EXPLANATION-BASED GENERALIZATION AS RESOLUTION THEOREM PROVING

Smadar T. Kedar-Cabelli
L. Thorne McCarty

ML-TR-10

Laboratory for Computer Science Research
Hill Center for the Mathematical Sciences
Rutgers University
New Brunswick, New Jersey 08903

This is a preprint of a paper to appear in the Proceedings of the Fourth International Machine Learning Workshop, University of California, Irvine, CA, June 1987.
Explanation-Based Generalization as Resolution Theorem Proving

SMADAR T. KEDAR-CABELLI  (KEDAR-CABELLI@RUTGERS.EDU)
L. THORNE MCCARTY  (MCCARTY@RUTGERS.EDU)

Department of Computer Science, Rutgers University, New Brunswick, NJ 08903 U.S.A

Abstract

This paper presents Explanation-Based Generalisation as an augmentation of resolution theorem proving for Horn Clause Logic. The corresponding implementation, PROLOG-EBG, performs generalization as a byproduct of standard PROLOG theorem proving. This results in very a concise (four-clause) implementation of EBG. The propagation of consistent variable bindings by PROLOG during theorem proving corrects an error in the previously published EBG algorithm.

1. Introduction and Motivation

Previously, EBG (Mitchell, Keller & Kedar-Cabelli, 1986) was introduced as an analytic method for solving the following Explanation-Based Generalization Problem:

**Given:**
- Target concept (describing the concept being learned);
- Training example (an example of the target concept);
- Domain theory (rules and facts about the domain);
- Operationality criterion (criterion for the form of the learned concept definition)

**Determine:**
- A generalisation of the training example that is a sufficient concept definition for the target concept and that satisfies the operationality criterion

EBG’s key power lies in its ability to form a generalization from a single training example by creating a proof that the example is a member of the target concept.

This paper presents Explanation-Based Generalization as an augmentation of SLD-resolution theorem proving for Horn Clause Logic (Sterling & Shapiro, 1986, Lloyd, 1984). Why is resolution a useful perspective? For one, it clarifies the EBG method, and the notions of proof and generalization. The EBG method becomes a slight perturbation of SLD-resolution, performing generalization as a byproduct of standard theorem proving. The construction of the generalized proof exactly mirrors that of the proof, except leaf nodes remain ununified. The proof demonstrates how the training example is a member of the target concept. The generalized proof characterizes those members of the target concept which have a proof of concept membership with the same structure as the training example (the same proof rules, applied in the same order).

There are several other motivations for adopting the perspective of Explanation-Based Generalization as SLD-resolution theorem proving. The theoretical underpinnings of SLD-resolution have been much investigated (Lloyd, 1984), and can be used to facilitate correctness and complexity analyses of EBG. SLD-resolution can also serve as a common perspective from which to compare EBG with related algorithms. Viewing EBG as resolution theorem proving has been useful in understanding and correcting an
error in the previously published EBG algorithm (Mitchell, Keller & Kedar-Cabelli, 1986). Finally, implementing EBG as an augmented PROLOG theorem prover results in a very concise and portable implementation.

The balance of the paper is devoted to describing the implementation, PROLOG-EBG. Section 2 describes PROLOG-EBG as a customized PROLOG meta-interpreter, and illustrates PROLOG-EBG by means of an example. Section 3 compares PROLOG-EBG to related algorithms, and describes the correction of an error in the previously published EBG algorithm. We conclude in section 4 with a summary of the key features of this approach, and a description of some open problems.

2. EBG as a PROLOG Meta-Interpreter

Modifying the PROLOG interpreter (theorem prover) to fit a particular application is often done by writing a PROLOG interpreter in PROLOG itself (a meta-interpreter), and then adding customizing features (Sterling & Shapiro, 1986). We now show how a simple PROLOG meta-interpreter is customized to perform EBG.

A simple PROLOG meta-interpreter for pure PROLOG (Sterling & Shapiro, 1986) is:

```prolog
/* PROVE(Goal) Goal is deducible from a pure PROLOG program */
prove(A) :- clause(A,true).
prove((A,B)) :- prove(A),prove(B).
prove(A) :- clause(A,B),prove(B).
```

'Prove' states that A is proved if it unifies directly with an assertion, 'clause(A,true)'. The conjunctive goal (A,B) is proved if A is proved, and then B is proved (goals are selected from left to right). Goal A is proved if it unifies with a clause 'A :- B', and B is proved.

There are two augmentations needed to transform the simple meta-interpreter into one that can do both proof and proof generalization, and generate the resulting proof trees.

Augmentation 1: Retain the Proof Tree. First, we modify the meta-interpreter to retain the proof tree as it solves a goal (also in (Sterling & Shapiro, 1986)). The augmented meta-interpreter becomes:

```prolog
/* PROVE(Goal,Proof) Proof is the proof tree for Goal, given a pure PROLOG program. */
prove(A,[A]) :- clause(A,true).
prove((A,B),Proof) :-
    prove(A,AProof),prove(B,BProof),
    append(AProof,BProof,Proof).
prove(A,Proof) :-
    clause(A,B),prove(B,BProof),
    append([A],[BProof],Proof).
```

'Prove' reads as follows: The proof tree of A is the list containing A itself, [A], if A is unified directly with an assertion. The proof tree of the conjunctive goal (A,B) appends lists representing the proof trees of each of the conjuncts. The proof tree of clause 'A :- B' is the list [A,BProof], where BProof is the proof tree of B.

Augmentation 2: Generalize the Proof Tree. The second augmentation to the meta-interpreter constructs a generalized proof tree. That proof tree is generalized in a
particular sense: it describes the class of all examples under which a proof of the same structure would succeed. We implement this by constructing two proof trees in parallel: the specific proof tree and the generalized proof tree. To create the generalized proof tree we follow the specific proof in parallel down to the assertions. When the specific proof has successfully terminated by unifying a goal with some ground assertion (fact), a copy of the ununified goal is retained in forming the generalized proof tree. Thus the generalized proof tree is created by propagating constraints on the variable bindings among the various proof rules (rule substitutions), but dropping any constraints introduced by the specific example (fact substitutions).

The resulting meta-interpreter is PROLOG-EBG:

```prolog
/* PROLOG-EBG(Goal,GenGoal,Proof,GenProof) Proof is the proof tree of Goal; GenProof is the generalized proof tree of GenGoal, (a possibly generalized form of Goal), given a pure PROLOG program*/

prolog_egb(A,GenA,[A],[GenA]) :- clause(A,true).
prolog_egb((A,B),(GenA,GenB),Proof,GenProof) :-
    prolog_egb(A,GenA,AProof,GenAProof),
    prolog_egb(B,GenB,BProof,GenBProof),
    append(AProof,BProof,Proof),
    append(GenAProof,GenBProof,GenProof).
prolog_egb(A,GenA,[Proof],[GenProof]) :-
    clause(GenA,GenB),copy((GenA:-GenB),(A:-B)),
    prolog_egb(B,GenB,BProof,GenBProof),
    append([A],[BProof],Proof),
    append([GenA],[GenBProof],GenProof).
```

The meta-interpreter has now been augmented to create two proof trees in parallel. The construction of the two proof trees is identical except when goal A unifies directly with an assertion (the first clause). The specific proof tree becomes the unified goal [A], while the the generalized proof tree of the corresponding generalized goal GenA, remains [GenA], ununified. The role of 'Copy'((GenA:-GenB),(A:-B))' in the third clause is to ensure that the generalized tree exactly mirrors the specific tree by having 'A :- B' and 'GenA :- GenB' be exact copies (with new variables).

```
copy(Old,New) :- assert('Smaker'(Old)), retract('Smaker'(New)).
```

(Note that given a goal A, PROLOG-EBG has the flexibility to take as input a more general goal, GenA (where constants, or constrained variable bindings may be further variabилиzed), so a more generalized proof tree can be created.)

**Example:** We now recast the Explanation-Based Generalization Problem in terms of inputs and outputs to PROLOG-EBG. We then illustrate how PROLOG-EBG implements the EBG method by means of the 'suicide' example from (DeJong & Mooney, 1986). The inputs to EBG are: The target concept, represented in PROLOG-EBG as a goal ':-kill(john,john)', and a (possibly) generalized target concept as goal ':-kill(X,Y)'; the domain theory, represented as a set of clauses in the PROLOG program:

1We are grateful to Armand Priedišis for pointing out an error in our home-grown version of 'copy', and to William Cohen and Neeraj Bhatnagar for suggesting that 'copy' from (Sterling & Shapiro, 1986, pg.180) be used instead.
kill(A,B) :- hate(A,B), possess(A,C), weapon(C).
hate(W,W) :- depressed(W).
possess(U,V) :- buy(U,V).
weapon(Z) :- gun(Z).

and the training example, represented by a set of ground assertions in the program:

depressed(john).
buy(john,gun1).
gun(gun1).

The EBG method creates a proof that the training example is a member of the target concept using the domain theory, and generalizes to an operational definition of the target concept. PROLOG-EBG implements that by finding a successful SLD-resolution proof of the goal from the rules and ground assertions in the PROLOG program. In parallel, PROLOG-EBG generalizes this proof to characterize the class of all examples that have the same proof of concept membership. In an optional post-processing phase, PROLOG-EBG extracts an operational definition of the target concept from the generalized proof tree.

Given the query:

:- prolog_egb(kill(john,john),kill(X,Y),Proof,GenProof).

PROLOG-EBG begins by searching for a clause in the database whose consequent unifies with 'kill(john,john)'; the clause

\[
\text{kill}(A,B) :- \text{hate}(A,B), \text{possess}(B,C), \text{weapon}(C).
\]

is found, along with the unifying substitution \(\theta = \{\text{john}/A, \text{john}/B\}\). The generalized goal 'kill(X,Y)' unifies with the same clause, with substitution \(\sigma = \{X/A, Y/B\}\). PROLOG-EBG now sets out to prove the rest of the goals, and generalized goals (with the substitution applied to them):

\[
\text{hate}(\text{john}, \text{john}), \text{possess}(\text{john}, C), \text{weapon}(C).
\]

\[
\text{hate}(X, Y), \text{possess}(X, C), \text{weapon}(C).
\]

To illustrate where the formation of the generalized proof diverges from that of the specific proof, consider what happens when the goal 'buy(john, C)' is proved. In the formation of the specific proof, the first clause of PROLOG-EBG is invoked, and 'buy(john, C)' is unified directly with an assertion in the database 'buy(john, gun1)', with mgu \(\theta = \{\text{gun1}/C\}\). The resulting leaf of the specific proof tree becomes the list \([\text{buy}(\text{john}, \text{gun1})]\). The corresponding leaf of the generalized proof tree, however, remains \([\text{buy}(X, C)]\), ununified.

PROLOG-EBG outputs the proof tree and generalized proof tree:

\[
\text{Proof:} \\
[kill(\text{john}, \text{john}), \\
[\text{hate}(\text{john}, \text{john}), \\
[\text{depressed}(\text{john})]], \\
[\text{possess}(\text{john}, \text{gun1})], \\
[\text{buy}(\text{john}, \text{gun1})], \\
[\text{weapon}(\text{gun1})]]]
\]

\[
\text{Generalized Proof:} \\
[kill(X, X), \\
[\text{hate}(X, X), \\
[\text{depressed}(X)]], \\
[\text{possess}(X, C)], \\
[\text{buy}(X, C)], \\
[\text{weapon}(C)], \\
[\text{gun}(C)]]]
\]

This is followed (optionally) by a post-processing phase which extracts an operational
definition of the target concept. The definition is extracted from the generalized proof
tree, using the operationality criterion. Currently, a node in the generalized proof tree
is tested for operationality by testing whether a copy of the node corresponds to an
assertion or a built-in function, with the query `:- operational(CopyP)`.

\[
\text{operational}(P) :- \text{clause}(P, \text{true}); \text{built-in}(P).
\]

('Built-in' is defined in terms of low-level system predicates for our particular version of
PROLOG.) 'Operational' roughly corresponds to the notion of operationality from
(Mitchell, Keller & Kedar-Cabelli, 1986) as "...predicates used in describing the training
example...or...a selected set of easily evaluable predicates from the domain theory...".
The post-processing phase finds a cutset of operational nodes in the generalized proof
tree. The conjunction of those predicates, along with the root of the generalized proof
tree, become the desired target concept and its operational definition:

\[
\text{kill}(X,X) :- \text{depressed}(X), \text{buy}(X, C), \text{gun}(C).
\]

3. Related Work

We have shown how two augmentations to a PROLOG meta-interpreter result in
PROLOG-EBG. The generalization in PROLOG-EBG is formed by propagating rule
substitutions but ignoring fact substitutions when creating the generalized proof tree.

**EGGS**: (Mooney & Bennett, 1986) presents a domain-independent EBG algorithm,
EGGS. We claim informally that EGGS and PROLOG-EBG are equivalent. The
algorithms differ only in the method by which the specific and generalized proof trees
are created. EGGS maintains an explanation structure, a SPECIFIC list for
substitutions of goals with rules and facts, and the GENERAL list for substitutions
with rules only. These lists are applied to the explanation structure at the end of
EGGS to create the specific and generalized proof trees. PROLOG-EBG creates the
two proof trees directly during the proof process, by applying both rule and fact
substitutions to form the specific tree, and only rule substitutions to form the
generalized tree. EGGS could be performed in PROLOG by customizing the meta-
interpreter to perform unification explicitly, rather than performing internal PROLOG
unification (see (Sterling & Shapiro, 1986) for a simple unification algorithm in
PROLOG). The advantage of maintaining substitution lists explicitly is the flexibility
in forming the operational definition at nodes other than the leaves of the tree. The
explicit substitution list is required since unifications have to be retracted for
generalizations formed from nodes at levels other than the leaves (DeJong & Mooney,
1986).

**MRS-EBG**: (Hirsh, 1987) presents an implementation of EBG in the MRS logic
programming environment. Like PROLOG-EBG and EGGS, the algorithm performs
explanation with generalization in a single pass. It differs in that operationality is
determined simultaneously with the proof process, unlike the post-processing phase in
both EGGS and PROLOG-EBG. If a node is operational, but is not a leaf node of
the explanation, the algorithm makes the operational node a terminal node in the
generalized tree, but continues the proof of the specific node until a leaf node is
reached. (This is to ensure that the generalization is based on a proof that in fact
succeeds.) PROLOG-EBG could be augmented to interleave operationality testing with
proof by checking each goal for operationality and not invoking the proof of the
generalized goal if it is deemed operational. The following code (due to William
Cohen) would have this effect:
/* PROLOG-EBG( Goal, GenGoal, OperDefn ) OperDefn is the operational defn for generalised goal GenGoal, given a pure PROLOG program */

prolog_egb(true,true,[]).
prolog_egb(A,GenA,[GenA]) :- operational(A),!;A.
prolog_egb(A,B,[GenA,GenB,G]) :-
    prolog_egb(A,GenA,GA),
    prolog_egb(B,GenB,GB),
    union(GA,GB,G).
prolog_egb(A,GenA,G) :-
    clause(GenA,GenB),
    copy([GenA:-GenB],A:-B),
    prolog_egb(B,GenB,G).

In this case, the operational definition is returned rather than the generalized proof tree. The advantage of creating a generalized proof tree rather than just a definition, however, is that the proof can be replayed to classify new examples by analogy, as in (Kedar-Cabelli, 1987).

Hirsh's algorithm can also handle a dynamic notion of operantality, i.e., a notion of operantality dependent on the proof progress. For example, a rule can specify that "P is operational if it has been proven N times", for example. Although PROLOG-EBG currently does not exploit that idea, the meta-interpreter could be augmented to monitor proof progress, and clauses for operantality could be used to check proof progress.

Correction of an Error in the Previous EBG Algorithm: PROLOG-EBG corrects an error in the previous EBG algorithm (Mitchell, Keller & Kedar-Cabelli, 1986), pointed out by DeJong and Mooney in (DeJong & Mooney, 1986) (and corrected first by EGGS). The generalization step in the previously published algorithm regressed the target concept through the explanation structure. The purpose was to propagate constraints on the variable bindings from the target concept and the rules of the proof to the leaves of the tree. In the 'suicide' example, regressing the target concept resulted in the following incorrect concept definition:

\[
\text{kill}(X,Y) :- \text{depressed}(Y), \text{buy}(Y,O), \text{gun}(O)
\]

which states that for all X and Y, X will kill any Y that is depressed and buys a gun! The problem is that constraints on the target concept imposed by the rules of the proof must also be taken into account. When hate(X,Y) is unified with the rule 'hate(W,W) :- depressed(W)', X and Y are constrained to be equal. The target concept is thus constrained to be 'kill(Y,Y)', that is, the proof applies only to suicides. The problem, more generally, is that any node in the proof could introduce constraints on the variable bindings for any other node in the proof. Propagation of constraints from each node to every other node is therefore needed to fully correct the problem. PROLOG-EBG correctly generalizes the proof tree since the constraints on variable bindings are collected throughout the SLD-resolution proof, and are propagated throughout the entire proof tree.

4. Conclusions
We have presented Explanation-Based Generalization as an augmentation of resolution theorem proving. PROLOG-EBG has been presented as the corresponding implementation of EBG in PROLOG. The conciseness of PROLOG-EBG results from customizing the PROLOG meta-interpreter to perform proof generalization in parallel
with proof. PROLOG-EBG was implemented on a SUN workstation, under Quintus Prolog, Release 1.6. We have described its equivalence to EGGS, and its relation to another EBG implementation in a logic programming environment. We have also discussed how the propagation of consistent variable bindings during theorem proving corrects an error in the previously published EBG algorithm.

PROLOG-EBG has been tested on the 'cup' and 'safe-to-stack' examples (Mitchell, Keller & Kedar-Cabelli, 1986), and the 'suicide' example (DeJong & Mooney, 1986). As EBG problems and the domain theory scale up, PROLOG-EBG will no doubt need augmentation and modification. For example, in many applications PROLOG-EBG would have to be extended beyond the Horn Clause subset of first-order logic. See (McCarty, 1986) for an example of a logic programming language which includes negation, embedded implication, and embedded universal quantification. See (Sims, 1987) for an implementation of EBG in an extended logic programming language. On the theoretical side, viewing EBG as an augmented SLD-resolution theorem prover can enable a more formal comparison with related algorithms, and facilitate theoretical analysis of complexity and correctness. See (Minton, 1987) for one approach to formalization.

Acknowledgments
We would like to thank Prasad Tadepalli, whose ideas had a significant impact on this work. This research benefited greatly from discussions with Neeraj Bhatnagar, Jan Chomicki, William Cohen, Haym Hirsh, Rich Keller, Sridhar Mahadevan, Steve Minton, Tom Mitchell, Ray Mooney, Jack Mostow, Armand Prieditis, Mike Sims, and Chris Tong. Thanks also go to many of those mentioned above, and Jack Mostow in particular, for invaluable comments on earlier drafts of this paper. This material is based on research supported by the National Science Foundation under Grant No. DCR-83-51523-02.

References