Extending EBG to Term-Rewriting Systems

Philip Laird  
(LAIRD@PLUTO.ARC.NASA.GOV)  
AI Research Branch

Evan Gamble  
(GAMBLE@PLUTO.ARC.NASA.GOV)  
Sterling Federal Systems, Inc.

NASA Ames Research Center  
Moffett Field, CA 94035

Abstract

We show that the familiar explanation-based generalization (EBG) procedure is applicable to a large family of programming languages, including three families of importance to AI: logic programming (such as Prolog); lambda calculus (such as LISP); and combinator languages (such as FP). The main application of this result is to extend the algorithm to domains for which predicate calculus is a poor representation. In addition, many issues in analytical learning become clearer and easier to reason about.

Introduction

Analytical learning, including the various methods collectively known as explanation-based learning (EBL), is motivated by the observation that much of human learning derives from studying a very small set of examples ("explanations") in the context of a large knowledge store. EBL algorithms may be partitioned into those that use explanatory examples to modify a deficient theory and those that rework a complete and correct theory into a more useful form. Among the latter are algorithms, such as the familiar EBG algorithm (Mitchell et al. 86, Kedar & McCarty '87) that learn from success, and other algorithms (e.g., (Minton 88, Mostow & Bhatnagar 87) ) that learn from failure. The EBG algorithm is the focus of this paper.

The EBG algorithm changes certain constants in the explanation to variables in such a way that similar instances may then be solved in one step without having to repeat the search for a solution. For example, given this simple logic program for integer addition:

\[
\begin{align*}
\text{plus}(0, x_1, x_1) & : - \text{true.} \\
\text{plus}(s(x_2), x_3, s(x_4)) & : - \text{plus}(x_2, x_3, x_4).
\end{align*}
\]

and the instance \(\text{plus}(s(0), 0, s(0))\), the EBG algorithm finds the new rule, \(\text{plus}(s(0), x, s(x)) : - \text{true}\) by analyzing the proof and changing certain occurrences of the constant 0 to a variable \(x\). Subsequently, the new instance \(\text{plus}(s(0), s(0), s(s(0)))\) can be solved in one step using this new rule, instead of the two steps required by the original program, provided the program can decide quickly that this new rule is the appropriate one for solving this new instance.

The results from applying this technique have been a bit disappointing. Among the reasons identified in the literature are the following:

- The generalizations tend to be rather weak. Indeed, the longer the proof—and thus the more information in the example—the fewer new examples are covered by the generalization.

- Many reasonable generalizations (such as the rule \(\text{phs}(z, s(0), s(z)) : - \text{true}\) in the above example) are not available using this method alone.

- Over time, as more rules are derived, simple schemes for incorporating these rules into the program eventually degrade the performance of the program, instead of improving it. The program spends most of its time finding the appropriate rule.

Other issues also need to be raised. While EBG is often described as a "domain-independent technique for generalizing explanations" (Mooney, 88), it is not a language-independent technique. Virtually all variants of the algorithm depend on a first-order logical language, in which terms can be replaced by variables to obtain a more general rule. Even when the algorithm is coded in, say, Lisp, one typically represents the rules in predicate calculus and simulates a first-order theorem prover. Yet, on occasion, researchers (e.g., Mitchell, Utgoff, & Banerji 83, Mooney 88) have encountered domains where predicate calculus is at best an awkward representation for the essential domain properties. In these situations the ability to use another language and still be able to apply analytical learning algorithms would be highly desirable.

One may ask whether EBG is just a syntactical trick that depends on logic for its existence. If so, its status...
as a bona fide learning method is dubious: important learning phenomena ought not to depend upon a particular programming language. If EBG is not dependent on logic, then how do we port EBG directly to other languages? For example, in a functional language the plus program might be coded:

\[
\text{plus } x \ y := \begin{cases} 
  \text{if } x = 0 \text{ then } y \\
  s(\text{plus } z y), \text{ where } z = s(z).
\end{cases}
\]

Given the input \(\text{plus } s(0) 0\), this program computes \(s(0)\) as output. Surely an EBG algorithm for this language should be able to generalize this example such that the input \(\text{plus } s(0) y\) produces \(s(y)\), without having to revert to a logical representation.

Also, while the formal foundations of EBL have been studied (e.g., (Greiner 89, Natarajan 88, Natarajan 89, Cohen 89)), most of this work has abstracted away the generalization process in order to model the benefits of path compression. (See, however, (Bhatnagar 88) and (Dietzen & Pfenning 89).) This is reasonable, since one might assume that the basic EBG algorithm is well understood by now. But such is not the case: presentations of the algorithm in the literature have generally been informal, and occasionally inaccurate. The elegant PROLOG-EBG algorithm (Kedar & McCarty 87) is a case in point; in certain cases it will overgeneralize. (As an example, given the instance \(\text{plus } (0, 0, 0)\) and the \(\text{plus}\) program above, it produces the overgeneralization \(\text{plus}(x, y, z) :- \text{true.}\) In the past two years several papers, a thesis, and even a textbook have reproduced this algorithm without noticing or correcting the problem.

This paper addresses two issues:

- **Language**: We show that the EBG algorithm is a special case of an algorithm that we call AI-1. We present this algorithm formally in a framework based on term-rewriting systems (TRS), a formalism that includes, as special cases, logic programming, lambda calculus, applicative languages, and other languages, showing that EBG is more than a logic programming hack.

- **Correctness**: In this formalism, the correctness, power, and limitations of the algorithm can be carefully studied. Proofs then apply immediately to each of the languages mentioned above.

In addition, many difficult issues (operationality, utility, etc.) become clearer because the TRS formalism separates the generalization aspects of the problem from other, language dependent, issues.

A more complete presentation of the ideas in this paper can be found in a report (Laird & Gamble 90) available from the authors.

### Figure 1: Grammar for a logic-programming language.

**Typed-Term Languages**

We first define a family of typed languages. Like many programming languages, the syntax is described by a context-free grammar. We constrain the form of these grammars so as to include certain features, mainly types and variables, that allow us to generalize expressions.

A **typed-term grammar (ttg)** is an unambiguous context-free grammar with the following characteristics:

- The set of nonterminal symbols is divided into two subsets: a set of **general types**, denoted \(\{G_0, G_1, \ldots\}\), and a set of **special types**, denoted \(\{S_1, S_2, \ldots\}\). \(G_0\) is the “start” symbol for the grammar.

- The set of terminal symbols is likewise divided into subsets. There is a finite set of **constants**, \(\{c_1, c_2, \ldots, c_p\}\), and for each general type \(G^i\) there is a countable set of **variables**, denoted \(\{x_1^i, x_2^i, \ldots\}\).

- The set of productions satisfies three conditions: (1) For any nonterminal \(N\), the set of sentences generated by \(N\) is nonempty and does not contain the empty string. (2) For any nonterminal \(N\), the right-hand sides of all its productions \((N \rightarrow \alpha_1 \ldots \alpha_n)\) have the same length \(k_N\). For general types this length is one. (3) For each general type \(G^i\) and each variable \(x_j^i\) of that type, there is a production \(G^i \rightarrow x_j^i\). No other productions contain variables.

The ttg is used to define, not the programs in the language, but the **states of the computational model** to which that language applies. In the case of logic programs, the states are sets of goals to be satisfied. For lambda calculus, states are lambda abstractions and applications to be reduced.

**Example 1**: The productions in Figure 1 generate the class of goals for the logic-programming language used in the *plus* example above. Instead of \(G^i\) and \(S^i\) we choose more mnemonic names for the nonterminals. \(\text{Goal}\) and \(\text{Term}\) are general types whose respective variables are \(y_i\) and \(x_i\). \(\text{Formula, Conjunction},\) and \(\text{Term1}\) are special types. \(\text{Goal}\) is the start symbol. The constant symbols are \(\text{true, plus, s, 0, \&, comma, and two parentheses.}\) Examples of goals generated by this language are
A conjunctive goal (with two or more subgoals) is represented in this language by a
\( pZu+(O), x17,0) \) and \( 911. \) A conjunctive goal (with two

Example 2. The productions in Figure 2 generate


\[
\begin{align*}
\text{Expression} & \rightarrow \text{Expression1} \\
\text{Expression} & \rightarrow \text{Lambda-param} \\
\text{Expression} & \rightarrow \text{plus} | \text{succ} | \text{zero} | \text{second} \\
\text{Expression} & \rightarrow x_i \quad (\text{for } i \geq 1) \\
\text{Expression1} & \rightarrow \lambda \text{ Lambda-param. Expression} \\
\text{Expression1} & \rightarrow (\text{Expression} \text{ Expression}) \\
\text{Lambda-param} & \rightarrow v_i \quad (\text{for } i \geq 1)
\end{align*}
\]

Figure 2: Grammar for a lambda-calculus language.

Example 3. In the grammar of Example 1, \( \gamma = \text{plus}(x_1, 0, x_1) \) is a sentence of type \( \text{Goal} \) and type

\[ \text{Formula}. \] When we apply the substitution \( \theta = \{s(0)/x_1\} \),
the result is \( \text{plus}(s(0), 0, s(0)) \). The subterm 0 occurs at

location \( \omega = 0 \cdot 0 \cdot 4 \) in \( \gamma \) with type \( \text{Term} \). The replacement

\( \gamma[0 \cdot 0 \cdot 4 \leftarrow s(x_2)] \) gives \( \text{plus}(x_1, s(x_2), x_1) \).

Other properties of first-order terms extend to ttg languages. Terms can be ordered by subsumption: \( \gamma_1 \sqsubseteq \gamma_2 \) if there is a substitution \( \theta \) such that \( \theta(\gamma_1) = \gamma_2 \). If we treat variants, that is, terms that differ only by renaming

variables, as equivalent, subsumption is a partial order. \( \theta \) is a unifier for terms \( \gamma_1 \) and \( \gamma_2 \) if \( \theta(\gamma_1) = \theta(\gamma_2) \). One can readily extend the first-order unification algorithm to compute the most general unifier \( \theta = \text{mgu}(\gamma_1, \gamma_2) \) of two
ttg terms, if they are unifiable. It can be shown that, for
each general type \( G^i \), if we regard variants as equivalent,
the set of sentences of type \( G^i \) with the addition of a distin-
tguished least element \( \bot \) is a complete lattice partially
ordered by \( \sqsubseteq \). This property allows us to generalize and
specialize sentences.

Nondeterministic Term-Rewriting Systems

As a computational model of typed-term languages, we
adopt a class of nondeterministic term-rewriting systems.
TRS’s are an active research area of theoretical computer
science (Avenhaus & Madlener 90) and have already been
applied to machine learning (e.g., (Kodratoff 88, Laird
88)). Mooney (Mooney 89) has applied them specifically
to analytical learning as an alternative to predicate logic.
Using a TRS we are able to express our learning algo-
rithms in a form applicable to many formal systems, like
logic programming and lambda calculus.

The sentences generated by a ttg are interpreted as
states of a computation. States are transformed by
rewriting steps in which one of a fixed set of rewrite rules
is chosen and used to modify a subterm of the state. Non-
determinism enters in two ways: in the choice of the rule
and in the choice of the subterm to be rewritten.

A rewrite rule \( \alpha \Rightarrow \beta \) is a pair of terms \( (\alpha \) and \( \beta) \)
of the same type \( T \). The set of rules is closed under
substitution: if \( \alpha \Rightarrow \beta \) is a rule and \( \theta \) is a substitution,
then \( \theta(\alpha) \Rightarrow \theta(\beta) \) is a rule. A rewriting step is carried
out as follows. Let \( \gamma \) be a state and let \( \omega \) be a location
such that \( \gamma[\omega] = \alpha \) i.e., \( \gamma[\omega] \) is the string \( \alpha \) and has
type \( T \). Then \( \gamma \) can be rewritten to \( \gamma' = \gamma[\omega \leftarrow \beta] \). The
notation \( \gamma \Rightarrow^* \gamma_2 \) indicates that a sequence of zero or
more steps transforms \( \gamma_1 \) into \( \gamma_2 \).

A computation is a finite sequence of state-location-rule
triples,

\[ [\gamma_1, \omega_1, \alpha_1 \Rightarrow \beta_1], \ldots, [\gamma_n, \omega_n, \alpha_n \Rightarrow \beta_n], [\gamma_{n+1}, *, *] \quad (1) \]
where, for \(1 \leq i \leq n\), a substitution instance of the rule \(\alpha_i \Rightarrow \beta_i\) applied to \(\gamma_i\) at location \(\omega_i\) yields \(\gamma_{i+1}\). (+ indicates “don’t care”.) The path of the computation consists of just the locations and the rules (omitting the states).

Example 4. In a logic program clauses serve as the rewrite rules, and the state is a (single or conjunctive) goal. For example, the rule \(\text{plus}(0, x, x) := \text{false}\) says that any instance of the term \(\text{plus}(0, s(a), s(a))\) can be replaced by the term \(\text{true}\). When applied to the state \((\text{plus}(0, s(0), s(0)))\) at the underlined position, the result is \((\text{false})\). Using the \(\text{plus}\) program to continue this computation for two more steps gives \((\text{false})\), at which point no more rules apply.

Example 5. Consider a lambda-calculus language with explicit recursion (like Lisp, but unlike pure lambda calculus, which uses the \(Y\) combinator) in which there are explicit recursion (like Lisp, but unlike pure lambda calculus, which uses the \(Y\) combinator). The \(\text{AL-1}\) algorithm (Figure 3) takes an example computation and determines the most general rewrite rule that accomplishes in one step the same sequence of rewrites. The algorithm starts with a variable \(Z\) initialized to \(x_1^0\), a fresh variable representing the most general state. Within the loop the same sequence of reductions is applied to \(Z\), at the same locations, as in the original computation. (Recall that \(\gamma_i\) denotes the state before the \(i\)'th rule is applied at position \(\omega_i\).)

A problem arises if there is no subterm at the required location \(\omega_i\) in the generalization \(Z\). The procedure \(\text{Stretch}\) is called so that, if necessary, \(Z\) acquires a subterm at \(\omega_i\) (see next paragraph). The rule \(\alpha_i \Rightarrow \beta_i\) is then applied to the subterm at this location. The for loop of the algorithm accumulates in the variable \(\theta\) all substitutions applied to \(Z\) during stretching and rewriting. These substitutions are the weakest conditions that the variables must satisfy in order for the computation to follow the path of the example. \(\theta(x_1^0)\) is then the most general sentence to which the path can be applied, and the final value of \(Z\) is the result of all these rewrites. The final output is the new rule: \(\theta(x_1^0) \Rightarrow Z\).

Suppose now that the state \(Z\) lacks a subterm at location \(\omega_i\). \(\text{Stretch}\) (Fig. 4) determines the most general state \(Z'\) such that \(Z \supseteq Z' \supseteq \gamma_i\) and \(Z'\) has a subterm at \(\omega_i\). For example, if the generalized state \(Z\) is just the variable \(g_0^a\), the example \(\gamma_i\) is \((\text{plus}(0, s(0), s(0)))\), and \(\omega_i\) is the location of the underlined subterm, then \(\text{Stretch}\) would specialize \(Z\) to \(Z' = (\text{plus}(0, s(0), s(0)))\).

The main property of the \(\text{AL-1}\) algorithm is given by the following theorem:

**Theorem.** Let \(\gamma_1\) be the initial state of the input computation, and let \(\alpha \Rightarrow \beta\) be the rule output by the \(\text{AL-1}\) algorithm. Then \(\alpha\) is the most general state \(\supseteq \gamma_1\) such that the sequence of rules in the input computation is
Input: An n-step computation with \( \gamma_i \) the \( i \)th state and \( \alpha_i \rightarrow \beta_i \) the \( i \)th rule.

Procedure:

1. Initialize: \( \theta = \text{the empty substitution. } Z = z_0^i \), a fresh variable of type \( G^o \).

2. For \( i = 1 \) up to \( n \):
   
   2.1 \( Z' := \text{Stretch}(Z, \gamma_i, \omega_i) \). /* If necessary, extend \( Z \) so that location \( \omega_i \) exists. */
   
   2.2 \( \theta_1 := \text{unify}(Z', Z) \). /* Unifier for unstretched and stretched \( Z \) */
   
   2.3 \( \theta_2 := \text{unify}(Z'[\omega_i] \alpha_i) \). /* Unifier for \( \alpha_i \) and the subterm at location \( \omega_i \) */
   
   2.4 \( Z' := \theta_2(Z' \gamma_i \alpha_i) \). /* Apply an instance of the rule to \( Z' \) at \( \omega_i \) */
   
   2.5 \( \theta := \theta_2 \circ \theta_1 \circ \theta \). /* Accumulate the substitutions applied to \( Z \) */
   
   2.6 \( Z := Z' \). /* Update \( Z \) for the next iteration. */

3. Output the new rule: \( \theta(x_0^i) \Rightarrow Z \).

Figure 3: The AL-1 Algorithm.

Example 6. With the \texttt{plus} program in Example 5 ((\texttt{plus} \( [s, 0] \) \( 0 \)) rewrites to \( [s, 0] \) in about twenty steps. If this computation is given to AL-1 as input, the result is the new rule: ((\texttt{plus} \( [s, 0] \) \( x \)) \( \Rightarrow [s, x] \)). Space does not allow us to follow the entire process, but we can trace the first few steps.

In the first step of the computation, \texttt{plus} is replaced by its lambda definition:

\[
((\texttt{plus} \ [s, 0] \ 0)) \Rightarrow ((\lambda v_1 . \lambda v_2 . (((\texttt{zero}\ ? v_1) v_2) Q[v_1, v_2]) [s, 0]) 0),
\]

where \( Q[v_1, v_2] \) stands for the subterm \( \texttt{succ} \ldots \). In AL-1, \( Z \) is initially \( x_1 \); and since this expression has no term at the location (underlined) of \texttt{plus} in the example, \texttt{Stretch} is called to specialize \( Z \), with the result:

\[
Z' := ((x_2 x_3) x_4).
\]

Now we apply the \texttt{plus} rule to \( x_2 \), with the result:

\[
Z' := ((\lambda v_1 . \lambda v_2 . (((\texttt{zero}\ ? v_1) v_2) Q[v_1, v_2]) x_3) x_4).
\]

For this first pass through the \texttt{for} loop, \( \theta_1 \) is empty, and \( \theta_2 := \{v_1/v_3, v_2/v_4, \texttt{zero}\ ?/x_71, \ldots \} \).

The remainder of the computation proceeds similarly. Eventually we find that \( [s, 0] \) is substituted for \( x_3 \), while no further substitution for \( x_4 \) is required. At the termination of the \texttt{for} loop, \( Z \) ends up with the value, \( [s, x_4] \). The resulting rule, therefore, is:

\[
((\texttt{plus} \ [s, 0]) x_4) \Rightarrow [s, x_4].
\]

Example 7. Let us see how the AL-1 algorithm generalizes the example \texttt{plus}(\( s(0), 0, s(0) \)) using the logic program for \texttt{plus}. The proof of the example has only two steps. Initialized to \( g_1 \), \( Z \) possesses the location 0 where the first rewrite occurs, so \texttt{Stretch} has no effect. Applying the second rule to \( g_1 \) gives

\[
Z' := \texttt{plus}(x_1, x_2, x_3),
\]

and results in the substitution

\[
\theta_2 := \{\texttt{plus}(s(x_1), x_2, s(x_3))/g_1\}.
\]

The second step is to apply the first rule at location 0, and again, no \texttt{Stretching} is required. \( Z \) is rewritten to \texttt{true}, with the substitution

\[
\theta_2 := \{0/x_1, x_3/x_2\}.
\]

The resulting rule is:

\[
\texttt{plus}(s(0), x_3, s(x_3)) : - \texttt{true}.
\]

Considering how different the lambda-calculus and logic-program computations are (22 steps vs. 2 steps) in the two examples above, the semantic similarity among the resulting rules is remarkable.
The nondeterministic TRS model has three features that enable it to extend the EBG algorithm to other languages:

- The ability to generalize and specialize while preserving types.
- A general computational process (rewriting) common to the programming languages used in AI.
- Nondeterminism.

The use of a nondeterministic model is appropriate because the algorithm learns from success, and the nondeterminism assumption abstracts away all of the backtracking search that occurs in any actual, deterministic system. Also many programming systems that are closely related when viewed as nondeterministic look very different when implemented as deterministic languages. We would lose much of our generality if we focused only on the deterministic models. To go from a nondeterministic system to a deterministic one, we require a function that chooses the location and the rewrite rule to apply at each step. Adding a new rewrite rule necessitates changing this function. The AL-1 algorithm proposes a new rule, but leaves unanswered the question of how best to modify the choice function. By separating the process of proposing new rules from questions of utility, the formal model makes it easier to reason about such questions.

Finally, in our scheme, a computation is any finite sequence of rewrites. In particular, there is no requirement that the length of the computation be maximal, i.e., that no further rules apply to the final state in the sequence. Thus given a computation of length 5, we could apply AL-1 to the entire computation, or only the first four steps, or the first three, or the last three, etc. Each of these yields a new rule that may, potentially, be used to improve the program. Which sub-computation(s) should we give to AL-1 for analysis? This issue is fundamental to the concept of operability that has been a focus of much discussion (Mostow 81, Mitchell et al. 86, Keller 87).

Since AL-1 generalizes over a path, it is easy to see that when a path is extended, more restrictions (θ₁ and θ₂ in steps 2.2 and 2.3 apply), and the resulting rule is therefore less general. For this reason it seems reasonable to us to recommend the following strategy: in any given computation, apply AL-1 to all sub-computations with a length of two steps. Why length two? Length one is too small: AL-1 will never generalize. Lengths longer than two are compositions of two-step paths, so if a particular path of length k > 2 occurs sufficiently often, the single rule compressing that path will eventually be obtained, two steps at a time, by successive applications of AL-1. It would be worthwhile to evaluate this conjecture theoretically or experimentally.

Acknowledgments

Helpful discussions with Peter Friedland, Smadar Kedar, Rich Keller, Steve Minton, and Masa Numao, and the support of the AI Research Branch at NASA-Ames, have all contributed significantly to this paper.

References


Mostow, J. and N. Bhatnagar. 1987 Failsafe - a floor planner that uses EBC to learn from its failures. In IJCAI'87 Proceedings, pages 249-255, IJCAI/Morgan Kaufmann.