Analytical Learning and Term-Rewriting Systems

PHILIP LAIRD AND EVAN GAMBLE

AI RESEARCH BRANCH, MAIL STOP 244-17
NASA AMES RESEARCH CENTER
MOFFETT FIELD, CA 94035

NASA Ames Research Center

Artificial Intelligence Research Branch

Technical Report FIA-90-06-17-7

June, 1990

Analytical Learning and Term-Rewriting Systems

Philip Laird Evan Gamble

June 15, 1990

Abstract

Analytical learning is a set of machine-learning techniques for revising the representation of a theory based on a small set of examples of that theory. When the representation of the theory is correct and complete but perhaps inefficient, an important objective of such analysis is to improve the computational efficiency of the representation.

Several algorithms with this purpose have been suggested, most of which are closely tied to a first-order logical language and are variants of goal regression, such as the familiar explanation-based generalization (EBG) procedure. But because predicate calculus is a poor representation for some domains, we would like to extend these learning algorithms to apply to other computational models.

In this paper we show that the goal-regression technique applies to a large family of programming languages, all based on the notion of term-rewriting systems. Included in this family are three language families of importance to artificial intelligence: logic programming (such as Prolog); lambda calculus (such as LISP); and combinator-based languages (such as FP). We also exhibit a new analytical learning algorithm, AL-2, that learns from success but is otherwise quite different from EBG.

These results suggest that term-rewriting systems are a good framework for analytical-learning research in general, and that further research should be invested in finding new learning techniques in the framework.

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 [23, 15], that learn from success, and other algorithms (e.g., [19, 26]) that learn from failure.

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, consider this simple logic program for integer addition, in which $plus(a, b, c)$ is intended to mean $a + b = c$ and $s(a)$ indicates $a + 1$:

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

With this program 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 the new rule, instead of the two steps required by the original program, provided the program can decide quickly that the new rule is the appropriate one for solving this new instance.

The results from applying this technique alone 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 and useful generalizations (e.g., in the example above, the rule $plus(z, s(0), s(z)) :- true$) 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" [24], 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 represents the rules in predicate calculus and simulates a first-order theorem prover. Yet domains arise in practice for which predicate calculus is at best an awkward representation for the essential domain properties [22, 24]. In these situations the ability to use another language and still be able to apply analytical learning algorithms would be highly desirable.

Is EBG, then, just a syntactical trick that depends on logic for its existence? If so, its status as a *bona fide* learning method is questionable, since 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 typical functional language the *plus* program might be coded:

$$\begin{array}{lll} plus\ x\ y & := & \text{if } x = 0 \text{ then } y \\ & & \text{else } s\,(plus\ z\ y), \text{ where } x = s\,(z). \end{array}$$

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 first translating to a logical representation.

Also, while the formal foundations of EBL have been studied (e.g., [11, 28, 27, 6]), most of this work has abstracted away the generalization process in order to model the benefits of path compression. Notable exceptions include [4], where a notion of correctness is defined for EBG and an EBG algorithm is proved correct, and [8], where EBG is treated as a higher-order process (since it handles programs as objects), and where modal logic is introduced to distinguish tentative, non-operational constructs from permanent, operational ones.

Aside from this work, presentations of the EBG algorithm in the AI literature have generally been informal, and often incomplete. The elegant PROLOG-EBG algorithm [15] is a case in point. In certain cases it will overgeneralize. For example, given the instance $plus(0, 0, 0)$ and the *plus* program above, it produces the overgeneralization $plus(x, y, z) :- true$. Recently several papers, a thesis, and even a textbook have reproduced this algorithm without noticing or correcting the problem. All of this points to the need for more rigorous presentations of analytical-learning algorithms and a consistent framework for such presentations.

This paper addresses both these 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.

- *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, by separating the mechanics of generalization from other issues that are more language dependent, the TRS formalisms help to clarify the fundamental learning problems.

To show that the TRS framework is also useful for formulating new analytical-learning algorithms, we describe a new algorithm, called AL-2. Like EBG, the algorithm learns from success while preserving the semantics, and uses the proof of the example to propose new rules for potential inclusion in the knowledge base. And like EBG, each new rule may have the effect of improving or degrading the average performance of the program, depending on what problem instances occur subsequently. Unlike EBG, the *language* in which the rules are expressed is modified. New symbols may be introduced in order to abbreviate frequently occurring terms and to shorten common sub-proofs. This resembles what humans do, for example, when we say "EBG" instead of "explanation-based generalization" or "prime" instead of "natural number divisible only by itself and one". Thus AL-2 joins EBG as another technique for learning from success, and can be added to a growing list of analytical learning methods (e.g., [19, 24, 26, 29]). As the toolbox for analytical learning expands, it becomes more important to abstract the algorithms away from specific domains, to formalize the procedures, and to characterize their properties. The TRS framework facilitates this task.

The style in which the results in this paper are presented is mathematical, since the main objective is formally to extend the EBG algorithm. The NASA research project under which this research has been conducted, however, is developing practical algorithms for Machine Learning. The applications we envision for algorithms of the type considered here include the case wherein a correct program or expert system learns from experience to reduce the amount of computation and the size of internal storage required to solve "typical" problem instances, without ever compromising correctness. Experience has shown that application programs often expend most of their resources repeating a rather small number of essentially identical steps. Effective methods for indexing and rearranging partial computations may, therefore, offer significant returns in overall performance.

The contents of this paper are as follows. We first develop a family of *typed-term languages* that possesses a lattice structure suited to the kind of generalization and specialization procedures needed for analytical learning. Based on these we define the class of term-rewriting systems that serve as computational models. The AL-1 and AL-2 algorithms are expressed in this framework and are accompanied by theorems that characterize their behavior. Along the way, we shall compare our framework to related work on term rewriting and unification theory.

Typed-Term Languages

The algorithms we shall develop operate on symbolic (syntactical) expressions that we call *terms*. The set of all admissible terms is a formal language generated by a special class of context-free grammars. The non-terminals of this grammar determine the *types* (also called *sorts*) assigned to each term; we therefore call these languages *typed-term languages*. In this section we define these languages and give examples. In the following section we develop a lattice structure over these terms, so that we can use the meet and join operations in our algorithms.

Notation. Familiarity with the basic concepts and conventions of formal language theory is assumed. Throughout this paper we use the symbol ϵ to designate the empty string. Concatenation is denoted by $\cdot$ or simply by juxtaposition. If A is a set of symbols, A^* denotes the Kleene closure of A under finite concatenation, and $A^+ = A^* - \{\epsilon\}$. N denotes the set of natural numbers. ∎

Definition 1 A *typed-term grammar (ttg)* is an unambiguous, context-free grammar $\langle \mathcal{N}, \mathcal{A}, \mathcal{P}, G^0 \rangle$, with the following special form:

- The set $\mathcal{N}$ of non-terminal 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 $\mathcal{A}$ 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 $\mathcal{P}$ of productions satisfies three conditions: (1) For any non-terminal N, the set of sentences generated by N is non-empty and does not contain the empty string. (2) For any non-terminal N, the right-hand sides of all its productions ($N \rightarrow \alpha_1 \ldots \alpha_{k_N}$) have the same length (or *arity*) k_N. For general types this length is one. (3) For each general type G^i and each variable x^i_j of that type, there is a production $G^i \rightarrow x^i_j$. No other productions contain variables. ∎

Without loss of generality we can assume that all useless symbols and productions have been removed from the grammar. We often refer to the non-terminal symbols in a typed-term grammar as *types*. The set of strings that can be generated from the non-terminal N is denoted $\mathcal{L}(N)$ and described as a *typed-term language (ttl) of type N*. Note that a string may have one or more types.

Also note that, according to our our terminology, variables (i.e., the symbols x^i_j) are *terminal* symbols. Since some texts describe the non-terminal symbols of a context-free grammar as variables, there is potential for confusion. In our terminology, variables are distinguished classes of terminal symbols that may occur in the strings generated by the grammar. Each countable set of such variables is associated with its own general type. Thus G^0 can generate the variables x^0_1, x^0_2, ..., G^1 can generate x^1_1, x^1_2, ... , and so forth. Variables in our terms play much the same role as universally quantified, bound variables in the formulas of first-order logic. The set of variables in a term τ is denoted $\mathcal{V}(\tau)$. The term is called *ground* if $\mathcal{V}(\tau)$ is empty.

Below we shall give three examples of ttg's generating, respectively, the terms of a logic programming language (LP), a simple applicative programming language (AP), and a lambda calculus-based programming language (LC). These three languages will serve as running examples throughout the presentation.

Example 2 [LP] The grammar below generates a class of goals appropriate to the logic program *plus* in the introduction. Accordingly we call our principal type *Goal* rather than G^0. There is one other general type, to which we assign the non-terminal *Term* (in preference to G^1), and several special types (*Formula*, *Conjunction*, and *Term1*). This language has constant symbols *plus*, *true*, *s*, 0, $\wedge$, comma, and two parentheses. The variables g_i (for $i > 0$) are generated by *Goal*, while the variables x_i are generated by *Term*. (Only the latter set of variables are used in conventional logic programming.)

$$
\begin{array}{rcl}
\mathit{Goal} & \rightarrow & \mathit{Formula} \\
\mathit{Goal} & \rightarrow & \mathit{Conjunction} \\
\mathit{Goal} & \rightarrow & \mathit{true} \\
\mathit{Goal} & \rightarrow & g_i \quad (\text{for } i \geq 1) \\
\mathit{Formula} & \rightarrow & \mathit{plus}\,(\,\mathit{Term}\,,\,\mathit{Term}\,,\,\mathit{Term}\,) \\
\mathit{Conjunction} & \rightarrow & \wedge\,(\,\mathit{Goal}\,,\,\mathit{Goal}\,) \\
\mathit{Term} & \rightarrow & 0 \\
\mathit{Term} & \rightarrow & x_i \quad (\text{for } i \geq 1) \\
\mathit{Term} & \rightarrow & \mathit{Term1} \\
\mathit{Term1} & \rightarrow & s\,(\,\mathit{Term}\,)
\end{array}
$$

Examples of goals generated by this language are $plus(s(0), x_{17}, 0)$ and g_{11}. A conjunctive pair of goals is represented in this language by a parenthesized conjunction, e.g., $\wedge$ ($plus(0,0,0)$, $plus(s(0),0,s(0))$). Strings that are not goals include x_1^1, $s(x_1, x_2)$, $plus()$, and $plus(true, 0, 0)$. ∎

Example 3 [AP] The grammar below generates the class of terms of a simple functional applicative programming language. Such languages are based on combinatory logic [12] and capture the basic notions of functional application. In this example, the language has one general type, which we shall write *Expression* instead of G^0. In addition it has one special type (*Application*), four constant symbols (*plus*, *succ*, open-paren, and close-paren) and a countable set of variables (x_i). The start symbol is *Expression*. The expression (τ_1 τ_2), where τ_1 and τ_2 are arbitrary terms, is intended to indicate that the function denoted by τ_1 is to be applied to the argument τ_2.

$$
\begin{aligned}
\textit{Expression} &\rightarrow \textit{Application} \\
\textit{Expression} &\rightarrow 0 \\
\textit{Expression} &\rightarrow \textit{succ} \\
\textit{Expression} &\rightarrow \textit{plus} \\
\textit{Expression} &\rightarrow x_i \quad (\text{for } i \geq 1) \\
\textit{Application} &\rightarrow (\ \textit{Expression}\ \ \textit{Expression}\ )
\end{aligned}
$$

Examples of terms generated by this grammar are: *plus*, x_5, (*plus* (*succ* x_5)) and ((*plus succ*) x_5). Note that the arity of the non-terminal *Application* is four, while that of *Expression* is (necessarily) one. ∎

Example 4 [LC] The grammar below generates 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. 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.

$$
\begin{aligned}
\textit{Expression} &\rightarrow \textit{Expression1} \\
\textit{Expression} &\rightarrow \textit{Lambda-param} \\
\textit{Expression} &\rightarrow \textit{plus} \mid \textit{succ} \mid \textit{zero?} \mid \textit{second} \\
\textit{Expression} &\rightarrow x_i \quad (\text{for } i \geq 1) \\
\textit{Expression1} &\rightarrow \lambda\ \textit{Lambda-param}.\ \textit{Expression} \\
\textit{Expression1} &\rightarrow (\ \textit{Expression}\ \ \textit{Expression}\ ) \\
\textit{Lambda-param} &\rightarrow v_i \quad (\text{for } i \geq 1)
\end{aligned}
$$

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

In the remainder of this section and in the next, we extend to ttl's such operations as substitution, replacement, and unification familiar from first-order unification theory. For this purpose, we need some notation. Fix a typed-term grammar (ttg), let N be any type, and let τ be a term of type N. The unique parse tree whose root is labeled N and whose yield (i.e., the string obtained by concatenating the labels on the leaves in order from left to right) is τ, is called the *parse tree of* τ and is written $tree(\tau)$. The yield of a parse tree Y is denoted $yield(Y)$. Thus $yield(tree(\tau)) = \tau$.

In this paper we shall adopt a specific data structure for parse trees. Let Y be a parse tree whose root is labeled r.

- If Y is a leaf, it is represented simply by its label, r.
- Otherwise, let $Y_1, \ldots, Y_k$ be the (representations of the) immediate subtrees of Y; then Y is represented:

$$r\ Y_1, \ldots, Y_k.$$

Since ttg's are unambiguous by definition, and since each non-terminal has a fixed arity, this representation is efficient for constructing parse trees from terms and for determining the yield of a parse tree.

Definition 5 To each node of a tree Y we assign a unique string of integers, called a *location*, as follows:

- The location of the root of Y is ϵ;
- Let $Y_1, \ldots, Y_k$ be the immediate descendents of a node whose location is ω; then for $1 \leq i \leq k$, the location of Y_i is $i \cdot \omega$.

For brevity the dot ($\cdot$) will often be omitted when confusion is unlikely, e.g., 12 instead of $1 \cdot 2$.

Example 6 [AP] Refer to the grammar above for the AP language. The parse tree for the term $\tau =$ (*plus* (x_1 0)) is

$$\begin{aligned} tree(\tau) \;=\; & \textit{Expression Application} \;(\; \textit{Expression plus} \\ & \textit{Expression Application} \;(\; \textit{Expression}\; x_1\; \textit{Expression}\; 0\;)\;) \end{aligned}$$

The location of the subtree *Expression* 0 is $1 \cdot 3 \cdot 1 \cdot 3$; the location of the subtree 0 is $1 \cdot 3 \cdot 1 \cdot 3 \cdot 1$. ∎

Definition 7 Let τ_1 be a term of type B_1 and τ_2, a term of type B_2 in a ttg. We say that τ_2 *occurs in* τ_1 *at location* ω if, at location ω in $tree(\tau_1)$, there is a node labeled B_2 and the yield of the subtree rooted at this node is τ_2.

The set of *term occurrences* in τ is the set of locations of nodes in the parse of τ that are labeled with any type N. This set will be written $\Omega(\tau)$.

The term that occurs at location ω in τ is denoted $\tau[\omega]$.

Example 8 [AP] In Example 6, $\Omega(\tau) = \{\epsilon, 1, 12, 13, 141, 1312, 1313\}$. The subterm (x_1 0) of type *Expression* occurs at location 13; the same subterm, but of type *Application*, occurs at location 131. ∎

Definition 9 (*Replacement*) Let τ_1 be a term of type B_1 and τ_2, a term of type B_2. Let $\omega \in \Omega(\tau_1)$ be the location of a subterm of type B_2 within τ_1. The string $\tau[\omega \leftarrow \tau_2]$ is obtained by replacing the term in τ_1 at location ω by τ_2.

Example 10 [AP] Refer again to Example 6. If τ_1 = (*plus* (x_1 0)) and τ_2 = *succ*, then $\tau_1[13 \leftarrow t_2]$ = (*plus succ*). ∎

Definition 11 (*Substitution*) With respect to a ttg, a *substitution* is a mapping $\theta: \mathcal{V} \rightarrow \mathcal{A}^+$ from the set of variables $\mathcal{V}$ to non-empty strings of terminals $\mathcal{A}^+$ such that (1) for all but finitely many variables x, $\theta(x) = x$, and (2) if $\theta(x_j^i) = \tau$, then $\tau \in \mathcal{L}(G^i)$. That is, a substitution changes only finitely many variables and maps variables only to terms of the same type. The set of variables x such that $\theta(x) \neq x$ is called the *domain* of θ, written $\mathrm{dom}(\theta)$. Its *codomain* is the set of terms $\theta(\mathrm{dom}(\theta))$.

Since the grammar is unambiguous, the sets $\mathcal{L}(N)$ are freely generated for each N [10]; hence there is a unique morphic extension $\hat{\theta}$ of θ from $\mathcal{V}$ to the domain of arbitrary strings in $\mathcal{A}^*$:

- $\hat{\theta}(\epsilon) = \epsilon$.
- $\hat{\theta}(x) = \theta(x)$ for all variables x.
- $\hat{\theta}(\tau) = \tau$ for all terminal symbols τ except variables; and
- $\hat{\theta}(\tau_1 \cdot \tau_2) = \hat{\theta}(\tau_1) \cdot \hat{\theta}(\tau_2)$ for any strings τ_1 and τ_2 in $\mathcal{A}^+$.

Henceforth we shall not distinguish between θ and its extension $\hat{\theta}$.

Lemma 12 For any type N in a ttg, the language $\mathcal{L}(N)$ is closed under replacement and substitution. Specifically, if N and N' are types, $\tau \in \mathcal{L}(N)$, $\tau' \in \mathcal{L}(N')$, and ω is the location of a term of type N' within τ, then the string $\tau[\omega \leftarrow \tau']$ obtained from τ and τ' by replacement is itself a term in $\mathcal{L}(N)$.

Likewise, for any substitution θ and string $\tau \in \mathcal{L}(N)$, the string $\theta(\tau)$ is itself a term in $\mathcal{L}(N)$.

(The easy proof consists in showing that the term obtained by replacement or substitution is still generated by the grammar starting from the non-terminal N.)

To summarize, we have defined a class of languages generated by typed-term grammars, and defined the notions of substitution for variables and replacement of a subterm at a specific location. Whereas substitution is purely a string operation, replacement requires reference to the parse tree in order to identify the subterm at the given location. Nevertheless, these notions are quite similar to the corresponding operations for first-order terms.

One of the most useful features of first-order terms is that they form a lattice under the subsumption ordering. The meet and join operations of this lattice reflect the semantic notions of specialization and generalization, respectively. In the next section, we develop a similar algebraic structure for the expressions of a typed-term algebra.

The Subsumption Lattice of Terms

In this section we order terms according to generality and develop a lattice structure over the set of strings generated by general terms. Much of this is based on the well-known theory of first-order terms, so proofs are sketched except where our formalism is substantially different.

Throughout this section we assume that the typed-term grammar $\mathcal{G} = \langle \mathcal{N}, \mathcal{A}, G^0, \mathcal{P} \rangle$ has been fixed. Let B denote an arbitrary non-terminal symbol in the grammar.

Definition 13 Let $\perp$ ("bottom") be a special symbol not found in the grammar $\mathcal{G}$. $\mathcal{T}(B)$ is the set $\{\perp\} \cup \mathcal{L}(B)$—that is, the set of terms generated from the non-terminal B together with the special term $\perp$. Similarly, let $\hat{\mathcal{T}}(B)$ be the set $\{\perp\} \cup \{tree(\tau) \mid \tau \in \mathcal{L}(B)\}$.

Definition 14 The binary *subsumption* relation $\sqsupseteq$ on $\mathcal{T}(B)$ is defined as follows:

- $\tau \sqsupseteq \perp$ for all $\tau \in \mathcal{T}(B)$;
- For $\tau_1, \tau_2 \in \mathcal{L}(B)$, $\tau_1 \sqsupseteq \tau_2$ iff there exists a substitution θ such that $\theta(\tau_1) = \tau_2$.

If both $\tau_1 \sqsupseteq \tau_2$ and $\tau_2 \sqsupseteq \tau_1$, then we say that τ_1 and τ_2 are *variants*, and write $\tau_1 \equiv \tau_2$; otherwise we write $\tau_1 \sqsupset \tau_2$. $\sqsubseteq$ is the inverse of $\sqsupseteq$: $\tau_1 \sqsupseteq \tau_2$ iff $\tau_2 \sqsubseteq \tau_1$. Similarly, $\sqsubset$ is the inverse of $\sqsupset$.

$\equiv$ is an equivalence relation on $\mathcal{T}(B)$. $\sqsupseteq$ is a quasi-ordering (a reflexive and transitive relation) but not a partial ordering; for example, $x_1^1 \sqsupseteq x_2^1$ and $x_2^1 \sqsupseteq x_1^1$.

Definition 15 If a substitution θ is a bijective mapping from $\mathcal{A}^+$ to $\mathcal{A}^+$, then we call θ a *permutation.*

Lemma 16 Two terms τ_1 and $\tau_2 \in \mathcal{L}(B)$ are variants iff there exists a permutation θ such that $\theta(\tau_1) = \tau_2$.

PROOF: If τ_1 and τ_2 are variants, then by Definition 14 there exist substitutions θ and ψ such that $\theta(\tau_1) = \tau_2$ and $\psi(\tau_2) = \tau_1$. For each variable x occurring in τ_1, $\psi(\theta(x)) = x$; thus $\theta(x)$ must be a variable. If x and y are distinct variables occurring in τ_1, then $\psi(\theta(x)) \neq \psi(\theta(y))$, and thus $\theta(x) \neq \theta(y)$. We may thus take as the required permutation a substitution θ' such that $\theta'(x) = \theta(x)$ and $\theta'(\theta(x)) = x$ for all variables x occurring in τ_1, and $\theta'(x) = x$ for variables occuring in neither τ_1 nor τ_2.

The opposite demonstration, that τ_1 and τ_2 are variants if there exists a permutation θ such that $\theta(\tau_1) = \tau_2$, is accomplished by setting $\psi = \theta^{-1}$ and noting that $\psi(\tau_2) = \tau_1$. ∎

Although substitution was defined on terms (Definition 11), there is an obvious parallel operation on parse trees, one that was implicitly used in Lemma 12. Whereas with terms one replaces a variable x_i^j by a term τ in $\mathcal{L}(G^j)$, with trees one replaces the subtree $G^j \cdot x_i^j$ by the subtree $tree(\tau)$ whose root is labeled G^j. Thus over parse trees a substitution θ is a mapping from *tree variables* (that is, trees of the form $G^j \cdot x_i^j$ for some j) to trees with the same root G^j such that θ is the identity mapping for almost all tree variables. As usual, θ extends morphically to a mapping on $\hat{\mathcal{T}}(B)$.

With substitution on $\hat{\mathcal{T}}(B)$ as the basis, we can define the subsumption ordering $\sqsupseteq$ and equivalence with respect to that ordering ($\equiv$) entirely analogously to Definition 14. The result corresponding to Lemma 16 also holds.

Lemma 17 Let m be the mapping from $\mathcal{T}(B)$ to $\hat{\mathcal{T}}(B)$ such that $m(\tau)$ is the parse tree for τ whose root is B, and $m(\bot) = \bot$. Then m is an order-isomorphism between $\mathcal{T}(B)$ and $\hat{\mathcal{T}}(B)$. That is, m is a bijection that preserves the ordering:

$$\tau_1 \sqsupseteq \tau_2 \text{ iff } m(\tau_1) \sqsupseteq m(\tau_2).$$

The trivial proof is based on the fact that the grammar is unambiguous and that $\bot$ is a distinguished symbol.

Definition 18 $\mathcal{T}(B)/_{\equiv}$ is the set of equivalence classes of $\mathcal{T}(B)$ with respect to the relation $\equiv$ on $\mathcal{T}(B)$. Similarly $\hat{\mathcal{T}}(B)/_{\equiv}$ is the set of equivalence classes of $\hat{\mathcal{T}}(B)$ with respect to the corresponding equivalence $\equiv$ on $\hat{\mathcal{T}}(B)$. $[\tau]$ denotes the $\equiv$-equivalence class of which τ is a member.

The purpose of this section is to argue that $\mathcal{T}(B)/_{\equiv}$ is a meet semilattice lattice for every type B, and a complete lattice for every general type G^i. The idea is to inject $\hat{\mathcal{T}}(B)$ into a lattice of first-order terms so as to preserve meets and joins.

Before doing so, however, let us review some results from unification theory. Recall the definition of a family FOT of first order terms. Let $\mathcal{F}$ be a countable set of *function symbols* each with a fixed arity, and let $\mathcal{V}$ be a countable set of variables such that $\mathcal{F} \cap \mathcal{V} = \emptyset$. The set FOT of first-order terms is the smallest set containing $\mathcal{V}$ and the nullary functions (constants) in $\mathcal{F}$ and closed under functional application, i.e., $F\ \tau_1\ \ldots\ \tau_n$, where $F \in \mathcal{F}$ is a function symbol of arity $n > 0$ and $\tau_i \in FOT$ for each i. With variants in FOT taken to be equivalent, the set $FOT \cup \{\bot\}$ partially ordered by subsumption is a complete lattice, with effective algorithms for join ($\sqcup$) and meet ($\sqcap$) [14, 30]. The meet operation is computed using a unification algorithm, since by a well-known theorem of Robinson, any finite, unifiable set of first-order terms has a most-general unifier that is unique modulo variants.

Since our terms are typed, the first-order theory does not apply directly, but the unification theory of many-sorted terms has also been studied [32]. Briefly, there is a set of sorts; for each sort there is a countable set of variables and a finite set of constants; and each function symbol f of k arguments is assigned a string $W_1 \ldots W_{k+1}$ indicating that the i'th argument has sort W_i (for $1 \leq i \leq k$) and the result has sort W_{k+1}. There is also a unification theorem

similar to Robinson's for the well-formed terms generated by these symbols. This assumes, however, that variables of one sort do not unify with variables of another sort. When one sort may be a subsort of another, the theory is more complex. For example, if x is a variable of type *real* and y is a variable of type *integer*, then we can unify the terms x and y by substituting y for x, but not conversely. Walther [32] shows that, when sorts are partially ordered, two unifiable terms may have zero, one, or many most general unifiers (*mgu*'s). He further shows that a necessary and sufficient condition for a Robinsonian property (existence of a unique *mgu*) is that this partial order among sorts be a forest.

Since unification is how we propose to implement our meet operations, we likewise seek a Robinsonian property to apply to our terms. Moreover, interpreting nonterminals in a ttg as "sorts", we see that the various sorts are related, in that if $N_1 \rightarrow N_2$, then N_2 is a subsort of N_1. However, the ordering is not a forest, and while it may be possible, we have not found a way to map our use of types onto a sort hierarchy that is a forest. We shall, however, obtain a Robinsonian property, indicating that our notion of types is somewhat different. This difference is briefly characterized in an appendix to this report.

The Meet Operation

Notice that if we view non-terminals as function symbols and relate constants and variables in the obvious way, parse trees look very much like first-order terms. Indeed, having established an order-isomorphism between the strings $\mathcal{T}(B)$ and their parse trees $\hat{\mathcal{T}}(B)$, we are tempted to establish a lattice isomorphism between $\hat{\mathcal{T}}(B)/_{\equiv}$ and the corresponding set of first-order terms. Unfortunately, this is not possible, because there are many first-order terms (for example, $B\,x^i$ where B is a special type) that correspond to no parse tree. But as it turns out, $\hat{\mathcal{T}}(B)/_{\equiv}$ is isomorphic to a sub-semilattice of $FOT/_{\equiv}$, and to a complete sublattice when B is a general type G^i.

To establish this correspondence requires some work, but having done so, we shall have, as a consequence of Lemma 17, that $\mathcal{T}(B)/_{\equiv}$ is a semilattice (ordered by subsumption) and that $\mathcal{T}(G^i)/_{\equiv}$ is a complete lattice. The particular result we need for our algorithms is that (apart from variants) there is a unique meet (greatest lower bound), $\tau_1 \sqcap \tau_2$, for any two terms τ_1 and τ_2, and an effective algorithm for computing it. Also, a theorem characterizing the AL-1 algorithm will be based on fact that $\mathcal{T}(G^0)/_{\equiv}$ is a complete lattice.

Example 19 [AP] A brief example will quickly illustrate how we compute the meet of two parse trees. Refer to Example 6, where the parse tree for the term $\tau = (\ plus\ (\ x_1\ 0))$ is given. Suppose we wish to find the greatest lower bound between this and the term $\tau' = (\ plus\ x_2)$. The parse tree for τ' is

$$Expression\ Application\ (\ Expression\ plus\ Expression\ x_2).$$

Treating *Expression* as a unary and *Application* as a 4-ary function symbol and *plus* and 0 as constants, we unify the parse trees for τ and τ' with the usual first-order unification algorithm. The resulting substitution replaces x_2 by the tree: *Application* (*Expression* x_1

Expression 0). The corresponding substitution for the two terms is then $\{x_2 := (x_1\ 0)\}$. The purpose of using the parse trees to construct the substitututution is to ensure that subterms are unified with other subterms of the appropriate type. ∎

We define a particular family of first-order terms, FOT. The function symbols in the first-order language consist of the types $\mathcal{N}$ in the ttg, with the same arities as in the grammar. Constants $\mathcal{A}$ and variables $\mathcal{V}$ in the grammar act as constants and variables, respectively, in FOT. Let $FOT_B \subseteq FOT$ be the subset that contains only variables and terms whose leftmost function symbol is B. Given the well-known lattice properties of FOT (when ordered by subsumption), one can readily show that FOT_B is a lattice when variants are taken to be equivalent and a unique smallest element $\perp$ is adjoined. We denote this lattice by $FOT_B/_{\equiv}$. Note that $x_1^1 \equiv x_1^2$ in FOT, but not in $\mathcal{T}$.

Next we establish a straightforward mapping μ from $\hat{\mathcal{T}}$ into FOT, as follows. Trees of the form $N\zeta_1 \ldots \zeta_k$, where the ζ_i's are constants, map to the identical first-order term $N\zeta_1 \ldots \zeta_k$. For any general type G^i and variable x_j^i, $\mu(G^i x_j^i) = x_j^i$. That is, tree variables map to first-order variables. $\mu(\perp) = \perp$. Recursively, we map $N\tau_1 \ldots \tau_k$ to $N\tau_1' \ldots \tau_k'$, where for each j, $1 \leq j \leq k$,

$$\tau_j' = \begin{cases} \tau_j & \text{if } \tau_j \text{ is a constant} \\ \mu(\tau_j) & \text{if } \tau_j \text{ is a parse tree} \end{cases}$$

Lemma 20 For each type B, the mapping $\mu: \hat{\mathcal{T}}(B) \rightarrow FOT_B$ is an injection and preserves the ordering $\sqsupseteq$: if $\tau_1 \sqsupseteq \tau_2$ then $\mu(\tau_1) \sqsupseteq \mu(\tau_2)$. ∎

Recall the following definitions for first-order terms. A *unifier* for a pair of terms τ_1, τ_2 is a substitution θ such that $\theta(\tau_1) = \theta(\tau_2)$. A unifier θ for τ_1 and τ_2 is a *most general unifier* (*mgu*) if, for any other unifier θ' of τ_1 and τ_2, $\theta(\tau_1) \sqsupseteq \theta'(\tau_1)$. The binary operation[1] $\sqcap$ on FOT is defined as follows:

1. If τ_1 or τ_2 is $\perp$, then $\tau_1 \sqcap \tau_2 = \perp$.
2. If any variable occurs in both τ_1 and τ_2, then let τ_1' be a variant of τ_1 such that τ_1' and τ_2 share no variables; $\tau_1 \sqcap \tau_2 = \tau_1' \sqcap \tau_2$.
3. If τ_1 and τ_2 are not unifiable, then $\tau_1 \sqcap \tau_2 = \perp$.
4. Else let θ be a *mgu* of τ_1 and τ_2; $\tau_1 \sqcap \tau_2 = \theta(\tau_1) = \theta(\tau_2)$.

On $FOT/_{\equiv}$, the operation $\sqcap$ is defined: $[\tau_1] \sqcap [\tau_2] = [\tau_1 \sqcap \tau_2]$; this is well defined, since $\tau_1 \sqcap \tau_2$ is unique up to variants.

A similar definition could be given directly for trees over $\hat{\mathcal{T}}$ (based on the subsumption ordering $\sqsupseteq$ for trees), but it is convenient simply to refer to the corresponding operations on FOT. This is possible because $\mu(\hat{\mathcal{T}}(B))$ is closed under $\sqcap$:

[1]The $\sqcap$ is not strictly an "operation" because the variant τ_1' (item 2 below) is not uniquely defined. It is an operation on $FOT/_{\equiv}$, however.

Lemma 21 Let τ_1 and τ_2 be trees in $\hat{T}(B)$. There exists an element $\tau \in \hat{T}(B)$ (either a tree or $\perp$), unique up to variants, such that $\mu(\tau) = \mu(\tau_1) \sqcap \mu(\tau_2)$. ∎

Definition 22 Let τ_1 and τ_2 be arbitrary elements of $\hat{T}(B)$. Their meet is given by

$$\tau_1 \sqcap \tau_2 = \mu^{-1}(\mu(\tau_1) \sqcap \mu(\tau_2)).$$

On $\hat{T}(B)/_{\equiv}$, $[\tau_1] \sqcap [\tau_2] = [\tau_1 \sqcap \tau_2]$.

Theorem 23 $\hat{T}(B)/_{\equiv}$, partially ordered by $\equiv$, is a meet semilattice whose minimum element is $\perp$ and whose meet operation is effectively computable by $\sqcap$.

PROOF: By Lemma 20, μ is an order-isomorphism between $\hat{T}(B)$ and it image $\mu(\hat{T}(B))$ under μ. Clearly $\mu(\tau_1) \equiv \mu(\tau_2)$ iff $\tau_1 \equiv \tau_2$. By Lemma 21, $[\tau_1] \sqcap [\tau_2]$ exists for any pair of trees in $\hat{T}(B)/_{\equiv}$, and is a greatest lower bound of $[\tau_1]$ and $[\tau_2]$ by the corresponding property for first-order terms. ∎

The Join Operation

$\hat{T}(B)/_{\equiv}$ is not a lattice because there may not exist any tree $\tau_1 \sqcup \tau_2$ that subsumes both τ_1 and τ_2. For example, if B is a special type, we cannot join $B\,c_1$ and $B\,c_2$ if c_1 and c_2 are distinct constants, because there is no variable of type B. For a general type G^i, however, $G^i x_1^i$ subsumes both $G^i\,c_1$ and $G^i\,c_2$. This is the intuition behind the fact that $\hat{T}(G^i)/_{\equiv}$ is a lattice. However we cannot define $\sqcup$ so easily as we did $\sqcap$—simply by mapping over to first-order terms—because subtrees may not join. For example, when we join $G^i\ B\ c_1$ and $G^i\ B\ c_2$ as trees, we cannot simply join the two subtrees $B\ c_j$ and then attach the result to a root labeled G^i, as we would for the corresponding first-order terms.

Definition 24 We define the binary operation $\tau_1 \sqcup \tau_2$ on $\hat{T}(G^j)$ as follows:

- If $\tau_1 = \perp$ then $\tau_1 \sqcup \tau_2 = \tau_2$.
- If $\tau_2 = \perp$ then $\tau_1 \sqcup \tau_2 = \tau_1$.
- If any variable occurs in both τ_1 and τ_2, then let τ_1' be a variant of τ_1 such that τ_1' and τ_2 share no variables; $\tau_1 \sqcup \tau_2 = \tau_1' \sqcup \tau_2$.
- Otherwise $\tau_1 \sqcup \tau_2 = \mathbf{sup}(\tau_1, \tau_2)$ where **sup** is computed by the algorithm in Figure 1.

Lemma 25 For any $\tau_1, \tau_2 \in \hat{T}(G^i)$, $\tau = \tau_1 \sqcup \tau_1$ is a least common generalization of τ_1 and τ_2. That is, $\tau \sqsupseteq t_1$, $\tau \sqsupseteq \tau_2$, and for any τ' such that $\tau' \sqsupseteq \tau_1$ and $\tau' \sqsupseteq \tau_2$, $\tau' \sqsupseteq \tau$.

For every general type G^j, let ϕ_{G^j} be an arbitrary injection from all pairs of trees in $\hat{\mathcal{T}}(G^j) - \{\perp\}$ to the tree variables $\{G^j x_1^j, G^j x_2^j, \ldots\}$. **fail** is a new symbol unique to this algorithm.

Algorithm 1 $\mathbf{sup}(\tau_1, \tau_2)$:

Input: A pair (τ_1, τ_2) of parse trees such that no variable occurring in τ_1 occurs in τ_2.
Output: A tree, or **fail**.
Procedure:

Case:

1. τ_1 or τ_2 is a tree of the form $Bc_1, \ldots, c_k$, where B is a special type and the c_i's are constants: if $\tau_1 = \tau_2$, then return τ_1. Else return **fail**.
2. τ_1 or τ_2 is a tree of the form $G^i\alpha$, where α is a constant or a variable: if both τ_1 and τ_2 have root G^i, then return $\phi_{G^i}(\tau_1, \tau_2)$. Else return **fail**.
3. Otherwise, let $\tau_1 = R_1 U_1^1, \ldots, U_{k_1}^1$ and $\tau_2 = R_2 U_1^2, \ldots, U_{k_2}^2$, where the U's are subtrees or constants.

   Case:

   3.1 $R_1 = R_2 = G^i$, where G^i is a general type (and hence $k_1 = k_2 = 1$):

   3.11 If $U_1^1 = U_1^2$, then return τ_1.

   3.12 Else if U_1^1 and U_1^2 are both trees and if $\mathbf{sup}(U_1^1, U_1^2) \neq$ **fail**, then return $G^i \cdot \mathbf{sup}(U_1^1, U_1^2)$.

   3.13 Else return $\phi_{G^i}(\tau_1, \tau_2)$.

   3.2 $R_1 = R_2 = B$, where B is a special type (and hence $k_1 = k_2$):

   3.21 For all j, $1 \leq j \leq k_1$, let $V_j = U_j^1$ if $U_j^1 = U_j^2$, or $\mathbf{sup}(U_j^1, U_j^2)$ if U_j^1 and U_j^2 are both trees, or **fail** otherwise.

   3.22 If $V_j \neq$ **fail** for all j, $1 \leq j \leq k_1$, then return $B \cdot V_1, \ldots V_{k_1}$. Else return **fail**.

   3.3 Otherwise, return **fail**.

Figure 1: The **sup** algorithm.

PROOF: Observe first that $\sqcup$ is an operation on $\hat{\mathcal{T}}(G^i)$—when computing $\tau_1 \sqcup \tau_2$ according to Definition 24, the result is in $\hat{\mathcal{T}}$, and is never **fail**.

In the case where either τ_1 or τ_2 is $\perp$, the proof is trivial. Otherwise, let us define the *depth* of a parse tree τ to be 0 for tree constants $(Bc_1, \ldots, c_k)$ and tree variables $(G^i x^i_j)$, and $1 + \max_{\{1 \le j \le k\}} depth(\tau_j)$ when $\tau = B\tau_1, \ldots, \tau_k$. The proof is by induction on $d = \min(depth(\tau_1), depth(\tau_2))$.

If either τ_1 or τ_2 has depth zero (cases 1 and 2 in the **sup** algorithm), it is easy to see that **sup** returns **fail** if the two trees have no common generalization, and a least common generalization otherwise.

For inductive purposes, assume that for every pair of trees τ'_1 and τ'_2 at least one of which has depth no greater than d, $\mathbf{sup}(\tau_1, \tau_2)$ returns a least common generalization if one exists, or **fail** otherwise. Suppose, without loss of generality, that τ_1 has depth $d+1$ and that τ_2 has depth at least $d+1$. Let $\tau_i = R_i U^i_1, \ldots, U^i_{k_i}$ $(i = 1, 2)$ as in step 3. If $R_1 \neq R_2$, there is clearly no common generalization, and the algorithm correctly returns **fail** (step 3.3). If $R_1 = R_2$, then necessarily $k_1 = k_2 \equiv k$ according to the arity requirements of the grammar.

Consider cases 3.1 and 3.2 where each pair of U^i_j's $(j = 1, 2)$ is an identical pair or one having a common generalization. Let $B = R_1 = R_2$, and $\tau = B \cdot \overline{U_1}, \ldots, \overline{U_k}$, where $\overline{U_j} = U^1_j$ for an identical pair or $\mathbf{sup}(U^1_j, U^2_j)$ otherwise. We argue that τ is a common generalization of τ_1 and τ_2. By the inductive hypothesis, $\overline{U_j}$ is a least common generalization of U^1_j and U^2_j; hence there are unifiers θ^1_j and θ^2_j (for $1 \le j \le k$) such that $\theta^i_j(\overline{U_j}) = U^i_j$. Let $\theta^1 = \theta^1_1 \circ \ldots \circ \theta^1_j$, the composition of all the θ^1_j's, and similarly for θ^2. We claim that $\theta^1(\tau) = \tau_1$ and $\theta^2(\tau) = \tau_2$. To see this, suppose a variable $G^p x^p_q$ occurs in two or more of the $\overline{U_j}$'s. Since ϕ_{G^p} is an injection, the two pairs of terms that gave rise to $G^p x^p_q$ must have been identical. Thus where the domains of the substitutions θ^1_j (for $1 \le j \le k$) agree, their codomains also agree, i.e., the same variables are mapped to the same values. Hence

$$\begin{aligned} \theta^1(B \cdot \overline{U_1}, \ldots, \overline{U_k}) &= B \cdot \theta^1_1(\overline{U_1}), \ldots, \theta^1_k(\overline{U_k}) \\ &= \tau_1, \end{aligned}$$

and, similarly, $\theta^2(\tau) = \tau_2$. τ is thus a common generalization of τ_1 and τ_2. Let τ' be another common generalization. Either B, the root of τ_1, is a general type G^i and $\tau' = G^i x^i_r$ for some variable x^i_r, or $\tau' = B\overline{V_1}, \ldots, \overline{V_k}$ for some subtrees $\overline{V_j}$ $(1 \le j \le k)$. In the former case, it is clear that $\tau' \sqsupseteq \tau$. In the latter, we know that $\overline{V_j} \sqsupseteq \overline{U_j}$ for each j, by the inductive hypothesis, and, again, $\tau' \sqsupseteq \tau$, It follows that τ is a least common generalization.

If, in cases 3.1 and 3.2, there is some j such that $\mathbf{sup}(U^1_j, U^2_j) = \mathbf{fail}$, then by the inductive hypothesis the U^i_j's have no common generalization. Thus there is no term $\tau' = B\overline{U_1}, \ldots, \overline{U_k}$ such that $\tau' \sqsupseteq \tau_1$ and $\tau \sqsupseteq \tau_2$. In case 3.2, **sup** correctly returns **fail**. In case 3.1, however, where B is a general type G^i, there is a generalization of the form $G^i x^i_r$, and such a generalization is returned by **sup**. Any other generalization τ' must also be a tree variable of type G^i, whereupon $\tau' \sqsupseteq \tau$.

Thus the inductive hypothesis holds for depth $d+1$ as well, and the proof is complete. ∎

As with the meet, we define $[\tau_1] \sqcup [\tau_2]$ to be $[\tau_1 \sqcup \tau_2]$, which is easily shown to be well defined on $\hat{T}(G^i)/_{\equiv}$.

Lemma 25, together with Theorem 23, gives us:

Theorem 26 $\hat{\mathcal{T}}(G^i)/_{\equiv}$, partially ordered by $\sqsupseteq$ is a lattice whose meet ($\sqcap$) and join ($\sqcup$) operations are effectively computable.

To argue that $\hat{\mathcal{T}}(G^i)/_{\equiv}$ is a *complete* lattice, we note that the ordering $\sqsubset$ is *Noetherian:* for any $[\tau]$, there exist only finite chains

$$[\tau] \sqsubset [\tau_1] \sqsubset \ldots \sqsubset [x_1^i].$$

Thus every subset of $\hat{\mathcal{T}}(G^i)/_{\equiv}$ has a maximum element in $\hat{\mathcal{T}}(G^i)/_{\equiv}$, and completeness then follows from basic lattice theory (e.g., [7, Chap. 1]) .

To summarize the main result of this section:

Theorem 27 Let B be a non-terminal symbol of a typed-term grammar. With the adjunction of a unique least element $\perp$, the set $\mathcal{L}(B)$ of terms generated by B, modulo equivalence under variants ($\equiv$) and partially ordered by subsumption ($\sqsupseteq$), is a meet semilattice. For a general type G^i, $\mathcal{L}(G^i)$ is a complete lattice. Finally, the meet and join operations on terms are effectively computable.

Non-deterministic Term-Rewriting Systems

We now define a class of term-rewriting systems over a typed-term algebra. TRS's are an active research area of theoretical computer science and have already been applied to machine learning (e.g., [18, 17]). Mooney [24] has applied them to analytical learning as an alternative to predicate logic. See [3] for a recent survey of general research on TRS's. For our purposes, a TRS enables us to express our learning algorithms in a form applicable to many computational models, including logic programming and lambda calculus.

The term-rewriting systems that we shall use are non-deterministic in that, of all rewrite rules that may be applicable at any stage of the computation, the system always chooses a rule that ultimately leads to a successful computation, unless no such rule exists. In effect, the assumption of non-determinism abstracts away all of the backtracking search that occurs in an actual, deterministic system. This is appropriate, since our analytical learning algorithms learn from the fruits of a successful search. Further, it is by focusing on non-deterministic term-rewriting systems that we are able to express our analytical learning algorithms in a general form applicable to many computational models. These models differ widely with regard to the mechanisms available for removing non-determinism. Hence we would lose this generality if we focused only on deterministic models.

We define a *non-deterministic, typed-term rewriting system* (*NTTRS*) as follows. Starting with a typed-term language (including the subsumption ($\sqsupseteq$) and meet ($\sqcap$) relations on the terms of that language), we add a *rewriting* relation, as follows:

- $\mathcal{L}(G^0)$, the set of terms generated from the principal general type G^0, is interpreted as a set of *configurations* (or *states*) of the system.

- $\Re$ is a recursive set of *rewrite rules* (*rules* for short) of the form $\langle \alpha, \beta \rangle$, where both α and β are in $\mathcal{L}(B)$ for some type B. B is called the *type of the rule*. A rule may have more that one type. $\Re$ is closed under substitution: for any substitution θ, $\langle \theta(\alpha), \theta(\beta) \rangle \in \Re$ if $\langle \alpha, \beta \rangle \in \Re$.
- A *rewriting step* (or *step*) is a binary relation, written $\Rightarrow$, on $\mathcal{L}(G^0)$. $\tau_1 \Rightarrow \tau_2$ iff:
  - $\tau_1, \tau_2 \in \mathcal{L}(G^0)$;
  - $\langle \alpha, \beta \rangle \in \Re$ is a rule (let B be the rule type);
  - ω is a position in τ_1 such that $\tau_1[\omega]$ is a subterm of type B and $\tau_1[\omega] = \alpha$;
  - $\tau_2 = \tau_1[\omega \leftarrow \beta]$.

More succintly, we rewrite a configuration τ_1 by finding a type-B occurrence of α in τ_1 and replacing that subterm by β. By Lemma 12, the resulting term τ_2 is also a configuration. $\Rightarrow^*$ is the reflexive, transitive closure of $\Rightarrow$.

A configuration to which no rules can be applied is said to be *irreducible*. The general theory of term-rewriting systems (TRS's) deals with such issues as the existence and uniqueness, for each configuration τ, of an irreducible form τ' such that $\tau \Rightarrow^* \tau'$, but these issues are beyond the scope of our concerns. A difference between our definition of TRS's and one that is often used in the literature is that we do not require that $\mathcal{V}(\beta)$ (the set of variables occurring in β) be contained in $\mathcal{V}(\alpha)$. Logic programming is an example of a TRS where rules may introduce new variables on the righthand side of a rule.

Example 28 [LP] Refer back to the ttg for logic programming (Example 2) and to the simple program for addition (*plus*) in the introduction. A configuration is a goal, possibly conjunctive. This also includes the goal *true* and goal variables such as g_1. The rewrite rules are the Horn clauses. For example, the first rule,

$$plus(0, x_1, x_1) :- true \quad (i)$$

can be viewed as a schema of rules in which goals of the form $plus(0, \tau, \tau)$ (where τ is any sentence of type *Term*) can be rewritten to the goal *true*.

The configuration $\wedge(plus(s(0), 0, s(0)),\ true)$ can be rewritten by applying rule (ii) to the subterm $plus(s(0), 0, s(0))$. More precisely, the rule we are applying is rule (ii) in which the value 0 has been substituted for each of the variables x_2, x_3, and x_4. By closure under substitution, this is also a rule.[2]

After rewriting, we have the new configuration:

$$\wedge(plus(0, 0, 0),\ true).$$

[2]The process of instantiating the left side α of a rule so as to match a subterm in the goal and then applying the resulting, instantiated rule to the configuration is often called *demodulation*, to make a procedural distinction from *rewriting*.

This new configuration can in turn be rewritten by applying rule (i) (with $x_1 = 0$), yielding

$$\wedge(true,\ true).$$

At this point the configuration is irreducible. ∎

Example 29 [AP] Refer back to the ttg for the simple applicative language in Example 2. Configurations are any sentence of type *Expression*. Apart from the syntax, rewrite rules resemble the Curried functional patterns used in such programming languages as FL, ML, and Haskell [13]. To emphasize this similarity, we shall write rules in the form $\alpha = \beta$, instead of $\langle \alpha, \beta \rangle$. For example, a program for addition similar to the one discussed in the preceding example is as follows:

$$\begin{array}{lcll} ((plus\ 0)\ x_1)) & = & x_1 & (i) \\ ((plus\ (succ\ x_2))\ x_3) & = & (succ\ ((plus\ x_2)\ x_3)) & (ii) \end{array}$$

In this language, zero is represented by the constant 0, and the successor of a number n is represented by $(succ\ n)$.

Using the two rules above and their instantiations, we obtain the following sequence of rewrites:

$$((plus\ (succ\ 0))\ 0) \Rightarrow (succ\ ((plus\ 0)\ 0)) \Rightarrow (succ\ 0).$$

Although the AP term rewriting system is completely different from that of LP, one can see that the program for *plus* is essentially the same as the one in Example 28, and that there is a direct correspondence between rules in the two systems. ∎

Example 30 [LC] A configuration in our lambda-calculus language is any term of type *Expression*. The rewrite rules fall into two groups. The first group contains all rules of the form:

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

where Q and R are configurations and $[R/v]Q$ is the result of substituting R for the free occurrences of v, according to the standard rules for β-reductions:

- $[R/v]v = R$;
- $[R/v]v_1 = v_1$ if $v_1 \neq v$;
- $[R/v](E\ F) = ([R/v]E\ \ [R/v]F)$;
- $[R/v]\lambda v.E = \lambda v.\bar{E}$;
- $[R/v]\lambda v_1.E = \lambda v_1.[R/v]E$ if $v_1 \neq v$, and either v_1 does not occur free[3] in R or v does not occur free in E;

[3] *free*: outside the scope of a λv_1.

- $[R/v]\lambda v_1.E = \lambda v_2.[R/v][v_2/v_1]E$ if $v_1 \neq v$, v_1 occurs free in R, and v occurs free in E. (v_2 is a fresh variable occurring in neither R nor E.)

For example,

$$((\lambda v_1.(v_1\ x_1))\ x_2) \Rightarrow (x_2\ x_1)$$

is a rule in the first group.

The second group of rewrites—the group that constitutes the actual "program"—consists of a list of name–expression pairs: $\langle f, (\text{some expression})\rangle$. Such a rule indicates that an occurrence of the name f in the configuration can be replaced by the associated expression. Often called a δ-reduction, this is also a popular way to implement recursion in programming languages (like Lisp), since the replacing expression may also contain the name f. (Fixpoint combinators are another way to define recursion.)

To illustrate, let us recode the *plus* program from the preceding example in LC. "Zero" (0) is encoded by the expression $\lambda v\,.\,v$. We represent ordered pairs $[x_1, x_2]$ of objects x_1 and x_2 as

$$[x_1, x_2] \equiv \lambda v_1\,.\,((v_1\ x_1)\ x_2).$$

The integer "one" is represented by $[s, 0]$, "two" by $[s, [s, 0]]$, etc., where s is an abbreviation for 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 $(zero?\ (succ\ x)) \Rightarrow^*$ *false*.

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

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

The program for integer addition consists of the 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 $(plus\ (second\ v_1)\ v_2)$, that is, *plus* is applied recursively.

Once again, although the definition of the *plus* program in lambda-calculus is completely different from the logic programming and the applicative versions, the structures of all three *plus* programs are quite similar. ∎

Definition 31 A *computation* in a NTTRS is a finite sequence of configuration-position-rule triples,

$$[\tau_1, \omega_1, \langle \alpha_1, \beta_1 \rangle], \ldots, [\tau_n, \omega_n, \langle \alpha_n, \beta_n \rangle], [\tau_{n+1}, *, *] \tag{1}$$

where, for $1 \leq i \leq n$, an instance of the rule $\langle \alpha_i, \beta_i \rangle$ applied to τ_i at location ω_i yields τ_{i+1} ($*$ indicates "don't care"). The *path* of the computation consists of just the locations and the rules (ignoring the configurations).

Note that, for a given rule $\langle \alpha, \beta \rangle$ and subterm $\tau[\omega]$, there is a unique instance of the rule that rewrites the subterm, namely, $\langle \theta(\alpha), \theta(\beta) \rangle$, where $\theta(\alpha) = \tau[\omega]$. This is a consequence of Theorem 27.

A path is said to be *maximally general* if each rule of the path is maximally general. That is, if $\langle \alpha_i, \beta_i \rangle$ is the i'th rule in the path, there is no rule $\langle \alpha', \beta' \rangle$ of which $\langle \alpha_i, \beta_i \rangle$ is a substitution instance. In this paper, "path" will always be taken to mean "maximally-general path".

Example 32 [LP] Consider once again the simple logic program for *plus*. When the initial configuration is the goal $plus(s(0), s(0), s(s(0)))$, we obtain the following computation:

$$[plus(s(0), s(0), s(s(0))), \epsilon, (ii)] \Rightarrow [plus(0, s(0), s(0)), \epsilon, (i)] \Rightarrow [true, *, *].$$

In each step the position is ϵ, so the path of this computation is $[\epsilon, (ii)] \Rightarrow [\epsilon, (i)]$. ∎

Example 33 [AP] The *plus* program of Example 29 gives the following computation corresponding to $1 + 1 \Rightarrow^* 2$.

$$\begin{aligned} [(plus\ (succ\ 0))\ (succ\ 0)), \epsilon, (ii)] &\Rightarrow \\ [(succ\ ((plus\ 0)\ (succ\ 0))), 1 \cdot 3, (i)] &\Rightarrow \\ [(succ\ (succ\ 0)), *, *]. & \end{aligned}$$

Example 34 [LC] Following is a portion of a computation corresponding to $1 + 1 \Rightarrow^* 2$, using the program in Example 30. In the sequence, successive configurations are shown. To save space, multiple steps have been combined. To the right of each configuration is an indication of what rules were applied: $[\beta^+]$ refers to one or more β-rewrites, $[plus]$ signifies a substitution for the name *plus*, etc. The subterm that is rewritten is underlined.

$$\begin{aligned} &((\underline{plus}\ [s,0])\ \ [s,0])) && [plus] \\ &\underline{((\lambda v_1 \,.\, \lambda v_2 \,.\, (((zero?\, v_1)\, v_2)\, (succ\, ((plus\, (second\, v_1))\, v_2)))\, [s,0])\, [s,0])} && [\beta^+] \\ &(\underline{(zero?\, [s,0])}\, [s,0])\, (succ\, ((plus\, (second\, [s,0]))\ [s,0])) && [zero?, \beta^+] \\ &\underline{(false\, [s,0])\, (succ\, ((plus\, (second\, [s,0]))\ [s,0]))} && [\beta^+] \\ &(succ\ ((plus\, \underline{(second\, [s,0])})\ [s,0])) && [second, \beta^+] \end{aligned}$$

$$(succ\ ((\underline{plus}\ 0)\ [s,0])) \qquad [plus]$$

... (several steps omitted here)

$$\underline{(succ\ [s,0])} \qquad [\beta^+]$$

$$[s,[s,0]]$$

Thus over this path we find that $(plus\ [s,0])\ \ [s,0]) \Rightarrow^* [s,[s,0]]$, as expected. ∎

In the three preceding examples, the *plus* programs are quite similar, both in the way they represent naturals and in the recursive definition of the *plus* function. The *paths*, however, are not all similar. Whereas the LP and AP paths for "1 + 1 = 2" are easily comparable, the LC path is very different, since the TRS rules are quite distinctive. In the next two sections we present two analytical learning algorithms, AL-1 and AL-2. The question we should anticipate is this: *will the results of the learning algorithms be comparable in all three TRS's, or will the results be difficult to relate, especially in the LC system* vis à vis *the other two*? The answer to this question gives us insight into the nature of the learned information. For, if the learned structures are similar in the LP and AP cases but not in the LC case, then what is learned pertains more to the path of the computation than to the semantics of what it is computing. Conversly, if the learned information is similar in all three languages, the learning algorithm is acquiring semantic concepts rather than syntactical or operational ones.

The AL-1 Algorithm

If, in Example 32, our initial configuration had been $plus(s(0), s(s(0)), s(s(s(0))))$ or $plus(s(0), 0, s(0))$, the total computation would have followed the identical path. It is not difficult to see that $plus(s(0), x_1, s(x_1))$ is the most general goal whose computation follows this path. In some sense, once we have proved the latter goal, we get all the former goals almost "for free", since they are just instances. This is the idea behind the well-known algorithm known as EBG or goal regression. We shall formalize this process for NTTRS's and call the resulting algorithm *AL-1*. The new name is justified, since new considerations arise in the more general setting of TRS's that are irrelevant to the special case of logic programming, as the next example illustrates.

Given a program and a path for that program, we should like to determine the most general configuration that can be rewritten using that path. Unfortunately, no such configuration exists, in general. The next example shows why.

Example 35 [LC] For purposes of this example let us modify the grammar in Example 4 by replacing the fifth production rule as follows:

Expression1 → λ *Lambda-param Expression.*

Algorithm 2 (Stretch)

Input: Configurations τ_1 and τ_2, with $\tau_1 \sqsupseteq \tau_2$;
A position $\omega \in \Omega(\tau_2)$.
Output: A configuration τ_{12} such that $\tau_1 \sqsupseteq \tau_{12} \sqsupseteq \tau_2$ and $\omega \in \Omega(\tau_{12})$.
Procedure:

1. If $\omega \in \Omega(\tau_1)$, then return τ_1.
2. Else let $\bar{\omega}$ be the longest prefix of ω such that $\bar{\omega} \in \Omega(\tau_1)$ and the node at location $\bar{\omega}$ is labeled by a general type, say G^j.
   Remark: The same general type G^j necessarily occurs at position $\bar{\omega}$ of the parse of τ_2, and $\tau_1[\omega] = G^j\ x_i^j$ for some i. See text.
3. Compute *Expand*$(\tau_2, \bar{\omega})$.
   Remark: *Expand* computes a replacement for the term at position $\bar{\omega} \in \Omega(\tau_1)$, consisting of an appropriate term of the same type. See Lemma 37.
4. Let θ be the substitution that maps x_i^j to *Expand*$(\tau_2, \bar{\omega})$ and maps other variables to themselves; let $\tau' := \theta(\tau_1)$.
5. Return $\tau_{12} = \mathit{Stretch}(\tau', \tau_2, \omega)$.

Figure 2: The *Stretch* algorithm.

(The dot separating the parameter from the expression has been omitted.) Now consider the following two non-unifiable LC configurations:

$$\tau_1 : \quad \lambda\ v_1\ \underline{(\ \lambda\ v_2\ v_2\ \ x_1\ )}$$
$$\tau_2 : \quad (\ \mathit{succ}\ \underline{(\ \lambda\ v_2\ v_2\ \ x_1\ )}\ )$$

The underlined expression, $(\ \lambda\ v_2\ v_2\ \ x_1\ )$, can be rewritten with a β-reduction rule to x_1. Moreover, in both τ_1 and τ_2, this term occurs at the same location, $\omega = 1 \cdot 3$. Thus, in each configuration, the same path of length one can be used to rewrite this underlined lambda expression to x_1. But what is the most general expression such that this rule can be applied at position $1 \cdot 3$? By the **sup** algorithm, $\tau_1 \sqcup \tau_2 = x$, where x is a fresh variable of type *Expression*. Because position ω does not occur in the term x, the path used to reduce τ_1 and τ_2 does not apply to their least generalization. Hence no configuration that subsumes both τ_1 and τ_2 contains the redex $(\lambda v_2\ v_2\ \ x_1)$ at position $\omega = 1 \cdot 3$, and there is no most-general configuration to which the path applies. ∎

If we cannot look for the most general configuration that can be rewritten along a given path, we can instead determine the most general configuration that *both* is rewritable along the path *and* subsumes τ_0, where τ_0 is an *explanation* (or *example*)— a ground configuration together with a computation along the path.

The AL-1 algorithm does just this for a general NTTRS. Before presenting it, however, we need to describe a procedure, *Stretch*, that plays an essential role in the AL-1 algorithm. *Stretch* takes two configurations τ_1 and τ_2, where $\tau_1 \sqsupseteq \tau_2$, and a position $\omega \in \Omega(\tau_2)$, and returns a maximal configuration τ_{12} such that $\tau_1 \sqsupseteq \tau_{12} \sqsupseteq \tau_2$ and $\omega \in \Omega(\tau_{12})$. Later we shall argue that this configuration is unique up to variants.

If $\omega \in \Omega(\tau_1)$ then we can simply let $\tau_{12} = \tau_1$ and stop. Otherwise let $\bar{\omega}$ be the longest prefix of the integer string ω such that $\bar{\omega}$ exists as a location in τ_1 and such that the node at location $\bar{\omega}$ in the parse of τ_1 is a non-terminal, say, B. Such an $\bar{\omega}$ surely exists, since, if nothing else, the empty string ϵ is a prefix of ω, and the node at location ϵ is labeled by the start symbol G^0. The corresponding node in the parse of τ_2 must be labeled with the same non-terminal B, since $\tau_1 \sqsupseteq \tau_2$. Furthermore the tree rooted at location $\bar{\omega}$ in the parse of τ_1 *must be a tree variable the form* $G^j x_i^j$ for some j, since for any other tree, either $\bar{\omega}$ is not the longest matching prefix of ω, or $\omega = \bar{\omega}$, or $\tau_1 \not\sqsupseteq \tau_2$, contrary to assumption. Let $G^j \rightarrow B_1$ be the production corresponding to the node at location $\bar{\omega}$ in τ_2. We replace the configuration τ_1 by a new configuration τ_1' whose parse tree is derived from that of τ_1 by replacing each occurrence of the subtree $G^j \cdot x_i^j$ (including the one at $\bar{\omega}$) by the tree $G^j \cdot B_1$. With a non-terminal leaf node B_1, this is, of course, an incomplete parse tree. But a simple algorithm can be invoked to expand B_1 to its unique most general parse subtree in $\mathcal{L}(B_1)$ while preserving the $\sqsupseteq$ relation to the parse of τ_2. Having thus "stretched" the term τ_1 to τ_1', we repeat the procedure until the resulting configuration contains the required location ω.

The algorithm *Stretch* is given in detail in Figure 2; the subroutine *Expand* is given in Figure 3.

Example 36 [LC] We illustrate the result of applying *Stretch* to two terms and a given position, following the steps in the *Stretch* and *Expand* procedures. Let $\tau_1 = x_1$, a maximal lambda term. Let $\tau_2 = \lambda v_1.\underline{(\lambda v_2.v_2\ x_7)}$ and $\omega = 1\ 4$ (the term at this position in τ_2 is underlined). The parse tree, $tree(\tau_1)$, is: *Expression* x_1. The tree, $tree(\tau_2)$, is:

Expression Expression1 λ *Lambda-param* v_1 . *Expression Expression1* (etc.).

ω does not occur in τ_1, and the maximal prefix $\bar{\omega}$ is ϵ. We therefore replace τ_1 by the term τ_1' with the incomplete parse tree, *Expression Expression1.* (This is the result of step 1 in *Expand.*) *Expression1* is a special type, so we expand it (*Expand*, step 2.3) in such a way as to unify with the parse tree for τ_2:

Expression Expression1 λ *Lambda-param* . *Expression*

Since the underlined elements in this tree are non-terminal leaf nodes, this tree is still incomplete. We therefore continue expanding them by recursively calling *Expand*, with the result:

Expression Expression1 λ *Lambda-param* v_3 . *Expression* x_2 .

Here, step 2.2 has been applied; v_3 is a fresh variable of type *Lambda-param*, and x_2 is a fresh variable of type *Expression*.

The resulting configuration, $\lambda v_3.\underline{x_2}$, now has a term, x_2, at position $\omega = 1\ 4$; hence this term is output as τ_{12}, and the *Stretch* process is complete. Note that $\tau_1 \sqsupseteq \tau_{12} \sqsupseteq \tau_2$, as required. ∎

The next two technical lemmas characterize the relevant properties of *Stretch*.

Lemma 37 Let τ be any configuration and $\bar{\omega}$ a location in τ such that $\tau[\bar{\omega}] \in \mathcal{L}(B)$. $Expand(\tau, \bar{\omega})$ returns a most general term τ' satisfying these conditions: (1) $\tau' \in \mathcal{L}(B)$; (2) $\tau' \sqsupseteq \tau[\bar{\omega}]$; and (3) for all $i \in \mathbf{N}$, if the node at location $\bar{\omega} \cdot i$ in $tree(\tau)$ is labeled by a type or a constant (anything other than a variable), then the same label is assigned to the node at location i in $tree(\tau')$. That is, the tree τ' matches the structure of $\tau[\bar{\omega}]$, except that some subtrees of $\tau[\bar{\omega}]$ may be replaced in τ' by tree variables.

Note: Another way to state this is that

$$[Expand(\tau, \bar{\omega})] = \bigsqcup\{[\tau'] \mid \tau' \text{satisfies conditions (1) - (3)}\}.$$

Although $\mathcal{L}(B)/_{\equiv}$ is not a lattice, the conditions ensure that this join exists.

PROOF: By induction on the height h of $\tau[\bar{\omega}]$. For $h = 1$, the parse of $\tau[\bar{\omega}]$ is either $B \cdot c_1 \ldots c_k$ where each c_i is some constant, or possibly $G^j \cdot x_i^j$ if B is the general type G^j. In the former case, step 2.1 applies, providing a term τ' such that $\tau' = \tau$. In the latter, step 2.2 provides a term x_i^j consisting of a fresh variable. Either way, the requirements of the lemma are satisfied.

Inductively, for $h > 1$ the production in step 1 is of the form $B \rightarrow \zeta_1 \ldots \zeta_k$, where at least one of the ζ_i is a non-terminal. (Note that productions of the form $G^j \rightarrow B'$ are covered by this case.) Proceding again by cases, for those ζ_i that are constants, step 2.1 applies. For $\zeta_i = B'$, a special type, case 2.3 applies: *Expand* is called recursively, and by induction, the result is a term of type B', maximally general while remaining $\sqsupseteq \tau[\bar{\omega} \cdot i]$. ζ_i cannot be a variable, so the only remaining case is where ζ_i is a general type G^r. Instead of expanding G^r further, *Expand* supplies the most general term of the same type—namely, a fresh variable—so as to fulfill the maximum-generality requirement. Finally, the results of expansion of all these ζ_i are concatenated in step 3 into τ', a most general term that is $\sqsupseteq \tau[\bar{\omega}]$ that preserves types at the level below B. ∎

Lemma 38 Using the notation of Figure 2, $Stretch(\tau_1, \tau_2, \omega)$ computes a most general configuration τ_{12} such that $\tau_1 \sqsupseteq \tau_{12} \sqsupseteq \tau_2$, and $\omega \in \Omega(\tau_{12})$.

Note: Another way to state this is that

$$[Stretch(\tau_1, \tau_2, \omega)] = \bigsqcup\{[\tau_{12}] \mid \tau_1 \sqsupseteq \tau_{12} \sqsupseteq \tau_2 \text{ and } \omega \in \Omega(\tau_{12})\}.$$

Since the τ_i are configurations—and hence terms in $\mathcal{L}(G^0)$— the join exists, by Theorem 27.

PROOF: If the algorithm exits in step 1, the proof is trivial. Otherwise, let $\bar{\omega}$ be the proper prefix of ω found in step 2 of the initial call to the algorithm. Since $\tau_1[\bar{\omega}]$ is a term of type

Algorithm 2a (Expand)

Input: A configuration τ;
A position $\bar{\omega} \in \Omega(\tau)$ such that a non-terminal B occurs at position $\bar{\omega}$ in the parse of τ.
Output: A term in $\mathcal{L}(B)$.
Procedure:

1. Let $B \rightarrow \zeta_1 \ \ldots \ \zeta_k$ be the production in the grammar that corresponds to the node at position $\bar{\omega}$ in the parse of τ. Initialize: $P_i := \epsilon$ for all i, $1 \leq i \leq k$.
2. For each i from 1 to k, do:
   CASE:
   2.1 ζ_i is a constant c: $P_i := c$.
   2.2 ζ_i is a general type G^τ or a variable x_j^τ: $P_i := x_m^\tau$, where x_m^τ is a fresh variable unused in any expression so far in this or any calling routine.
   2.3 ζ_i is any other non-terminal B': $P_i := \mathit{Expand}(\tau, \bar{\omega} \cdot i)$.
3. Return the term $P_1 \cdot \ \ldots \cdot \ P_k$.

Figure 3: The *Expand* algorithm.

G^j for some j, $\bar{\omega} \cdot 1$ is also a prefix of ω. As noted in the discussion preceding the algorithm, $\bar{\omega}$ is in fact the maximum prefix of ω for which there is a term in τ_1 at location $\bar{\omega}$ of *any* type B; for if B were not a general type G^j, then either there would be a non-terminal at location $\bar{\omega} \cdot 1$ of $tree(\tau_1)$ (contradicting the maximality of $\bar{\omega}$) or there would be a terminal symbol at that location and we would have $\bar{\omega} = \omega$ (contradicting the assumption that $\bar{\omega}$ is a proper prefix of ω). Therefore, the term $\tau_1[\bar{\omega}]$ is a variable, which we'll call x_i^j. Moreover, the node at location $\bar{\omega} \cdot 1$ in $tree(\tau_2)$ is labeled by a non-terminal, B_1.

Let us analyze the configuration $\tau' = \theta(\tau_1)$ obtained in step 3 after substituting for x_i^j the result of the call to *Expand*. By Lemma 37, this term, $\tau'[\bar{\omega}]$, is a most general term in $\mathcal{L}(G^j)$ such that $\tau'[\bar{\omega}] \sqsupseteq \tau_2[\bar{\omega}]$ and the term at location $\bar{\omega} \cdot 1$ in τ' is at least as general as the corresponding term at location $\bar{\omega} \cdot 1$ in τ_2. Hence $\tau'[\bar{\omega} \cdot 1]$ is a term of type B_1, and so $\tau_1[\bar{\omega}] \sqsupset \tau'[\bar{\omega}] \sqsupseteq \tau_2[\bar{\omega}]$. Let θ be the substitution that maps x_i^j to $\tau'[\bar{\omega}]$ and other variables to themselves; then $\tau_1 \sqsupset \theta(\tau_1) = \tau' \sqsupseteq \tau_2$. τ' is then passed as the first argument to the recursive invocation of *Stretch* in step 4.

In successive calls to the *Stretch* algorithm (step 4), let τ_1', τ_2', ... be the sequence of configurations that are passed as the first argument. In particular, $\tau_1' = \tau_1$, and if the algorithm halts, the last term in this sequence is the final output. But we have just seen that this sequence of configurations is monotone decreasing with respect to the subsumption ordering $\sqsupseteq$; and the sequence of prefixes $\bar{\omega}$ of ω in step 2 increases in length by at least one on each successive call. Thus this sequence of calls must terminate. And since, by Lemma 37, each configuration τ_{k+1} in this sequence is maximally general for those terms having this sequence of positions, the final output has the required properties. ∎

With these preliminaries we can now develop the AL-1 algorithm. The algorithm applies to any NTTRS, but the system and language are "built in", i.e., are not input parameters to the algorithm. For this reason the algorithm is actually an algorithm schema, to be instantiated for any particular TRS. Note that, for expository purposes, the formal version of the algorithm in Figure 4 contains many more variables than are really required.

Input to the algorithm is a ground computation of length $n \geq 1$. The output from AL-1 is a new rewrite rule $\langle \hat{\alpha}, \hat{\beta} \rangle$, valid in the sense that $\hat{\alpha} \Rightarrow^* \hat{\beta}$ according to the existing rewrite rules $\mathfrak{R}$. Informally, this rule is sufficient to accomplish in a single rewriting step what the original ground computation achieved in n steps, and moreover is the most general such rule for the path. What one might use this new rule for is outside the scope of the algorithm, but the rule can be viewed as a "chunk" or "macro-operator", potentially useful for making the program more efficient. Such considerations belong to the deterministic computation model (and as such we shall discuss them later).

The procedure is quite simple. Let

$$[\tau_1, \omega_1, \langle \alpha_1, \beta_1 \rangle], \ldots, [\tau_n, \omega_n, \langle \alpha_n, \beta_n \rangle], [\tau_{n+1}, *, *]$$

be the computation, and let A_1 and B_1 be program variables, each with an initial value of x_1^0, a fresh variable of type G^0. For each step i in the path, we shall apply substitutions to A_i and rewrite rules to B_i at the same positions as are applied to the example configuration τ_i in the computation. The resulting rule will be $\langle A_{n+1}, B_{n+1} \rangle$.

Algorithm 3 (AL-1)

Input: A ground computation $[\tau_1, \omega_1, \langle \alpha_1, \beta_1 \rangle], \ldots, [\tau_n, \omega_n, \langle \alpha_n, \beta_n \rangle], [\tau_{n+1}, *, *]$ of length n. (We may assume that no two rules in the path have variables in common.)
Output: A rule $\langle \hat{\alpha}, \hat{\beta} \rangle$.
Procedure:

1. Initialize: $A_1 := x_1^0$, a fresh variable of type G^0. $B_1 := x_1^0$. The algorithm uses the additional variables A_i, AA_i, B_i, BB_i, ψ_i, and θ_i, for $1 \leq i \leq (n+1)$.
2. For $i := 1, \ldots, n$:
   - 2.1 $BB_i := Stretch(B_i, \tau_i, \omega_i)$.
   - 2.2 $\psi_i := mgu(BB_i, B_i)$.
   - 2.3 $AA_i := \psi_i(A_i)$.
   - 2.4 $\theta_i := mgu(BB_i[\omega_i], \alpha_i)$.
   - 2.5 $B_{i+1} := \theta_i(BB_i[\omega_i \leftarrow \beta_i])$. /* Apply the most general instance of the rule $\langle \alpha_i, \beta_i \rangle$ to BB_i at location ω_i */
   - 2.6 $A_{i+1} := \theta_i(AA_i)$.
3. Output $\langle \hat{\alpha}, \hat{\beta} \rangle = \langle A_{n+1}, B_{n+1} \rangle$.

Figure 4: The AL-1 algorithm.

Suppose we want to rewrite B_1 using the same rule $\langle \alpha_1, \beta_1 \rangle$ and position ω_1 as in the first step of the computation path. Since position ω_1 may not exist in B_1, we must first *stretch* it so that, if necessary, it acquires a subterm at position ω_1 while remaining as general as τ_1. Let BB_1 be the result of stretching B_1. Let ψ_1 be the substitution such that $\psi_1(B_1) = BB_1$. Since $\tau_1[\omega_1]$ is an instance of α_1 and $BB_1[\omega_1] \sqsupseteq \tau_1[\omega_1]$, it follows that $BB_1[\omega_1]$ *unifies with* (but may not be an instance of) α_1:

$$BB_1[\omega_1] \sqcap \alpha_1 \sqsupseteq \tau_1[\omega_1] \sqcap \alpha_1 = \tau_1[\omega_1].$$

Let θ_1 be a most general unifier of $BB_1[\omega_1]$ and α_1; we rewrite BB_1 to $\theta_1(BB_1[\omega_1 \leftarrow \beta_1])$, and call this term B_2[4]. A_2, in turn, is obtained by applying the same substitutions ψ_1 and θ_1 to A_1 that were necessary in order to rewrite B_1. This process then repeats for the remaining steps of the computation.

[4] In effect we are paramodulating BB_1.

Theorem 39 Refer to the notation of Figure 4, in which τ_1 is the initial ground configuration of the computation, and $\langle \hat{\alpha}, \hat{\beta} \rangle$ the resulting rewrite rule. Let π denote the path of the computation. Then $\hat{\alpha}$ is the most general configuration $\sqsupseteq \tau_1$ for which π is a valid path, and $\hat{\beta}$ is the configuration that results from rewriting $\hat{\alpha}$ along the path π. Equivalently,

$$\hat{\alpha} = \bigsqcup \{\tau \in \mathcal{L}(G^0) \mid \tau \sqsupseteq \tau_1 \text{ and there exists a computation using } \pi \text{ starting from } \tau\}.$$

PROOF: The proof of this theorem is not deep, but it does involve quite a lot of bookkeeping. Therefore we include only enough detail (hopefully) to convince the reader of the claims. We shall continue using the notation of Figure 4. Note especially that the "variables" A_i, B_i, AA_i, BB_i, and so forth are assigned only once during the algorithm, so that we can refer to their values unambiguously in the proof.

The proof is by induction on the length n of the path, with the following inductive hypothesis $H(k)$:

For all $i \leq k$,

1. There exists a substitution ζ_i such that $\zeta_i(A_i) = \tau_1$;
2. There exists a substitution η_i such that $\eta_i(B_i) = \tau_i$;
3. For all variables $v \in \mathrm{dom}(\zeta_i) \cap \mathrm{dom}(\eta_i)$, $\zeta_i(v) = \eta_i(v)$;
4. $A_i \Rightarrow^* B_i$ via a path consisting of the first $i-1$ steps of the path π of the input computation.

If this holds for all k, then it follows that the algorithm outputs a rewrite rule $\langle A_{n+1}, B_{n+1} \rangle$ such that $A_{n+1} \Rightarrow^* B_{n+1}$ over the n-step path π. That A_{n+1} is the most general such configuration follows directly from Lemma 38 and the fact that the ψ_i's and θ_i's in the algorithm are most-general unifiers.

The basis ($n = 1$) corresponds the the initial situation ($A_1 = x_1^0 = B_1$) and a path of length zero (no rewrites). It is clear that the substitutions $\zeta(x_1^0) = \eta(x_1^0) = \tau_1$ satisfy the hypothesis, and that A_1 "rewrites" to B_1 trivially over the empty path. Note also that A_1 is a most general configuration.

We now assume $H(i)$ and show that $H(i+1)$ holds. A_i becomes A_{i+1} by first computing $AA_i = \psi_i(A_i)$ and then applying θ_i to AA_i. Similarly, B_i becomes B_{i+1} by first computing $BB_i = \psi_i(B_i)$, and then rewriting BB_i using the i'th rule in the path.

We first argue that the four properties of the hypothesis are also true of AA_i and BB_i (i.e., they are preserved by the substitution ψ_i). To see this, note first that, by Lemma 38, $B_i \sqsupseteq BB_i \sqsupseteq \tau_i$, verifying property (2).

Let η' be the substitution such that $\eta'(BB_i) = \tau_i$. To verify property (1) for AA_i, we construct a substitution ζ' such that $\zeta'(AA_i) = \tau_1$. For this purpose, partition the variables $\mathcal{V}(A_i)$ in A_i into two groups: those that are in $\mathrm{dom}(\psi_i)$ and those that are not. If v belongs to the latter, then $\zeta_i(v)$ must be a ground term, and for each occurrence of v in A_i, $\zeta_i(v)$ matches this ground term at the same position in τ_1. The stretching process replaces some variables by terms containing fresh variables only, so no new occurrences of this variable v

will be created in AA_i by applying the substitution ψ_i. Therefore, by defining $\zeta'(v) = \zeta_i(v)$, we preserve property (1) for the subset of variables not in $\mathrm{dom}(\psi_i)$.

For property (3), if v occurs in B_i, then by hypothesis $\zeta_i(v) = \eta_i(v)$, a ground term, and since v is not in $\mathrm{dom}(\psi_i)$, $\eta_i(v) = \eta'(v) = \zeta'(v)$. Also, if v does not occur in B_i, (3) holds vacuously.

Now consider a variable $u \in \mathrm{dom}(\psi_i)$. $\psi_i(u)$ is a term containing only fresh variables. The substitution η' maps each of these fresh variables to the ground term at the corresponding location in τ_i, so $\eta_i(u) = \eta'(\psi(u))$. But $\zeta_i(u) = \eta_i(u)$ by property (3), so if ζ' is defined so as to agree with η' on these fresh variables, again we will retain both properties (1) and (3).

Thus, by composing the two cases above, we see that the substitution $\zeta' \equiv \zeta \circ \eta'$ maps AA_i to τ_1 and agrees with η' for variables in the intersection of their domains.

Finally, for property (4), we need to check that $\psi_i(A_i) \Rightarrow^* \psi_i(B_i)$ over the same path (the first $i-1$ steps of π). From the facts that $A_i \Rightarrow^* B_i$ over this path, $AA_i \sqsupseteq \tau_1$, and ψ_i introduces only fresh variables, this verification is straightforward but tedious, so we omit the details.

Next, we argue that $B_{i+1} \sqsupseteq \tau_{i+1}$. Recall that $B_{i+1} = \theta_i(BB_i[\omega_i \leftarrow \beta_i])$, where θ_i is the *mgu* of $BB_i[\omega_i]$ and α_i. By assumption, no variables are common to both α_i and BB_i, so $\mathrm{dom}(\theta_i) = \mathcal{V}(BB_i[\omega_i]) \cup \mathcal{V}(\alpha_i)$. We partition the variables in BB_i into those in $\mathrm{dom}(\theta_i)$ and those not in the domain. We construct a substitution η_{i+1} such that $\eta_{i+1}(B_{i+1}) = \tau_{i+1}$. Since $BB_i[\omega_i] \sqsupseteq \tau_i[\omega_i]$ and $\alpha_i \sqsupseteq \tau_i[\omega_i]$, it follows that

$$BB_i[\omega_i] \sqcap \alpha_i \sqsupseteq \tau_i[\omega_i].$$

But

$$\theta_i(BB_i[\omega_i]) = BB_i[\omega_i] \sqcap \alpha_i.$$

Thus there is a substitution ϕ such that $\phi \circ \theta_i = \eta'$ (see Figure 5). Thus if v is a variable in $\mathrm{dom}(\theta_i) \cap \mathcal{V}(BB_i)$, $\phi \circ \theta_i(v)$ is the ground term that occurs at the same location(s) in $\tau_i[\omega_i]$ as v does in $BB_i[\omega_i]$.

If v is not in $\mathrm{dom}(\theta_i)$ but occurs in BB_i, then from the above discussion we know that $\eta'(v)$ is the ground term that occurs at the same location(s) in τ_i as v does in BB_i. Since these two sets of variables are disjoint, $\eta' \circ \phi \circ \theta_i(BB_i) = \tau_i$.

Now,

$$\begin{aligned}
\tau_{i+1} &= \tau_i[\omega_i \leftarrow \phi \circ \theta_i(\beta_i)] \\
&= \eta' \circ \phi \circ \theta_i(BB_i[\omega_i \leftarrow \phi \circ \theta_i(\beta_i)] \\
&= \eta' \circ \phi \circ \theta_i(BB_i[\omega_i \leftarrow \beta_i]) \\
&\sqsubseteq \theta_i(BB_i[\omega_i \leftarrow \beta_i] \\
&= B_{i+1}.
\end{aligned}$$

To argue that $A_{i+1} \sqsupseteq \tau_1$, we proceed similarly. Partition $\mathcal{V}(AA_i)$ into variables in $\mathrm{dom}(\theta_i)$ and other variables. It is the former that are significant, so let v be in $\mathrm{dom}(\theta_i) \cap \mathcal{V}(AA_i)$. From the discussion above, we know that $\zeta'(v) = \eta'(v) =$ a ground term in τ_1 (and also in

τ_i). From Fig. 5, $\theta_i(v) \sqsupseteq \eta'(v)$, so $\theta_i(v) \sqsupseteq \zeta'(v)$. Hence $\zeta' \circ \theta_i = \zeta'$ (recalling that ζ' is a ground substitution), and

$$\begin{aligned} \tau_1 &= \zeta'(AA_i) \\ &= \zeta' \circ \theta_i(AA_i) \\ &= \zeta'(A_{i+1}). \end{aligned}$$

It is routine to show that that variables common to A_{i+1} and B_{i+1} are mapped to the same ground terms in τ_1 and τ_{i+1}, respectively, and that $A_{i+1} \Rightarrow^* B_{i+1}$ over the path consisting of the first i steps of the path in the example. Then the induction is complete, and with it, the proof. ∎

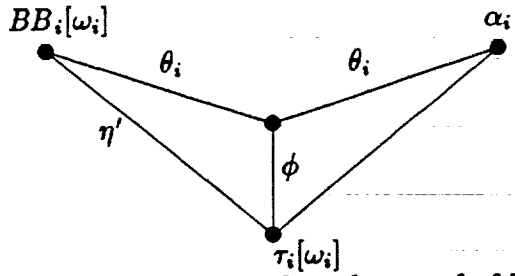

Figure 5: Substitutions used in the proof of Theorem 39.

Example 40 [LP] To illustrate AL-1 in a familiar setting, let us see how the AL-1 algorithm generalizes the example $plus(s(s(0)), 0, s(s(0)))$ using the logic program for *plus*. The proof of the example has three steps, shown in the first column of the table (the first line is the initial state). The numbers in parentheses are the rewrite rules (clauses) used in each of the steps. To avoid variable conflicts, a variant of each rule using fresh variables has been applied in each step.

The second and third columns show the values of the variables A_i and B_i after each step. The substitutions θ_i are shown in the fourth column. The calls to *Stretch* are all "no-ops" in this example since the rewrite positions ω_i in the computation are all ϵ. Thus the substitutions ψ_i are all identity functions. Finally, if we apply all the substitutions to the initial value x_1, we obtain, as output, the rule $plus(s(s(0), x_3, s(s(x_3))) :- true$. ∎

τ_i	*rule*	A_i	B_i	θ_i
$plus(s(s(0)), 0, s(s(0)))$	*(ii)*	g_1	g_1	
$plus(s(0), 0, s(0))$	*(ii)*	$plus(s(x_2), x_3, s(x_4))$	$plus(x_2, x_3, x_4)$	$g_1 := plus(s(x_2), x_3, s(x_4))$
$plus(0, 0, 0)$	*(i)*	$plus(s(s(x_5)), x_3, s(s(x_6)))$	$plus(x_5, x_3, x_6)$	$x_2 := s(x_5), x_4 := s(x_6)$
true		$plus(s(s(0)), x_3, s(s(x_3)))$	*true*	$x_5 := 0, x_6 := x_3$

Example 41 [AP] We apply AL-1 to the *plus* program in Example 29 and to the computation in Example 33. The steps are summarized in the following table, and the resulting rule is: $((plus\ (s\ 0))\ x_3) = (s\ x_3)$. ∎

τ_i	A_i	B_i	θ_i
$((plus\ (s\ 0))\ (s\ 0))$	x_1	x_1	
$(s\ ((plus\ 0)\ (s\ 0)))$	$((plus\ (s\ x_2))\ x_3)$	$(s\ ((plus\ x_2)\ x_3))$	$x_1 := ((plus\ (s\ x_2))\ x_3)$
$(s\ (s\ 0))$	$((plus\ (s\ 0))\ x_3)$	$(s\ x_3)$	$x_2 := 0$

Example 42 [LC] Let us apply AL-1 to the computation sketched in Example 34: $((plus\ [s,0])\ [s,0]) \Rightarrow^* [s,[s,0]]$. By analogy with the preceding example, we expect the result to be

$$\langle((plus\ [s,0])\ x_1),\ [s,x_1]\rangle$$

(or a variant thereof), and, indeed, this is the outcome. But because the path for the complete computation is so long, we shall follow only the first few steps.

The input configuration τ_1 is $((plus\ [s,0])\ [s,0])$, and the path is sketched in Example 34. We initially take our configuration to be x_1 (an expression variable).

The first step, a replacement for *plus*, occurs at a location not in x_1; so $B_1 = x_1$ is first stretched into $BB_1 = ((plus\ x_2)\ x_3)$. To apply the *plus* rule, we unify *plus* with itself (so θ_1 is an identify), and replace *plus* by its corresponding lambda expression (with fresh variables), to obtain

$$B_2 = ((\lambda v_1 \,.\, \lambda v_2 \,.\, (((zero?\, v_1)\, v_2)\, (succ\, ((plus\, (second\, v_1))\, v_2)))\, x_2)\ x_3)$$

and $A_2 = ((plus\ x_2)\ x_3)$.

The next several steps are β-reductions at locations already in B_2, so that stretching has no effect. Consider the first of these, the β-substitution of x_2 for the parameter v_1. The maximal rewrite rule applicable here is the β-reduction rule:

$$\begin{aligned}((\lambda v_{91} \,.\, \lambda v_{92} \,.\, (((x_{101}\ v_{91})\ v_{92})\ (x_{102}\ ((x_{103}\ (x_{104}\ v_{91}))\ v_{92})))\ x_{105}) \ \Rightarrow \\ \lambda v_{92} \,.\, (((x_{101}\ x_{105})\ v_{92})\ (x_{102}\ ((x_{103}\ (x_{104}\ x_{105}))\ v_{92}))).\end{aligned}$$

The result of applying this rule to B_2 is

$$B_3 = (\lambda v_2 \,.\, (((zero?\, x_2)\, v_2)\, (succ\, ((plus\, (second\, x_2))\, v_2)))x_3).$$

The cumulative result of just these two steps is the rule:

$$((plus\ x_2)\ x_3) \Rightarrow B_3.$$

The remaining steps are similar to the first two, with the final rule being:

$$((plus\ [s,0])\ x_3) \Rightarrow [s,x_3].\ \blacksquare$$

Finally, a simple but useful observation is that $A_1 \sqsupseteq A_2 \sqsupseteq \ldots \sqsupseteq A_{n+1}$, that is, the lefthand side $\hat{\alpha}$ of the final rule $\langle\hat{\alpha},\hat{\beta}\rangle$ becomes monotonically less general as the length of the path increases. Since $\hat{\alpha}$ contains the "pre-conditions" that must be satisfied before the new rule is applicable, it follows that the rule becomes less general as the length of the path—and in some sense, the amount of information in the computation—becomes larger. With AL-1, it seems that we learn less and less from more and more. One way to avoid this problem is discussed later when we consider deterministic rewriting systems.

A Counterexample: Elementary Formal Systems

Elementary formal systems (EFS's) [1, 2, 31] are a form of logic programming useful in learning formal languages. Briefly, an EFS is a Horn-clause logic programming language with string concatenation as the only function symbol. For example, let $\Sigma = \{a, b, c\}$ be an alphabet, and let x_1, x_2, ... be variables taking values in Σ^+. The clause

$$p(ax_1babx_2) :- true \quad (i)$$

says that any string of the form ax_1babx_2 (with nonempty strings substituted for the x_i's) has property p. Similarly, the clause

$$p(cx_3x_4) :- p(x_3bx_4) \quad (ii)$$

states that an instance of the string pattern cx_3x_4 has property p if the instance of x_3bx_4 with the same substitution for x_3 and x_4 also has property p. Combining these two clauses, we can prove $p(caababba)$, for example, by matching $x_3 = aaba$ and $x_4 = bba$ in the second clause, reducing the goal to $p(aababbba)$, and then matching $x_1 = a$ and $x_2 = bba$ in the first clause. The computation is as follows:

$$[p(caababba), \epsilon, (i)] \Rightarrow [p(aababbba), \epsilon, (ii)] \Rightarrow [true, *, *].$$

EFS's diverge from our definition of nondeterministic TRS's in an interesting way: because string concatenation obeys an associative equational theory, there may be more than one way to unify two terms. For example, in matching $p(caababba)$ to $p(cx_3x_4)$ above, we could also have taken $x_3 = aa$ and $x_4 = babba$. Doing so leads to a different proof of $p(caababba)$, over the same path. The computation is as follows:

$$[p(caababba), \epsilon, (i)] \Rightarrow [p(aabbabba), \epsilon, (ii)] \Rightarrow [true, *, *].$$

For each of these two proofs we can apply the AL-1 algorithm, and as a result *we derive two different rewriting rules*: $p(cax_5bax_6) :- true$ in the first case, and $p(cax_5abx_6) :- true$ in the second. Both are valid clauses in this theory, and yet neither is a variant of the other. We conclude that *Theorem 39 does not hold for EFS's.*

What property of EFS's keeps them from qualifying as typed-term rewriting systems? Clearly, the existence of multiple *mgu*'s for a pair of terms is part of the problem, but the Robinsonian nature of *mgu*'s is an inferred, not a defined, property of our typed-term languages. The actual reason is that, to admit associativity in string concatenation, the ttg for the language needs to be ambiguous. For example, to match the pattern x_1x_2 to the string aba, we must be able to parse aba as both $a \cdot (b \cdot a)$ and $(a \cdot b) \cdot a$.

This counterexample is interesting because it shows that the standard EBG algorithm as it is currently used in Machine Learning does not work as expected for at least one important family of computational languages. Two ways come to mind whereby we could bring EFS's within the scope of our nondeterministic term-rewriting systems. One is to adjoin

explicit rewrite rules for the associative theory—$(x_1 \cdot (x_2 \cdot x_3), (x_1 \cdot x_2) \cdot x_3)$, for example. The computational path would thereby record explicitly the steps taken to associate strings in a particular way for matching, and thereby implicitly record the unifiers used in the computation. The other way is to extend the definition of a computation so as to record explicitly the particular unifiers used in matching rules to examples; a computation would thus be a sequence of 4-tuples instead of 4-tuples. The AL-1 algorithm then remains essentially the same, except that the particular *mgu*'s computed in steps 2.2 and 2.4 would be determined by those used in the example. Theorem 39 also holds, with the understanding that the word "path" includes the *mgu*'s along with the rules and positions.

The AL-2 Algorithm

The formalism we have developed based on ttg's and TRS's has been useful for extending one algorithm to a large family of programming languages. Our belief, however, is that the formalism is useful in general for studying analytical learning algorithms. As evidence, we shall use the same framework to develop another analytical learning algorithm, AL-2. Like AL-1, the new algorithm learns from success, preserves the correctness of the program, and outputs new rules for potential inclusion in the knowledge base. Unlike AL-1, however, AL-2 modifies the *language* in which the rules are expressed. New constants and/or function symbols may be introduced in order to abbreviate frequently occurring terms and to shorten common sub-proofs. This resembles what humans do, for example, when we say "EBG" instead of "explanation-based generalization" or "prime" instead of "natural number divisible only by itself and one". Moreover, AL-1 and AL-2 can be used together or independently.

Example 43 [LP] Consider again the program for addition in Example 28. After using this program for addition many times, an observer might determine that the term $s(s(0))$ occurs sufficiently often that significant savings might well be sustained by shortening this term to just a single character, say "2". As a result, every time this term occurs in a computation, instead of writing seven characters, only one need be written. The symbol "2" is not now in the language, so the grammar must be modified in a straightforward way to generate this additional constant. Some goals, such as $plus(0, 2, s(s(0)))$, may then be correctly handled without any modifications to the program. Other goals, such as $plus(2, 0, 2)$, fail because the program is not designed to handle such terms. ∎

Example 44 [LC] In Example 30 we introduced several abbreviations in order to shorten the expressions we were working with. For example, we wrote 0 in place of $\lambda v_1.v_1$ and *false* in place of $\lambda v_1.\lambda v_2.v_2$. We treated these as abbreviations for human consumption only, but it is reasonable to consider how to incorporate these into the program so that, for example, $((plus\ 0)\ 0) \Rightarrow^* 0$. If one tried to carry out this computation as things stand, the subterm,

$$(0\ (\lambda v_2.\lambda v_3.v_2))$$

would soon occur. This can be rewritten only after replacing 0 by the equivalent lambda expression. ∎

The AL-2 algorithm takes a computation and an abbreviation, and returns a list of new rewrite rules which, now that they are available, enable us to do the computation— and some related generalizations of it—after substituting the abbreviated term into the initial configuration. Like AL-1, the input to the learning process is a successful computation, and the result of learning is a set of new rewrite rules.

Before describing the AL-2 algorithm in detail, let us consider simple examples of how it works.

Example 45 [LP] Suppose that we decide to abbreviate the term $s(s(0))$ by the new symbol "2". How should we modify the logic program to utilize this abbreviation? There is, of course, a trivial way to incorporate the abbreviation: provide a preprocessor that changes all occurrences of 2 in the input into $s(s(0))$, use the program as is, and then replace all $s(s(0))$ terms in the output by 2. But this clearly saves neither time nor space in the computation.

Here is a better way. When we try to satisfy the goal $plus(2, 0, x)$, using the program above, we run into trouble because 2 fails to unify with $s(x_2)$ in the second clause. Since 2 is just an abbreviation for $s(s(0))$, the clause

$$plus(2, x_3, s(x_4)) :- plus(s(0), x_3, x_4)$$

is clearly valid. We obtain it by instantiating x_2 to $s(0)$ and replacing the resulting occurrence of $s(s(0))$ on the left by 2. With the addition of this clause to the program, the family of goals $plus(2, x_1, s(x_2))$ can be solved in terms of the new symbol 2. ∎

Example 46 [LC] Suppose we wish to perform the computation $(zero?\ 0)$ without replacing 0 by $\lambda v_1.v_1$. We have no trouble with the first step—replacing *zero?* by its definition,

$$(zero?\ 0) \Rightarrow (\lambda v_2.(v_2\ true)\ 0),$$

—nor with the next, a β-reduction leading to $(0\ true)$. At this point, however, we are stuck. If we check how this step is done when 0 is written $\lambda v_3.v_3$, we see that the rule being applied is $\langle(\lambda v_3.v_3 v_4), v_4\rangle$. After introducing the abbreviation into this rule, we see that the only new rule we need in order to complete the computation is

$$\langle(0\ v_4), v_4\rangle.$$

The computation then concludes successfully, with a result of *true*. ∎

Definition 47 Let B be a type in a typed-term grammar. A *synonym of type* B is a pair (σ, τ), where σ is a symbol not in the grammar (terminal or otherwise) and $\tau \in \mathcal{L}(B)$ is a ground term of type B. We shall often refer to σ as simply a synonym for τ, and write $\sigma \sim_B \tau$. Normally, the type of the synonym is clear, and we omit the subscript B.

Example 48 [LP] In the preceding LP example, "2" is a synonym of type *Term* for $s(s(0))$. It is equally a synonym of type *Term1*, but in general it is preferable to consider it an element of the most general type possible in the grammar. ∎

The AL-2 algorithm outputs rewrite rules in which the new symbol σ occurs; consequently the grammar $\mathcal{G}$ must be extended to a new ttg $\mathcal{G}'$ in such a way that both σ and τ are in $\mathcal{L}(B)$. One way to accomplish this is as follows;

1. If B has arity one, add the production $B \rightarrow \sigma$ to the grammar.
2. Otherwise, make B have arity one by replacing all productions $B \rightarrow \ldots$ by $B' \rightarrow \ldots$, where B' is a new non-terminal, and then add the productions $B \rightarrow \sigma$ and $B \rightarrow B'$.

We assume henceforth that $\mathcal{G}$ has been so modified.

Definition 49 Let $\sigma \sim \tau$ be a synonym of type B and let ζ be a term in $\mathcal{L}(B')$ for some non-terminal B'. The *σ-abbreviation of ζ*, written $\zeta|_\sigma$, is obtained from ζ by replacing all type-B occurrences in ζ of the subterm τ by σ.

Lemma 50 For every non-terminal B', $\mathcal{L}(B')$ is closed under σ-abbreviations. ∎

Just as *Expand* is the operation upon which AL-1 is based, *Reunify* forms the basis for AL-2. Input to *Reunify* consists of a synonym $\sigma \sim_B \tau$ of some fixed type B, a term ζ, and a ground instance ζ_0 of ζ. The output is a term ζ' such that $\zeta \sqsupseteq \zeta' \sqsupseteq \zeta_0$ and $\zeta'|_\sigma \sqsupseteq \zeta_0|_\sigma$. Intuitively, *Reunify* finds an instance ζ' of ζ that is as general as ζ_0 both with and without the abbreviation σ. For example, when ζ is $plus(s(x), y, s(z))$ and ζ_0 is $plus(s(s(0)), s(s(0)), s(s(s(s(0)))))$, then $\zeta_0|_2$ is not an instance of $\zeta|_2$. However, $\zeta' = plus(s(s(0)), y, s(z))$ is an instance of ζ such that $\zeta' \sqsupseteq \zeta_0$ and $\zeta'|_2 \sqsupseteq \zeta_0|_2$.

The algorithm for *Reunify* is shown in Figure 6.

Lemma 51 In the notation in Figure 6, the output of *Reunify* is a least upper bound of all terms ζ' such that $\zeta \sqsupseteq \zeta' \sqsupseteq \zeta_0$ and $\zeta'|_\sigma \sqsupseteq \zeta_0|_\sigma$.

PROOF: The proof is by induction on n, the number of locations ω such that $(\zeta_0|_\sigma)[\omega] = \sigma$ and $(\zeta|_\sigma)[\omega] \not\sqsupseteq \zeta_0|_\sigma[\omega]$. Suppose $n = 0$. Then $\zeta' = \zeta$ and $\zeta = \zeta' \sqsupseteq \zeta_0$. We claim that, with $n = 0$, there is no location ω' such that $(\zeta|_\sigma)[\omega'] \not\sqsupseteq (\zeta_0|_\sigma)[\omega']$; it then follows that $\zeta|_\sigma \sqsupseteq \zeta_0|_\sigma$. Suppose such a location ω' exists. Since $\zeta \sqsupseteq \zeta_0$, there exists a substitution θ such that $\theta(\zeta) = \zeta_0$, and hence $(\theta(\zeta))[\omega']|_\sigma = \zeta_0[\omega']|_\sigma$. By assumption, however, $(\theta(\zeta|_\sigma))[\omega'] \neq (\zeta_0|_\sigma)[\omega']$. In other words, substituting the abbreviation σ for τ in ζ_0 has the effect that θ is no longer a unifier for the subterms at position ω'. The term $(\zeta|_\sigma)[\omega']$, therefore, must contain at least one variable x such that $\theta(x)$ is a subterm of τ. Let ω'' be the location of this occurrence of τ in ζ_0. Then both $(\zeta|_\sigma)[\omega''] \not\sqsupseteq (\zeta_0|_\sigma)[\omega'']$ and $(\zeta_0|_\sigma)[\omega''] = \sigma$. But then $n > 0$, contrary to our assumption. Thus no such ω' exists, and ζ satisfies the lemma.

Next, assume that the lemma holds for all terms for which $n \leq k$, and suppose ζ is a term such that $\zeta \sqsupseteq \zeta_0$ and $n = k + 1$. Let ω be the location chosen in step 2.1 and θ, the substitution in step 2.2. Since τ is ground, θ is unique. Clearly $\zeta \sqsupseteq \theta(\zeta)$, and $\theta(\zeta) \sqsupseteq \zeta_0$. When θ is applied to ζ, n decreases by at least one. Hence the term $\theta(\zeta)$ satisfies the inductive hypothesis, and the result of subsequent iterations of the algorithms is a term ζ' such that $\zeta \sqsupseteq \theta(\zeta) \sqsupseteq \zeta' \sqsupseteq \zeta_0$ and $\zeta'|_\sigma \sqsupseteq \zeta_0|_\sigma$, in accordance with the claim in the lemma.

Algorithm 4 (Reunify)

Input: A synonym $\sigma \sim_B \tau$.
Terms ζ and ζ_0, with $\zeta \sqsupseteq \zeta_0$ and ζ_0 ground.
Output: A term ζ' such that $\zeta \sqsupseteq \zeta' \sqsupseteq \zeta_0$ and $\zeta'|_\sigma \sqsupseteq \zeta_0|_\sigma$.
Procedure:

1. Initialize: $\zeta' := \zeta$.
2. While $\zeta'|_\sigma \not\sqsupseteq \zeta_0|_\sigma$, do:
   2.1 Let ω be a location such that $(\zeta'|_\sigma)[\omega] \not\sqsupseteq (\zeta_0|_\sigma)[\omega]$ and $(\zeta_0|_\sigma)[\omega] = \sigma \in \mathcal{L}(B)$.
   2.2 Let $\theta = mgu(\zeta'[\omega], \tau)$.
   2.3 $\zeta' := \theta(\zeta')$.
3. Output ζ'.

Figure 6: The *Reunify* algorithm.

To argue that ζ' is maximally general, suppose that another term $\zeta'' \sqsupseteq \zeta'$ can be found such that $\zeta \sqsupseteq \zeta'' \sqsupseteq \zeta_0$ and $\zeta''|_\sigma \sqsupseteq \zeta_0|_\sigma$. Note that the substitution θ such that $\theta(\zeta) = \zeta'$ is a ground substitution (it substitutes ground terms for some of the variables of ζ). Thus if $\zeta' \not\sqsupseteq \zeta''$, then there exists a variable x in $\mathrm{dom}(\theta)$ not in $\mathrm{dom}(mgu(\zeta, \zeta''))$. But we have seen that every variable in $\mathrm{dom}(\theta)$ occurs within a subterm $\zeta[\omega]$ of ζ such that $(\zeta|_\sigma)[\omega] \not\sqsupseteq (\zeta_0|_\sigma)[\omega]$. Thus $(\zeta''|_\sigma)[\omega] \not\sqsupseteq (\zeta_0|_\sigma)[\omega]$, a contradiction. Thus the only possibility is that $\zeta'' \equiv \zeta'$. ∎

Example 52 [LP] Suppose we invoke *Reunify* with the synonym $2 \sim s(s(0))$, and use it to specialize $\zeta = plus(s(x), y, s(z))$ so that the resulting term subsumes $\zeta_0|_\sigma = plus(2, 2, s(s(2)))$ after introducing the abbreviation. Initially $\zeta' = plus(s(x), y, s(z))$. Thus $\zeta'|_2 = \zeta'$. The subterm $s(x)$ occurs at a location in $\zeta'|_2$ that does not unify with the corresponding term ("2") in $\zeta_0|_2$. So we substitute $x := s(0)$ and replace ζ' with $plus(s(s(0)), y, s(z))$. Now $\zeta'|_2 \sqsupseteq \zeta_0|_2$, and the *Reunify* procedure terminates. ∎

The AL-2 algorithm takes a synonym $\sigma \sim \tau$ of some type B and a computation of length $n \geq 1$, and produces a set $\mathcal{Q}$ of new rules incorporating the synonym. The intention is that, by combining the rules in $\mathcal{Q}$ and $\mathfrak{R}$, one can subsequently perform calculations over the same path using the extended language, except that some of the rules $\langle \alpha_i, \beta_i \rangle$ in the path may be replaced by their counterparts from $\mathcal{Q}$.

The algorithm (Figure 7) is quite simple. The left-hand side of each rule along the path is reunified with the corresponding subterm of the configuration, using the synonym σ. The resulting substitution is applied to both sides of the rule; after introducing σ into the result, this pair is added to the set Q as a new rule.

Algorithm 5 (AL-2)

Input: A synonym $\sigma \sim \tau$ of type B.
A ground computation $[\tau_1, \omega_1, \langle \alpha_1, \beta_1 \rangle], \ldots, [\tau_n, \omega_n, \langle \alpha_n, \beta_n \rangle], [\tau_{n+1}, *, *]$ of length n.
Output: A set $\mathcal{Q}$ (perhaps empty) of rewrite rules.
Procedure:

1. Initialize: $\mathcal{Q} = \emptyset$.
2. For $i := 1, \ldots, n$:
   2.1 $\alpha_i' := Reunify(\sigma \sim \tau, \alpha_i, \tau_i[\omega_i])$.
   2.2 If $\alpha_i' \neq \alpha_i$, then
      2.21 $\theta_i := mgu(\alpha_i', \alpha_i)$.
      2.22 $\mathcal{Q} := \mathcal{Q} \cup \{\langle \alpha_i'|_\sigma, \beta_i'|_\sigma \rangle\}$, where $\beta_i' = \theta_i(\beta_i)$.
3. Output $\mathcal{Q}$.

Figure 7: The AL-2 algorithm.

As in our presentation of the AL-1 algorithm, the algorithm in Figure 7 uses more variables than necessary in order to simplify the proof of the next theorem. The following lemma, whose proof is routine, is also used:

Lemma 53 Let ζ_1 and ζ_2 be configurations with a common location ω such that

1. $\zeta_1 \sqsupseteq \zeta_2$;
2. both ζ_1 and ζ_2 can be rewritten at location ω using the rewrite rule $\langle \alpha, \beta \rangle$.

Then there are substitutions θ_1 and θ_2 such that $\theta_1(\alpha) = \zeta_1[\omega]$, $\theta_2(\alpha) = \zeta_2[\omega]$, and $\theta_1(\zeta_1[\omega \leftarrow \beta]) \sqsupseteq \theta_2(\zeta_2[\omega \leftarrow \beta])$, i.e., the ordering relation still holds after rewriting. ∎

Theorem 54 Refer to the notation of Figure 7, in which τ_1 is the initial ground configuration of the computation. Let $\mathfrak{R}' = \mathfrak{R} \cup \mathcal{Q}$, the set of rewrite rules obtained by combining the original rules $\mathfrak{R}$ with the set $\mathcal{Q}$ returned by the AL-2 algorithm. Let ζ_1 be any configuration such that $\zeta_1|_\sigma \sqsupseteq \tau_1|_\sigma$ and $\zeta_1 \Rightarrow \ldots \Rightarrow \zeta_{n+1}$ over the same path π as the input computation. Then $\zeta_1|_\sigma \Rightarrow^* \zeta_{n+1}|_\sigma$ over a path of length n, using the rules in $\mathfrak{R}'$. In particular, $\tau_1|_\sigma \Rightarrow^* \tau_{n+1}|_\sigma$, so that the original example can be recomputed with the abbreviation substituted into the original configuration.

PROOF: Since the path π applies to ζ_1, $\alpha_1 \sqsupseteq \zeta_1[\omega_1]$. If it happens that $\alpha_1 \sqsupseteq (\zeta_1|_\sigma)[\omega_1]$, then $\alpha_1' = \alpha_1$. Otherwise α_1' is the result of reunification in step 2.1. By Lemma 51,

$$\alpha_1 \sqsupseteq \alpha_1' \sqsupseteq \zeta_1[\omega_1] \sqsupseteq \tau_1[\omega_1]$$

and

$$\alpha_1'|_\sigma \sqsupseteq (\zeta_1|_\sigma)[\omega_1] \sqsupseteq (\tau_1|_\sigma)[\omega_1].$$

Let $\langle \alpha_1'|_\sigma, \beta_1'|_\sigma \rangle$ be the rule added to $\mathcal{Q}$ in step 2.22 if $\alpha_1' \neq \alpha_1$, or $\langle \alpha_1, \beta_1 \rangle$ otherwise. By Lemma 53, if we use this rule to rewrite $\zeta_1|_\sigma$ to $\zeta_2|_\sigma$ and to rewrite $\tau_1|_\sigma$ to $\tau_2|_\sigma$ at location ω_1, we have that $\zeta_2|_\sigma \sqsupseteq \tau_2|_\sigma$. By assumption, $\zeta_2 \Rightarrow^* \zeta_{n+1}$ over the last $n-1$ steps of the path π. Thus we can repeat this argument $n-1$ times more, obtaining a path

$$\ldots, [\langle \alpha_i', \beta_i' \rangle, \omega_i], \ldots$$

over which $\zeta_1|_\sigma \Rightarrow^* \zeta_{n+1}|_\sigma$. ∎

Example 55 [LP] Consider the 3-step computation

$$plus(s(s(0)), s(0), s(s(s(0)))) \Rightarrow plus(s(0), s(0), s(s(0))) \Rightarrow plus(0, s(0), s(0)) \Rightarrow true.$$

With the synonym $2 \sim s(s(0))$, the initial goal is $plus(2, s(0), s(2))$. For the first step of the computation, AL-1 adds the rule

$$plus(2, x_3, s(2)) :- plus(s(0), x_3, 2) \tag{2}$$

to $\mathcal{Q}$. For the second step, the rule

$$plus(s(x_4), x_5, s(2)) :- plus(x_4, x_5, s(0)) \tag{3}$$

is added. No new rule is required for the final step. Thus $\mathcal{Q}$ consists of the two rules, (2) and (3). ∎

In the preceding example, suppose that we subsequently apply the AL-1 algorithm to the computation using the new rules starting from the goal $plus(2, s(0), s(2))$. The result is the rule:

$$plus(2, s(0), s(2)) :- true.$$

This can be used to supplant rule (2), since for this particular program, $y := s(0)$ is the only valid instance of x_3. Similarly, applying AL-1 to the second goal $plus(s(0), s(0), 2)$ in the path yields the rule

$$plus(s(0), s(0), 2) :- true.$$

That the the condition $\zeta_1|_\sigma \sqsupseteq \tau_1|_\sigma$ in Theorem 54 is necessary is illustrated by the next example.

Example 56 [LP] With the three-step computation starting from $plus(s(s(0)), 0, s(s(0))) \Rightarrow^* true$ and the synonym $2 \sim s(s(0))$, AL-2 constructs the one rule,

$$plus(2, x_1, 2) :- plus(s(0), x_1, s(0)). \tag{4}$$

With this rule, the goal $plus(2, 0, x_2)$ is computable over the same path, but $plus(s(x_3), 0, 2)$ is not. (Note that $plus(s(x_3), 0, 2) \not\supseteq plus(2, 0, 2)$.) ∎

In the two preceding examples, the rules found by AL-2 each have only a single valid instance, and hence are not very interesting. In the next two examples, however, the algorithm produces stronger generalizations.

Example 57 [LC] If we apply the AL-2 algorithm to the computation

$$plus\ [s, [s, 0]]\ 0 \Rightarrow^* [s, [s, 0]],$$

using the lambda-calculus program of Example 30 and the synonym $2 \sim [s, [s, 0]]$, the AL-2 algorithm fails to find any need to reunify until reaching the following configuration:

$$(((\underline{2\ (\lambda v_2.\lambda v_3.v_2)})\ 0)\ J),$$

where $J = (succ((plus(second[s, [s, 0]]))\ 0))$. The β-reduction rule that is used in the (unabbreviated) ground computation is:

$$(\lambda v_4.((v_4\ x_1)\ x_2)x_3) \Rightarrow ((x_3\ x_1)\ x_2).$$

When we reunify the underlined subterm and the synonym for 2 with the left-hand side of this rule, and then apply the resulting substitution to the right-hand side of the rule, we obtain the following new rule:

$$(2\ x_3) \Rightarrow ((x_3\ s)\ [s, 0]). \tag{5}$$

Continuing the computation, we again encounter a reduction of the form $2\ (\lambda v_2.\lambda v_3.v_2)$, but the same new rule (5) is derived. In the end, the set Q consists of just the rule (5).

It is interesting to note that this rule is neither a β-reduction rule nor a name-replacement rule, but instead a combinator applying the constant "2" to an arbitrary argument—in fact, a complete and correct definition of the meaning of this symbol in the enlarged language. ∎

Example 58 [AP] When we apply the *AL-2* algorithm to the computation $((plus\ 2)\ 0) \Rightarrow^* 2$, using the program of Example 29 and the synonym $2 \sim (succ\ 0)$, the result is the new rule,

$$((plus\ 2)\ x_1) = (s\ ((plus\ (s\ 0))\ x_1)).$$

When AL-1 is also applied to the example computation, the result is the more concise rule, $((plus\ 2)\ x_1) = (s\ (s\ x_1))$. ∎

Comparing the three preceding examples, we note that, whereas the three computations are semantically the same ($2+0=2$), the rules found by AL-2 on behalf of the abbreviation 2 are entirely different, in both form and generality. This stands in marked contrast to the AL-1 algorithm, where the semantics of the learned rules were, in some sense, independent of the syntax of the TRS. It is not hard to see why: an abbreviation is, after all, a syntactic modification, and the operational role of a symbol, such as "2", is expected to vary with the TRS. Thus 2 is a combinator (like *plus*) in LC, a function in AP, amd a constant term in LP.

Finally, let us note that a more general algorithm than AL-2 can be devised that introduces more complex abbreviations than just constants. For example, if we wanted to introduce the formal abbreviation "$[x_1, x_2]$" for "$\lambda v_1.((v_1 x_1)x_2)$", we could not do it using the AL-2 procedure, because this abbreviation is not a constant. Extending the algorithm in this way entails modifying the language, including the ttg, in ways that are difficult to generalize over the full range of our NTTRS's, but the fundamental concepts and procedures remain essentially the same.

Deterministic Term-Rewriting Systems

The NTTRS model has three main 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 (term rewriting) common to the programming languages used for Artificial Intelligence.
- Nondeterminism.

As noted above, the use of a nondeterministic model is appropriate for algorithms that learn from success, because 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.

A deterministic rewriting system requires a recursive "choice" algorithm for selecting the next position to rewrite and the rule to apply. Whereas the "state" of a nondeterministic computation is just the current configuration, the state of a deterministic computation may depend upon the entire sequence of configurations since the beginning of the computation. The results of learning in a deterministic system may lead to changes in both the rewrite rules and the choice algorithms.

The AL-1 and AL-2 algorithms propose changes only to the rules. The AL-1 algorithm proposes a single rule that compresses an n-step computation into a single rewriting step, and the AL-2 algorithm offers a set of rules that enable each step of the computation to be carried out in an enhanced language that has abbreviations for some of the ground terms. A real programming system can apply the AL-x algorithm(s), but will also need additional procedures for incorporating the learned rules into the deterministic process.

Questions of how best to incorporate the new rules into the program or to modify the choice functions are outside the scope of these algorithms. Often called *utility problems*, such questions require that we make explicit assumptions, such as what possible choice functions are available to the system, how problem instances are presented, what performance characteristics we want to optimize by learning, etc. Studies such as [5, 9, 20, 21] have looked at ways to measure the potential effects on performance of incorporating certain rules into a program, and ways to decide if and how to make such changes. Since no universal results have yet emerged, the problem is evidently quite difficult. But by separating the process of proposing new rules from questions of utility, our model may make it easier to formalize and reason about such matters. To be sure, the failure to separate these issues has complicated a number of previous presentations of Analytical Learning research results.

Conclusions

A Nondeterministic, Typed-Term Rewriting System is programming language schema that captures enough of the features of AI programming languages to enable us to cast the EBG learning algorithm in a very general form. In essence, this algorithm compresses a multi-step computation into one step and produces a single rewrite rule to carry out this step. An important consequence of this generalization is that EBG-like analytical learning can be applied to languages other than first-order predicate calculus. To show that the usefulness of these TRS's extends beyond a single algorithm, another analytical learning algorithm (AL-2) has been derived and analyzed within the same TRS framework. This algorithm is similar to AL-1 in that it learns from a successful computation, preserves the semantics of the original program, and proposes new rules that streamline computations along the same path. It differs in deriving multiple rules from the computation and in helping to install new symbols, chosen by an outside element, that abbreviate certain frequently occurring terms.

We have not discussed the computational complexity of our algorithms, but they are easily seen to require time and space polynomial in the length of the input computation. Algorithms for parsing a sentence in a context-free grammar and for computing a most-general unifier account for most of the running time.

Since the focus of this work is theoretical and no extensive empirical tests of these algorithms have been carried out, the usefulness of these algorithms for Machine Learning remains to be investigated. Nevertheless, we would like to make a conjecture. In our scheme, a computation is any finite sequence of rewrites. In particular, there is no requirement that the final state in the sequence be a Church-Rosser normal form. Thus given a computation of length 5, we could apply AL-1 to the entire computation, or to only the first four steps, or to 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 [16, 23, 25].

We have already remarked that when a path is extended, more restrictions apply, and the resulting rule from AL-1 is therefore less general. For this reason it seems reasonable

to recommend the following strategy: in any given computation, apply AL-1 to all subcomputations 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.

Acknowledgments

Discussions with Peter Friedland, Smadar Kedar, Rich Keller, Steve Minton, Monte Zweben, and Masa Numao have benefitted the authors in this work.

References

[1] S. Arikawa. Elementary formal systems and formal languages—simple formal systems. *Memoirs of Fac. Sci., Kyūshū University, Ser. A (Math)*, 24:47–75, 1970.

[2] S. Arikawa, T. Shinohara, and A. Yamamoto. Elementary formal system as a unifying framework for language learning. In *Proc. COLT '89*, Morgan Kauffman, 1989.

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

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

[5] W. W. Cohen. Generalizing number and learning from multiple examples in explanation based learning. In *Proc. Fifth Int. Machine Learning Workshop*, Morgan Kaufmann, 1988.

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

[7] P. Cohn. *Universal Algebra.* Dordrecht: D. Reidel, 1981.

[8] S. Dietzen and F. Pfenning. *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, 1989.

[9] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3, 1972.

[10] J. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper and Row, 1986.

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

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

[13] P. Hudak. Conception, evolution, and application of functional programming languages. *Computing Surveys*, 21(3), 1989.

[14] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *J.ACM*, 27, 1980.

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

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

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

[18] P. Laird. *Learning from Good and Bad Data.* Kluwer Academic, 1988.

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

[20] S. Minton. Quantitative results concerning the utility of explanation- based learning. In *AAAI-88 Proc.*, Morgan Kaufmann, 1988.

[21] S. Minton. Selectively generalizing plans for problem solving. In *Proc. Ninth IJCAI*, Morgan Kaufmann, 1985.

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

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

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

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

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

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

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

[29] J.-F. Puget. Learning invariants from explanations. In *Proc. Sixth Int. Machine Learning Workshop*, Morgan Kaufmann, 1989.

[30] J. Reynolds. Transformational systems and the algebraic structure of atomic formula. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, Edinburgh University Press, 1970.

[31] R. Smullyan. *Theory of Formal Systems.* Princeton University Press, 1961.

[32] C. Walther. Many-sorted unification. *J.ACM*, 35(1):1–17, 1988.

APPENDIX: A Note on Sort Hierarchies and TTL's

In the discussion of the subsumption lattice of typed-term languages, we noted that ttl's enjoy a Robinsonian unification property: any two unifiable terms have a unique *mgu*, apart from syntactic variants. Walther [32] has shown that unification in a many-sorted language with a sort hierarchy is Robinsonian iff the ordering is a forest. The purpose of this appendix is to clarify the relationship between unification in many-sorted languages and unification in ttl's. We argue that a many-sorted language with a forest-ordered sort hierarchy can be generated by a ttg, but that the converse does not seem to be the case.

We first need to define what constitutes a sub-sort (or sub-type) in a ttl. Only general types are considered here, since there is nothing corresponding to special types in Walther's theory. The natural way is to define one type G^2 to be a sub-type of another type G^1 if the symbol G^2 can be generated starting from G^1, i.e., $G^1 \rightarrow^* G^2$. Then the variables x^1 and x^2 (of types G^1 and G^2, respectively) are unifiable in our sense by the (unique) *mgu*: $\theta = \{x^1 := x^2\}$. The sort hierarchy induced by a ttg need not be a forest, however, since the grammar may contain productions such as $G^1 \rightarrow G^3$ and $G^2 \rightarrow G^3$. With such a grammar, our unification algorithm would not admit unifying the terms x^1 and x^2, since $x^1 \notin \mathcal{L}(G^2)$ and $x^2 \notin \mathcal{L}(G^1)$. By contrast, in Walter's formalism, the term x^3 (of sort G^3) would be an *mgu*. If, however, there were a type (say, G^0) of which both G^1 and G^2 were sub-types, then x^1 and x^2 would become unifiable in the ttl framework, and (as in many-sorted languages) the *mgu* would not be unique: both x^1 and x^2 are *mgu*'s. Note, however, that introducing the type G^0 has rendered the grammar ambiguous; hence it is not a ttg. (Compare the earlier discussion about Elementary Formal Systems, where, again, ambiguity led to loss of the Robinsonian property for unifiers.)

Turning this around, we can make a many-sorted language with a forest-structured sort hierarchy into a ttg. Consider, for example, a sort A with sub-sorts A^1 and A^2 and let B be a sort for which there are no sub-sorts. Suppose there is a function f that takes as arguments a pair of terms of sort A^1 and B and returns a term of sort A^2. Assume, also, that there are constants a, a^1, a^2, and b of sort A, A^1, A^2, and B, respectively, and a countable set of variables associated with each sort. We can construct a ttg that generates the terms in this algebra by assigning a general type to each sort and using the following productions:

$$
\begin{aligned}
A^i &\rightarrow x^i_j, \quad j \geq 1, i \in \{1,2\} \\
A^i &\rightarrow a^i, \quad i \in \{1,2\} \\
A &\rightarrow A^i, \quad i \in \{1,2\} \\
A &\rightarrow x_j, \quad j \geq 1 \\
A &\rightarrow a \\
B &\rightarrow y_j, \quad j \geq 1 \\
B &\rightarrow b \\
A^2 &\rightarrow F \\
F &\rightarrow f\,(\,A^1, B\,)
\end{aligned}
$$

Here we have introduced F as a special type and used "f", comma, and parentheses as constants. For example, the term $f(a^1, y_2)$ is in $\mathcal{L}(A^2)$. This illustrates how to construct a ttg that generates the terms in a free many-sorted algebra with a forest hierarchy of types.

By contrast, there does not seem to be an obvious way to represent the typed terms of a ttl as a simple many-sorted language, even if we impose a partially-ordered sort hierarchy.