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:

$$\begin{array}{ll} plus(0, x_1, x_1) & :- true. \\ plus(s(x_2), x_3, s(x_4)) & :- plus(x_2, x_3, x_4). \end{array}$$

and the instance $plus(s(0), 0, s(0))$, the EBG algorithm finds the new rule, $plus(s(0), z, s(z)) :- true$ by analyzing the proof and changing certain occurrences of the constant 0 to a variable z. Subsequently, the new instance $plus(s(0), s(0), s(s(0)))$ 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 $plus(z, s(0), s(z)) :- true$ 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:

$$plus\ x\ y \quad := \quad \text{if } x = 0 \text{ then } y \text{ else } s\,(plus\ z\ y), \text{ where } x = s\,(z).$$

Given the input *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 *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 $plus(0, 0, 0)$ and the *plus* program above, it produces the overgeneralization $plus(x, y, z) :- true.$) 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 AL-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.

Goal	→	*Formula* | *Conjunction* | *true*
Goal	→	g_i (for $i \geq 1$)
Formula	→	*plus* (*Term* , *Term* , *Term*)
Conjunction	→	(∧ *Goal* *Goal*)
Term	→	*Term1* | 0
Term	→	x_i (for $i \geq 1$)
Term1	→	*s* (*Term*)

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 ($N \rightarrow \alpha_1 \ldots \alpha_{k_N}$) 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. *Goal* and *Term* are general types whose respective variables are g_i and x_i. *Formula*, *Conjunction*, and *Term1* are special types. *Goal* is the start symbol. The constant symbols are *true*, *plus*, *s*, 0, ∧, comma, and two parentheses. Examples of goals generated by this language are

Expression	→	*Expression1*
Expression	→	*Lambda-param*
Expression	→	*plus* | *succ* | *zero?* | *second*
Expression	→	x_i (for $i \geq 1$)
Expression1	→	λ *Lambda-param*. *Expression*
Expression1	→	(*Expression Expression*)
Lambda-param	→	v_i (for $i \geq 1$)

Figure 2: Grammar for a lambda-calculus language.

$plus(s(0), x_{17}, 0)$ and g_{11}. A conjunctive goal (with two or more subgoals) is represented in this language by a conjunction, e.g., ($\wedge$ $plus(0,0,0)$ $plus(s(0),0,s(0))$).

Example 2. The productions in Figure 2 generate the class of terms of a lambda calculus-based programming language. This language has two general types: *Expression* and *Lambda-param*, whose respective variables are labeled x_i and v_i. *Expression* is the start symbol. The type *Lambda-param* is unusual in that variables are the only strings of that type. *Expression1* is a special type. The constants of the language are λ, period, open-paren, close-paren, *plus*, and others whose utility will become apparent in subsequent examples.

Examples of terms generated by this grammar are: *plus*, x_5, (*plus* (*succ* x_4)), and λv_7 .(*plus* (v_2 v_7)).

As with first-order terms, we define *substitutions* and *replacements* over the sentences of a typed-term grammar. We say that a sentence has *type* N if it can be generated by the grammar starting from nonterminal N. Note that a sentence may have several types. Let γ and β be sentences of type N and G^i, respectively, and let x^i_j be a variable of type G^i. The substitution $\theta = \{\beta/x^i_j\}$ is applied to γ by simultaneously replacing each occurrence of the variable x^i_j in γ by β. Substitutions may be composed: $(\theta_1 \circ \theta_2)(\gamma) = \theta_1(\theta_2(\gamma))$).

For replacements we first need a measure of *location* in a sentence. If γ is a sentence of type G^i it has a unique parse tree T_γ, since the grammar is unambiguous. To each node in T_γ we assign a location as follows: the root has location 0; and the k subtrees of a node whose location is ω have locations $\omega{\cdot}0, \ldots, \omega{\cdot}(k-1)$. A substring γ' of γ is a *subterm* if it is the yield of a subtree of T_γ. We also define the type of a nonleaf subtree of T_γ to be the label of its root node.

We shall denote the subterm of γ at location ω by $\gamma[\omega]$. If β is a sentence of type N and $\gamma' = \gamma[\omega]$ is a subterm of γ of type N, then the *replacement* $\gamma[\omega \leftarrow \beta]$ is obtained by replacing the occurrence γ' in γ at location ω by β.

Example 3. In the grammar of Example 1, $\gamma = plus(x_1, 0, x_1)$ is a sentence of type *Goal* and type *Formula*. When we apply the substitution $\theta = \{s(0)/x_1\}$, the result is $plus(s(0), 0, s(0))$. The subterm 0 occurs at location $\omega = 0{\cdot}0{\cdot}4$ in γ with type *Term*. The replacement $\gamma[0 \cdot 0 \cdot 4 \leftarrow s(x_2)]$ gives $plus(x_1, s(x_2), x_1)$.

Other properties of first-order terms extend to ttg languages. Terms can be ordered by *subsumption*: $\gamma_1 \sqsupseteq \gamma_2$ if there is a substitution θ 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. θ is a *unifier* for terms γ_1 and γ_2 if $\theta(\gamma_1) = \theta(\gamma_2)$. One can readily extend the first-order unification algorithm to compute the most general unifier $\theta = mgu(\gamma_1, \gamma_2)$ 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 distinguished least element $\perp$ is a complete lattice partially ordered by $\sqsupseteq$. 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 algorithms 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. Nondeterminism 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 (α and β) of the same type T. The set of rules is closed under substitution: if $\alpha \Rightarrow \beta$ is a rule and θ is a substitution, then $\theta(\alpha) \Rightarrow \theta(\beta)$ is a rule. A rewriting step is carried out as follows. Let γ be a state and let ω be a location such that $\gamma[\omega] = \alpha$—i.e., $\gamma[\omega]$ is the string α and has type T. Then γ can be rewritten to $\gamma' = \gamma[\omega \leftarrow \beta]$. The notation $\gamma_1 \Rightarrow^* \gamma_2$ indicates that a sequence of zero or more steps transforms γ_1 into γ_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 γ_i at location ω_i yields γ_{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 $plus(0, x, x) := true$ says that any instance of the term $plus(0, x, x)$ can be replaced by the term *true*. When applied to the state $(\wedge\ plus(s(0), 0, s(0))\ \underline{plus(0, 0, 0)})$ at the underlined position, the result is $(\overline{\wedge\ plus(s(0), 0, s(0))}\ true)$. Using the *plus* program to continue this computation for two more steps gives $(\wedge\ true\ true)$, 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 two groups of rules. The first contains all rules of the form:

$$((\lambda v_i.Q)\ R) \Rightarrow [R/v_i]\,Q,$$

where $[R/v_i]Q$ is the result of simultaneously replacing every free occurrence of v_i in Q by R, i.e., a β-reduction.[1] The second group—the rules comprising the actual program—is a list of name-expression pairs, $f \Rightarrow expression$, indicating that the constant f can be replaced by the given expression.

For example, we can recode the *plus* program in lambda calculus as follows. "Zero" (0) is encoded by the expression $\lambda v\,.\,v$. We represent pairs $[t_1, t_2]$ of objects t_1 and t_2 as

$$[t_1, t_2] \equiv \lambda v_1\,.\,((v_1\ t_1)\ t_2).$$

The integer "one" is represented by $[s, 0]$, "two" by $[s, [s, 0]]$, etc., where s is the expression $\lambda v_1.\lambda v_2.v_2$. The successor $(succ\ t)$ of an integer t is computed by the function

$$succ \Rightarrow \lambda v\,.\,[s, v]$$

Let $\lambda v_1.\lambda v_2.v_1$ and $\lambda v_1.\lambda v_2.v_2$ represent *true* and *false*, respectively. A predicate *zero?* that tests whether an integer is zero, giving *true* if so and *false* if not, is as follows:

$$zero? \Rightarrow \lambda v_1\,.\,(v_1\ (\lambda v_2\,.\,\lambda v_3\,.\,v_2)).$$

One can check that $(zero?\ \ 0) \Rightarrow^*$ *true* and $(zero?\,(succ\ v)) \Rightarrow^*$ *false*.

We also need a predicate that returns the second member of a pair:

$$second \Rightarrow \lambda v_1\,.\,(v_1\ (\lambda v_2\,.\,\lambda v_3\,.\,v_3)).$$

[1] Some renaming of parameters may also be needed to avoid variable capture (Hindley & Seldin 86) .

The program consists of rewrite rules for *succ*, *zero?*, and *second* above, and the following rule for *plus*:

$$plus \Rightarrow$$
$$\lambda v_1\,.\,\lambda v_2\,.\,(((zero?\ v_1)v_2)\,(succ\,((plus\,(second\ v_1))\ v_2))).$$

plus begins by applying *zero?* to its first argument. If the result is *true*, the *true* expression selects the second of the two arguments, v_2. If *false*, the result is the successor of $((plus\ (second\ v_1))\ v_2)$, with *plus* applied recursively.

Although the lambda-calculus TRS is completely different from the logic programming one, the semantic structures of the two *plus* programs are quite similar. Thus we should expect the rules learned by AL-1 in the two languages to be sematically comparable.

The AL-1 Algorithm

The 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 γ_i denotes the state before the i'th rule is applied at position ω_i.)

A problem arises if there is no subterm at the required location ω_i in the generalization Z. The procedure *Stretch* is called so that, if necessary, Z acquires a subterm at ω_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 θ 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 ω_i. *Stretch* (Fig. 4) determines the most general state Z' such that $Z \sqsupseteq Z' \sqsupseteq \gamma_i$ and Z' has a subterm at ω_i. For example, if the generalized state Z is just the variable g_1^0, the example γ_i is $(\wedge\ plus(0, 0, 0)\ \underline{plus(s(0), 0, s(0))})$, and ω_i is the location of the underlined subterm, then *Stretch* would specialize Z to $Z' = (\wedge\ g_2^0\ \underline{g_3^0})$.

The main property of the AL-1 algorithm is given by the following theorem:

Theorem. *Let γ_1 be the initial state of the input computation, and let $\hat{\alpha} \Rightarrow \hat{\beta}$ be the rule output by the AL-1 algorithm. Then $\hat{\alpha}$ is the most general state $\sqsupseteq \gamma_1$ such that the sequence of rules in the input computation is*

Input: An n-step computation with γ_i the i'th state and $\alpha_i \Rightarrow \beta_i$ the i'th rule.
Procedure:

1. Initialize: θ = the empty substitution. $Z = x_1^0$, a fresh variable of type G^0.
2. For $i = 1$ up to n:
   - 2.1 $Z' := \mathit{Stretch}(Z, \gamma_i, \omega_i)$. /* If necessary, extend Z so that location ω_i exists. */ /* (See Fig. 4 and related text.) */
   - 2.2 $\theta_1 := \mathit{mgu}(Z', Z)$. /* Unifier for unstretched and stretched Z */
   - 2.3 $\theta_2 := \mathit{mgu}(Z'[\omega_i], \alpha_i)$. /* Unifier for α_i and the subterm at location ω_i */
   - 2.4 $Z' := \theta_2(Z'[\omega_i \leftarrow \beta_i])$. /* Apply an instance of the rule to Z' at ω_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: $\theta(x_1^0) \Rightarrow Z$.

Figure 3: The AL-1 Algorithm.

applicable; and $\hat{\beta}$ is the configuration that results from applying this sequence of rules to $\hat{\alpha}$.

Example 6. With the *plus* program in Example 5 ((*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: ((*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, *plus* is replaced by its lambda definition:

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

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

$$Z' := ((\underline{x_2}\ x_3)\ x_4).$$

Now we apply the *plus* rule to x_2, with the result:

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

For this first pass through the *for* loop, $\theta_1 := \{((x_2\ x_3)\ x_4)/x_1\}$, and $\theta_2 := \{\mathit{plus}/x_2\}$.

The next rewrite is a β-reduction based on the rule:

$$(\lambda v_3 . \lambda v_4 . (((x_{71}\ v_3)\ v_4)(x_{72}((x_{73}\ (x_{74}\ v_3))v_4)))\ x_{75})$$
$$\Rightarrow \lambda v_4 . (((x_{71}\ x_{75})\ v_4)(x_{72}((x_{73}\ (x_{74}\ x_{75}))v_4))).$$

The location ($\omega_2 = 0$) of the rewrite already exists in the generalization, so *Stretch* has no effect on Z. Applying the rule to Z gives

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

For this pass through the loop, θ_1 is empty, and $\theta_2 := \{v_1/v_3, v_2/v_4, \mathit{zero?}/x_{71}, \ldots\}$.

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 *for* loop, Z ends up with the value, $[s, x_4]$. The resulting rule, therefore, is:

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

Example 7. Let us see how the AL-1 algorithm generalizes the example $\mathit{plus}(s(0), 0, s(0))$ using the logic program for *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 *Stretch* has no effect. Applying the second rule to g_1 gives

$$Z' := \mathit{plus}(x_1, x_2, x_3),$$

and results in the substitution

$$\theta_2 := \{\mathit{plus}(s(x_1), x_2, s(x_3))/g_1\}.$$

The second step is to apply the first rule at location 0, and again, no *Stretch*ing is required. Z is rewritten to *true*, with the substitution

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

The resulting rule is:

$$\mathit{plus}(s(0), x_3, s(x_3)) :- \mathit{true}.$$

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.

Conclusions

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 operationality 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 (θ_1 and θ_2 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.

1. Initialize: $Z' := Z$.
2. While Z' has no subterm at ω,
   2.1 Let $\bar{\omega}$ be the longest prefix of ω such that Z' has a variable x_i^j at $\bar{\omega}$.
   2.2 Let θ be the substitution $\{Expand(\gamma_i, \bar{\omega}) \,/\, x_i^j\}$.
   2.3 $Z' := \theta(Z')$.
3. Return Z'.

where $Expand(\gamma_i, \bar{\omega})$ is:

1. In the parse tree of γ_i, let the production for the nonterminal N at location $\bar{\omega}$ be $N \rightarrow \zeta_1 \ \ldots \ \zeta_k$. Initialize $P_i :=$ *empty-string* for all i, $0 \leq i \leq k-1$.
2. For each i from 0 to $k-1$:
   Case:
   2.1 ζ_i is a constant c: $P_i := c$.
   2.2 ζ_i is a general type G^r or a variable x_j^r: $P_i := x_m^r$, where x_m^r is a fresh variable unused in any expression so far in this or any calling routine.
   2.3 ζ_i is any other nonterminal: $P_i := Expand(\gamma_i, \bar{\omega} \cdot i)$.
3. Return the string $P_0 \cdot \ \ldots \ \cdot P_{k-1}$.

Figure 4: The routine $Stretch(Z, \gamma_i, \omega_i)$.

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

Avenhaus, J. and K. Madlener. 1990. Term rewriting and equational reasoning. In *Formal Techniques in Artificial Intelligence. A Sourcebook*, R. Banerji, editor. Elsevier Science.

Bhatnagar, N. 1988. A correctness proof of explanation-based generalization as resolution theorem proving. In *Proceedings: AAAI Explanation-based Learning Symposium*, AAAI (Spring Symposium Series).

Cohen, W. W. 1989. *Solution path caching mechanisms which provably improve performance.* Technical Report DCS-TR-254, Rutgers University.

Dietzen, S. and F. Pfenning. 1989. *Higher-order and modal logic as a frameowrk for explanation-based generalization.* Technical Report CMU-CS-89-160, Carnegie-Mellon University School of Computer Science.

Greiner, R. 1989. Towards a formal analysis of EBL. In *Proc. Sixth Int. Machine Learning Workshop*, Morgan Kaufmann.

Hindley, J. R. and J. P. Seldin. 1986. *Introduction to Combinators and Lambda-Calculus.* Cambridge University Press.

Kedar-Cabelli, S. and T. McCarty. 1987 Explanation-based generalization as resolution theorem proving. In *Proc. 4th International Workshop on Machine Learning*, 1987.

Keller, R. M. 1987. Defining operationality for explanation-based learning. In *Proceedings of AAAI-87*, Morgan Kauffman.

Kodratoff, Y. 1988. *Introduction to Machine Learning.* Morgan Kaufmann, 1988.

Laird, P. 1988. *Learning from Good and Bad Data.* Klüwer Academic.

Laird, P. and E. Gamble. 1990. *On Analytical Learning.* Technical Report RIA-90-01-17-7, NASA Ames Research Center, AI Research Branch.

Minton, S. 1988. *Learning effective search control knowledge: an explanation-based approach.* PhD thesis, Carnegie Mellon University.

Mitchell, T. M., P. Utgoff, and R. Banerji. 1983. Learning by experimentation. In *Machine Learning: An Artificial Intelligence Approach*, Morgan Kaufmann.

Mitchell, T. M. , R. M. Keller, and S. Kedar-Cabelli 1986. Explanation-based generalization: a unifying view. *Machine Learning*, 1.

Mooney, R. 1988. *A general explanation-based learning mechanism and its application to narrative understanding.* PhD thesis, University of Illinois at Urbana-Champaign.

Mostow, J. 1981. *Mechanical transformation of task heuristics into operationsl procedures.* PhD thesis, Carnegie Mellon University.

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

Natarajan, B. and P. Tadepalli. 1988. Two new frameworks for learning. In *Proceedings, 5th International Machine Learning Conference*, pages 402–415.

Natarajan, B. 1989. On learning from exercises. In *Proc. 2nd Workshop on Computational Learning Theory.*