the modifications to the operationality criterion of EBG that control unfolding. Finally we describe the performance of a series of EBG-like systems on a static metric evaluation learning problem, including SOAR, a simple PROLOG-EBG engine, a PROLOG partial evaluator, and the RECEBG system presented here, and show that only the last adequately handles the problem.

EBG as Transformation of PROLOG Programs

In this paper we view EBG as a technique for transforming logic programs [Prieditis and Mostow, 1987; Hirsh, 1987; van Harmelen and Bundy, 1988], specifically pure PROLOG programs. The advantages of this model of EBG are several. Pure PROLOG programs have a very simple structure that is convenient to analyze, yet they are expressively as or more powerful than any other formalism to which EBG techniques have been applied. A PROLOG program is simply an AND/OR tree, augmented with recursion and variable-binding machinery. This may be contrasted, for example, with the languages used by EBL-systems such as PRODIGY. PRODIGY specifies programs as sets of operators with first-order pre- and post-conditions, plus separate sublanguages for specifying domain theories and search control rules. The language is considerably more complex without being more expressively powerful. PROLOG has the additional advantage of being better known and more widely available, so that program transformation algorithms expressed relative to PROLOG can be more readily understood and tested by others.

The relative simplicity of PROLOG as a programming language carries over to program transformations on the language. EBG has a very simple description in terms of PROLOG programs [van Harmelen and Bundy, 1988]: it corresponds to repeated application of a program transformation I will call *expansion*, which means the replacement of a subgoal by one of its definitions.[2] Since PROLOG goals may have more than one than one definition, more than one such substitution is possible. Eg. given `p(X):-q(X)`, `q(X):-r(X)` and `q(X):-s(X)`, expansion of the definition of `q` within the definition of `p` generates two new definitions of `p`: `p(X):-r(X)` and `p(X):-s(X)`. Exhaustive application of this transformation converts nonrecursive PROLOG programs to disjunctive normal form. [3] Expansion of recursive programs can generate infinitely many expansions or simply fail to terminate.

Note that this model captures only certain aspects of EBG, chiefly learning from success. In order to capture other aspects, notably learning from failure, one must consider transformations that reorder conjuncts and rules (see eg. [Etzioni, 1990]).

The RECEBG Operationality Criterion

Within the model of EBG set forth in [Mitchell *et al.*, 1986] and [Keller, 1986] expansion is controlled by an *operationality criterion*, which determines which predicates may be replaced by their definitions during the creation of learned rules. The operationality criterion controls the level of generality of the learned rules: if it is reluctant to expand, the rules produced will be more general; if too eager, the rules will be more specialized. In order to permit possible nonexpansion of recursive definitions, the basic EBG algorithm must be modified to allow for the possibility that more than one learned rule will be produced from a single example. For example, suppose that a call to a recursively defined predicate called `subgoal` occurs somewhere in the middle of a computation of a predicate called `goal`. If the decision is made not to expand the occurrence of `subgoal`, then two rules may be learned: one for `goal`, containing an unexpanded reference to `subgoal`, and one for `subgoal`, which captures the specialized aspects of the example with respect to `subgoal`. [Shavlik, 1989] also describes this modification. Once the algorithm has been modified in this way to permit nonexpansion of recursions, the interesting question is when to expand.

The following example will be used to elucidate the operationality criterion:

[2] Expansion is the PROLOG analog of the λ-calculus operation of β-reduction. The term *unfolding* is also widely used for this transformation.

[3] Such exhaustive expansion in general causes an exponential blow-up in program size, which is certain to increase the time to fail a goal and very likely to increase the average time to succeed. This highlights the importance of the operationality criterion for controlling the application of this potentially dangerous transformation.

```
goal(X) :- ...,getlist(Y),member(X,Y),... .
member(A,[A|Rest]).
member(A,[B|Rest]) :- member(A,Rest).
```

Here a nonrecursive predicate, called **goal**, contains a reference to the recursively defined predicate **member**. The list argument to **member** is supplied by a second predicate called **getlist** whose definition is not shown. It is important to distinguish between references to a recursive predicate that appear outside of the predicate's definition, such as the call to **member** inside **goal**, and recursive calls occurring within a predicate's definition, such as the call to **member** in the second clause of **member**. We will refer to the former as *external calls*, and the latter as *internal calls*. The following sections explain the different components of the operationality criterion.

Rule#1: Never Expand Internal Calls

Expansion of internal calls can license an infinite number of learned rules, corresponding to the infinitely many possible finite recursion depths. In the PRODIGY blocks-world example, such expansion leads to rules for 2-towers, 3-towers, etc. In the case of **member**, it leads to additional clauses such as `member(A,[B,A|T]).`, `member(A,[B,C,A|T]).`, `member(A,[B,C,D,A|T]).`, etc. The assumption that the cost of matching such rules will remain small as their number grows without bound is questionable. Rule#1 says to never do this, that such expansions will not, in general, be a win over simply keeping the compact recursive definition around. Even if recursion depth is bounded, if a predicate has several recursive clauses in its definition, expansion of internal calls can generate explosively many learned rules: in fact, B^D learned rules, where B is the branching factor – i.e., number of disjuncts – and D is the depth of recursion. In the algebraic equation solving domain that was studied in this research, this effect led to learned rules that were formed by composing primitive algebraic rewrite rules so that the example equation could be solved in a single rule; such rules were highly specific and very numerous.

Rule#1, in conjunction with the algorithmic modifications that allow multiple learned rules from a single example, is equivalent to Shavlik's *generalization to N* technique.

Rule#2: Well-Founded Expansion of External Calls

Overly eager expansion of external calls can potentially reproduce all of the problems arising from expansion of internal calls, by creating new definitions of the caller instead of the recursive predicate. For example we could generate the following rules for **goal** by expanding the calls that led to the infinite set of rules for **member** shown above:

```
goal(X) :- ..., getlist([X|T]), ....,
goal(X) :- ..., getlist([A,X|T]), ....,     etc.
```

However, if expansion of **getlist** were to result in a binding of the list variable Y to a list such as `[a,b]`, as in `goal(X) :- ..., member(X,[a,b]), ....` then expanding the call to member would be a *good* thing, because the predicate could be satisfied at compile (or learning) time. (Two different rules could be generated by such an expansion, in which X would be bound in turn to **a** and to **b**.)

The problem is to find a way to license those expansions which can perform useful work at compile time, without opening the door to infinite rule sets in the caller. The solution used in RECEBG is to allow the expansion of an external call where that expansion can be shown to be making progress. The technique is similar to the approach used in [Boyer and Moore, 1979] theorem prover to show that inductive proofs are valid; the same idea is used in proving termination of programs. The key idea is that in order for the expansion to be safe, there must exist a *well-founded ordering* on the arguments to the recursive predicate which is decreasing over the expansion, and which has a smallest value. [4] Finding such an ordering in general is equivalent to being able to show whether any program will terminate; i.e., it is equivalent to the halting problem. However, in practice a few simple heuristics are capable of generating progress metrics for a broad class of common problems. For programs that manipulate list structure, a structural induction metric which counts the size of the (partially instantiated) list structure is usually sufficient to provide a well-founded ordering. Progress metrics for programs which count down to zero or count up to a fixed value can also be defined. Together these metrics are sufficient to prove termination for the class of bounded loop programs, i.e., those programs that can be written using only *for-loop*-type iteration constructs.

The structural induction metric in RECEBG counts the number of leaves in a term's expression tree, with variables and atoms counting as one. To determine whether an external call is well-founded, RECEBG compares the size of the arguments in the call with the size of the arguments in the recursive call, i.e., the recursive call in the body of the predicate after the head has been unified with the external call. This criterion can easily be improved, since a recursion is well founded if *any* argument (or subset of the arguments) is getting smaller, which may be true even when the size of all of the arguments together is remaining constant – eg., list **reverse**. Another simple extension would be handling of counter controlled loops, where $<$ on the natural numbers provides the well-founded ordering.

Rule#2 says that external calls can be expanded if a

[4] The *has a smallest value* restriction rules out infinite descending chains, such as might arise using orderings based on integers if negative numbers are allowed, or by asymptotically approaching a finite value using real-valued numbers.

well-founded ordering can be found on any of the arguments. For example, if expansion of `getlist` resulted in `Y` being bound to `[A,B,X]`, the second argument to the `member` call would be `[A,B,X]` before expansion, and `[B,X]` in the new call resulting from the expansion (assuming that `A` was not equal to `X` in the example). Since `[B,X]` has fewer list elements than `[A,B,X]` the expansion is making progress, and therefore can be carried out safely.

Note that an external call containing only variables, such as `member(X,Y)`, would never be expanded under this rule, while an call containing partially instantiated variables, such as `member(X,[Y,Z])`, will be expanded. In the former case, however, expansion of other predicates that share the variable `Y` could specialize it so that the call to `member` would become expandable. Therefore whenever a variable is specialized by expansion during the EBG process, any unexpanded recursive calls must be rechecked to see if they have become expandable. Note also that such variable specializations are determined by the code only, they are not supplied by the example. The example enters in only in the selection of control flow paths through the code. The result is that calls are expanded only to the extent warranted by the variable specializations in the learned rule, rather than the bindings in the example.

Rule#3: Forced Termination Expansion

The trickiest aspect of the expansion control concerns the conditions under which, when the example chose the terminal case in an external call, the learned rule should incorporate that choice rather than simply retaining the call to the recursive predicate. Eg., suppose that during an call of `member(X,Y)`, the corresponding goal in the example was `member(1,[1,2])`, which succeeds immediately. Should the call be expanded, eliminating the call to `member` and binding `Y` to `[X]` in the learned rule? Unwanted expansions yield overly specific learned rules that incorporate too much detail from the example, while missed opportunities to expand can lead to inefficient rules that fail to exploit the information provided by the example. The inefficiency can extend well beyond the predicate in question, since the failure to expand one predicate can cause variables to go unspecialized, leading to a failure to expand other predicates via Rule #2.

Examination of numerous examples suggests that sometimes expanding the terminal case is the right thing to do, while other times retaining the call leads to better rules. The feature that distinguishes those cases where expansion is desirable is that in those cases, *the terminal branch is the only logically possible choice*, at the level of the learned rule rather than the example. In other words, the recursive branch(es) can be statically shown to be contradictory, so that the recursive branch can never be taken. Rule#3 says that if this can be proved, then the recursive call should be replaced by its terminal case definition.

Proving the applicability of Rule#3 requires statically showing that the recursive definitions are inapplicable. The example cannot be appealed to to assist in this proof, because the proof must hold across all examples. It is not possible to provide a complete capability for detecting forced termination, short of providing a complete theorem prover. However, numerous simple cases can be recognized, which provide considerable leverage in practice. The easiest refutations are obtained when the call fails to unify with the heads of the recursive clauses, as occurs in the most common PROLOG recursion cliche, where `p([])` cannot match `p([H|T])`. If no recursive clause can unify, termination is forced. In addition, RECEBG contains a small set of simplification rules which can sometimes simplify conjuncts in the recursive clause to false after the head has been unified with external call, thus showing the inapplicability of the clause. These rules provide a poor-man's theorem prover; adding strong typing and static type-checking machinery would be a useful extension of this idea. Incompleteness in the reasoner's ability to refute the recursive branches causes Rule#3 to fail to apply in places where it ought to, resulting in inefficient, overly general learned rules. If one must err, it is better to err on the side of overly general rules, because there are usually fewer of them than overly specific rules, and the effect on performance is likely to be less bad.

Evaluating RECEBG

This section describes the outcome of attempts to solve a learning problem involving complex recursion unfoldings on four different EBG-like systems: the SOAR learning architecture, a simple EBG-engine for pure PROLOG, a partial evaluator for pure PROLOG, and finally the RECEBG system that incorporates the operationality criterion described above. We first describe the target problem.

The *Isolation* Example

PRESS [Bundy, 1983] is a program for solving algebraic equations. It is notable in that it provides a coherent theory of search control for the application of algebraic rewrite rules. In PRESS, the rewrite rules of algebra are grouped into sets called *strategies*, each of which helps achieve a different subgoal of the task of solving an equation. Here we will consider only one such strategy, called *Isolation*, which applies to equations that contain a single occurence of the unknown on the left hand side, and no occurrences on the right hand side. Isolation strips away away function applications from the left hand side until the unknown is isolated. At that point the equation is solved. Isolation rules work by applying the inverse of the function being eliminated to both sides of the equation. Eg.

```
A + B = C   ⇒   A = C - B
A * B = C   ⇒   A = C / B
sin(A) = B  ⇒   A = arcsin(B)
```

PRESS's search-control theory can be viewed as a sequence of loops, each of which applies operators until some metric of badness is reduced to zero. In the case of the isolation strategy, the metric is *depth of nesting of the unknown on the left-hand side.*

The problem of learning search control for an operator given a progress metric can be formulated as follows: form new "search-controlled" operators from the original uncontrolled operator by adding to its preconditions the weakest additional constraints that will guarantee that the operator application reduces the metric. For example, if the operator is the rewrite rule `A+B=C` $\Rightarrow$ `A=C-B` and the metric is depth of nesting of the unknown on the LHS, the additional precondition would be that the unknown is a subterm of `A`. To do this, the learner must statically evaluate the operators' effects on the metric's value. Below we describe attempts to implement this behavior in 4 different EBG-engines.

PRESS-SOAR

PRESS-SOAR is an attempt to learn PRESS-style search control within SOAR. SOAR [Laird *et al.*, 1987; Rosenbloom and Laird, 1986] is a problem-solver that incorporates an EBG-style learning component. The intent was to provide the system with algebraic operators – rewrite rules – together with a metric for assessing progress, and have it learn the conditions under which the operators make progress in reducing the metric. The metric used was the depth-of-nesting metric appropriate for isolation. In PRESS-SOAR problem solving begins in a problem space containing only rules with the desired search control knowledge; this space is initially empty. When no operator is applicable, an impasse occurs, invoking a *promiscuous-proposal* problem space. In this space any rewrite rule matching the expression can apply, but a check must be performed to verify that the rule application does in fact reduce depth of nesting before the rule will be selected. The goal was to have SOAR's chunking machinery chunk over these impasses to yield rewrite rules which are only considered in circumstances where they reduce depth of nesting.

The early versions of PRESS-SOAR which functioned correctly apart from chunking did not generate the desired chunks. Instead they generated radically overspecific chunks, which incorporated many of the features of the particular problem instances that SOAR had been exposed to, including such features as the entire left hand side term structure, or the particular nesting depth before and after rewriting. Developing a version of PRESS-SOAR which produced the desired chunks required careful attention to what information was computed before versus during the impasse. In general, to get desired chunks in SOAR, one must ensure that all the "compile time" computations occur within an impasse. In effect the impasse/multiple problem space mechanisms in SOAR function to allow the programmer to determine which parts of computation should occur "at compile time", thereby allowing the programmer to define an operationality criterion tailored to the problem at hand. The question of how to automatically determine operationality is thus closely related to the question of how to automatically organize knowledge into separate problem spaces.

PROEBG: A PROLOG EBG-Engine

PROEBG is a simple EBG-engine for pure PROLOG, closely modelled on the **ebg** program presented in [van Harmelen and Bundy, 1988]. It builds a new disjunct for a goal by finding the set of primitives evaluated in the course of solving an example. The results contain no calls to defined functions, and tend to contain many trivial wasteful steps – eg., goals that are obviously true by inspection, regardless of how the variables are bound – that are simplified in a postprocessing step.

```
% Apply rewrite rules that reduce the measure, until done.
isolate(X,X=Exp,X=Exp).
isolate(X,LHS=RHS,Solution) :-
      not(LHS=X),
      rule(LHS=RHS,NewLHS=NewRHS),
      don(X,LHS,Don),
      don(X,NewLHS,NewDon),
      NewDon < Don,
      isolate(X,NewLHS=NewRHS,Solution).

don(Var,Var,0).
don(Var,Trm,Don) :-
      Trm =..  [Op|Args],
      member(Sub,Args).
      don(Var,Sub,D),
      Don is D+1.

rule(A+B=C,A=C-B).     rule(A+B=C,B=C-A).
rule(A*B=C,A=C/B).     ...
```

PROEBG was applied to the PROLOG version of **isolate** shown above. From the standpoint of the isolation search-control learning task, PROEBG was a disappointment. Its problems were very similar to those of PRESS-SOAR: unwanted incorporation of specific details of the examples resulting in undergeneral chunks. A rule learned by PROEBG is essentially a composition of all the rewrite rules used in solving the example. For example, given the goal `isolate(x,3*x+2=y,Result)`, it would learn `isolate(X,A*X+B=C,X=(C-B)/A)`.

PROPE: A PROLOG Partial Evaluator

PROPE is a partial evaluator for pure PROLOG programs written by the author. To the simple **peval** code of described in [van Harmelen and Bundy, 1988], PROPE adds the full RECEBG operationality criterion, plus some algebraic simplification capabilities. Note however that a partial evaluator, unlike an EBG engine, performs its analysis statically, without benefit of an example.

The problem with PROPE was that, although it was able to generate the desired rules, it generated too many learned rules, many of which corresponded to computations that could never occur. For example, when expanding the rule `A+B=C` ⇒ `A=C-B` inside `isolate`, it was able to perform well-founded expansions of the `don` predicate over the equations `A+B=C` and `A=C-B`. The predicate `member` is expanded two different ways within the first call to `don` – one assumes that `X` is a subterm of `A`, the other, `B`. The second call to `don` assumes `X` is a subterm of `A`. Hence, one pair of assumptions will be impossible:

```
isolate(X,A+B=C,New) :- ...,
    don(X,B,I),
    don(X,A,J),
    I < J,... .
```

The problem is that PROEBG lacks some knowledge about these equations, namely, that there is only one occurence of `X` and that if it is a subterm of a rule variable before rewriting, it will be a subterm of that same variable after rewriting. Both domain knowledge and some reasoning ability are required to prove that the execution paths corresponding to these "impossible" rules contained contradictory assumptions. Here we see the EBG advantage over partial evaluation: by focussing attention on paths that are known to be realizable, EBG gets the equivalent of theorem-proving power and domain-knowledge from the world cheaply.

RECEBG

RECEBG incorporates the inproved operationality criterion within an example-guided framework. In addition to the modified operationality criterion described above, RECEBG incorporates a *Fortuitous Unification Rule*, which allows it to collapse 2 calls to the same predicate when those calls return identical results in an example. This rule allows the two calls to `don` in `isolate` to be reduced to a single call, after expansion has rendered their arguments identical. RECEBG also requires some post-EBG simplification of algebraic inequalities in learned rules.

RECEBG is able to produce the desired rules by expanding recursions at compile time as much as possible, while restricting its attention to logically possible control-flow paths because of the example guidance. For example, the rule learned to govern the isolation of sum-terms is:

```
isolate(X,A+B=C,New) :-
     don(X,A,I),
     isolate(X,A=C-B,New).
```

Note that the output `I` of `don` is no longer used; the call to `don` now functions only to ensure that `A` contains `X`. Similar rules are learned for the other operators.

Conclusions

Recursive predicates are not well-handled by current EBG systems. A simple-minded, overly eager operationality criterion can produce an infinite number of overly specific learned rules that incorporate too much example detail and provide very little coverage. An operationality criterion which is overly conservative about expanding recursions will avoid these infinities but be condemned to repeatedly computing at runtime results which could be computed at learning time. A partial evaluator which statically expands recursions as much as possible can generate large numbers of spurious learned rules due to lack of domain knowledge and/or theorem proving power. An EBG engine that opportunistically expands recursions at learning time was exhibited which avoids all of these pitfalls.

Acknowledgements

The author thanks Scott Dietzen, Oren Etzioni and Allen Newell for helpful comments on this manuscript.

References

Boyer, Robert S. and Moore, J. Strother 1979. *A Computational Logic.* Academic Press.

Bundy, Alan 1983. *The Computer Modelling of Mathematical Reasoning.* Academic Press.

Etzioni, Oren 1990. Why prodigy/ebl works. In *Proceedings of the Eighth National Conference on Artificial Intelligence.*

Hirsh, Haym 1987. Explanation-based generalization in a logic-programming environment. In *Proceedings of AAAI-87.* American Association for Artificial Intelligence.

Keller, Richard 1986. Defining operationality for explanation-based learning. In *Proceedings of AAAI-87.* American Association for Artificial Intelligence.

Laird, J.; Newell, A.; and Rosenbloom, P. 1987. SOAR: An architecture for general intelligence. *Artificial Intelligence* 33(1):1–64.

Minton, Steve 1988. *Learning Effective Search-Control Knowledge: An Explanation-Based Approach.* Ph.D. Dissertation, Carnegie-Mellon University.

Mitchell, T.; R.Keller, ; and S.Kedar-Cabelli, 1986. Explanation-based generalization: A unifying view. *Machine Learning* 1(1).

Mooney, Raymond and Bennett, Scott 1986. A domain independent explanation-based generalizer. In *Proceedings of AAAI-86.* American Association for Artificial Intelligence.

Prieditis, Armand and Mostow, Jack 1987. Prolearn: Towards a prolog interpreter that learns. In *Proceedings of AAAI-87.* American Association for Artificial Intelligence.

Rosenbloom, Paul and Laird, John 1986. Mapping Explanation-Based Generalization onto SOAR. In *Proceedings of AAAI-86.* American Association for Artificial Intelligence.

Shavlik, Jude 1989. Acquiring recursive and iterative concepts with explanation-based learning. Technical Report #876, University of Wisconsin, Madison.

van Harmelen, Frank and Bundy, Alan 1988. Explanation-Based Generalisation = Partial Evaluation. *AI Journal* 36:401–412.