Explanation-Based Generalization in a Logic-Programming Environment*

Haym Hirsh
Computer Science Department
Stanford University
Stanford, CA 94305

Abstract

This paper describes a domain-independent implementation of explanation-based generalization (EBG) within a logic-programming environment. Explanation is interleaved with generalization, so that as the training instance is proven to be a positive example of the goal concept, the generalization is simultaneously created. All aspects of the EBG task are viewed in logic, which provides a clear semantics for EBG, and allows its integration into the logic-programming system. In this light operationality becomes a property requiring explicit reasoning. Additionally, viewing EBG in logic clarifies the relation of learning search-control to EBG, and suggests solutions for dealing with imperfect domain theories.

I Introduction

Mitchell, Keller, and Kedar-Cabelli (1986) present a unifying framework for an explanation-based approach to generalization. Its underlying idea is to form an explanation structure (such as a plan or a proof tree) for a specific situation, and generalize the explanation structure so that it applies to a wider range of situations. *Explanation-based generalization* (EBG) uses a logical representation for knowledge, and an inferential view of problem solving. DeJong and Mooney (1986) suggest a more general term, *explanation-based learning* (EBL), to also cover systems that may specialize knowledge using information from an explanation structure. They take a problem-space view of problem-solving, in which generalization is a method of acquiring schemata for problem solving.

This paper describes the result of viewing all aspects of EBG in the logical formalism of the MRS logic-programming system (Russell 1985). MRS provides many forms of inference, including forward-chaining, backward-chaining, resolution, and residues (a form of abduction that has many similarities to EBG), and allows specification of proof strategies in meta-level theories. The underlying representation of MRS is logic. A user provides the system with a set of rules, and selects a form of inference with which facts are to be proved. The discipline of logic provides a clear semantics for EBG, and allows integration of EBG with other logic-inference methods. Under such strict formal representation it becomes possible to reason about operationality, as well as provide a consistent framework for learning search control in EBG. It furthermore clarifies the difficulty of imperfect domain theories in EBG, and suggests some solutions to this problem.

II Framework

The logical framework of Mitchell, Keller, and Kedar-Cabelli was taken as the appropriate starting point for this work. EBG takes knowledge (rules and facts) about a goal concept, an instance of the concept, and operationality of predicates. Given a specific instance of a concept (the *training instance*), and knowledge about that concept (the *domain theory*), the task for EBG is to find a definition of the concept expressible in terms of operational predicates. It does this by using the domain theory to prove that the instance is an example of the goal concept, generalizing the proof to find an operational description of a larger class of instances that are verifiable examples of the goal.

III Example

This framework for EBG is illustrated using the `Safe-To-Stack` example from Mitchell, Keller, and Kedar-Cabelli (1986). Given the following facts about two objects, `Obj1` and `Obj2`, that satisfy `Safe-To-Stack(Obj1,Obj2)`,

*Discussions with Paul Rosenbloom, Devika Subramanian, Ray Mooney, Smadar Kedar-Cabelli, Stuart Russell, Rich Keller, Allen Van Gelder, and Michael Genesereth had a significant impact on this work. Comments by Paul Rosenbloom, Marianne Winslett, Stuart Russell, and Jane Hsu on earlier drafts of this paper were invaluable. MRS has been developed by the Logic Group at Stanford University; the EBG program was written on top of the existing MRS architecture, and incorporates modified versions of Jeff Finger's RESIDUE method (Finger and Genesereth 1985). Computing resources were funded in part under NIH grant RR-00785 from the Division of Research Resources Biomedical Research Technology Program. The arrangement of facilities at ISI by Norm Sondheimer and Bob Neches for work on this paper is greatly appreciated.

```
On(Obj1,Obj2)
Isa(Obj1,Box)
Isa(Obj2,Endtable)
Color(Obj1,Red)
Color(Obj2,Blue)
Volume(Obj1,1)
Density(Obj1,0.1),
```

and rules about the safety of stacking one object on another (with appropriate procedural attachment for "×" and "<"),*

```
Not(Fragile(y))→Safe-To-Stack(x,y)
Lighter(x,y)→Safe-To-Stack(x,y)
Volume(p1,v1)∧Density(p1,d1)∧×(v1,d1,w1)
    →Weight(p1,w1)
Isa(p1,Endtable)→Weight(p1,5)  [Default Rule]
Weight(p1,w1)∧Weight(p2,w2)∧<(w1,w2)
    →Lighter(p1,p2),
```

and facts about the operationality of predicates

```
Operational(Volume(p,v))
Operational(Density(p,d))
Operational(On(x,y))
Operational(Color(p,c))
Operational(Isa(x,o))
Operational(×(x,y,z))
Operational(<(x,y)),
```

the EBG system should verify that the training instance is indeed a correct example of **Safe-To-Stack**, and generalize its verification to form a rule specifying a larger set of cases that are **Safe-To-Stack**:

```
Volume(v2,v37)∧Density(v2,v38)
        ∧×(v37,v38,v33)
        ∧Isa(v3,Endtable)∧<(v33,5)
    →Safe-To-Stack(v2,v3).
```

EBG constructs a proof that the example satisfies the goal, as in Figure 1. This proof is generalized, resulting in the generalized explanation structure for **Safe-To-Stack(v2,v3)** shown in Figure 2. The conjunction of the operational predicates (**Volume**, **Density**, ×, **Isa**, and <) in the generalized proof form a condition on the generalized goal predicate. Mooney and Bennett (1986) provides more detail on three other versions of EBG. Further detail on the implementation of EBG for this work follows.

*Throughout this paper variables begin with lower-case letters and are assumed to be universally quantified. All relations, including <, are written in prefix form, and n-ary functions are written as $n+1$-ary relations, with the result as the additional $n+1$st argument.

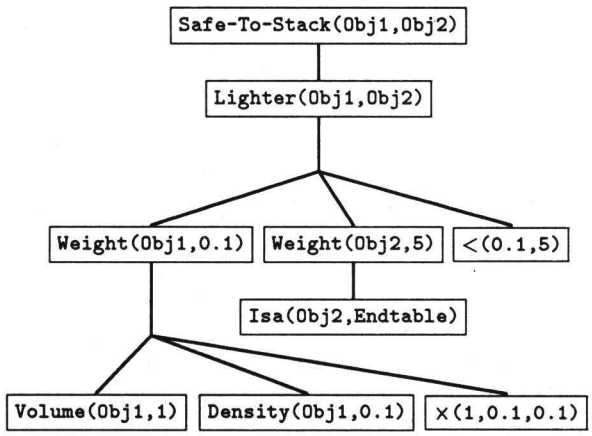

Figure 1: Explanation structure (proof tree) for **Safe-To-Stack(Obj1,Obj2)**

IV Interleaved Explanation and Generalization

As just described, the task of EBG is to prove that an instance is an example of a concept, and generalize the proof to form an operational description of the concept. These two stages, *explanation* and *generalization*, are typically done sequentially: after creating a proof, a generalization step forms the operational description from the proof. This work takes a different approach—the two stages are done simultaneously. The system attempts to prove that the instance is an example of the concept (such as **Safe-To-Stack(Obj1,Obj2)** above) by backward chaining on the goal through rules until training-instance facts are reached. However, each time a rule is used, it is simultaneously applied backward to the variablized goal concept (such as **Safe-To-Stack(v2,v3)**), creating the generalized explanation structure in parallel with the instantiated explanation structure.

EBG is started on a specific goal, such as **Safe-To-Stack(Obj1,Obj2)**. The first step is to find all rules that could potentially conclude this fact, namely all those whose consequent unifies with the goal concept. They are

```
Not(Fragile(y))→Safe-To-Stack(x,y)
```

and

```
Lighter(x,y)→Safe-To-Stack(x,y).
```

The first is tried (since it occurs earlier in the data-

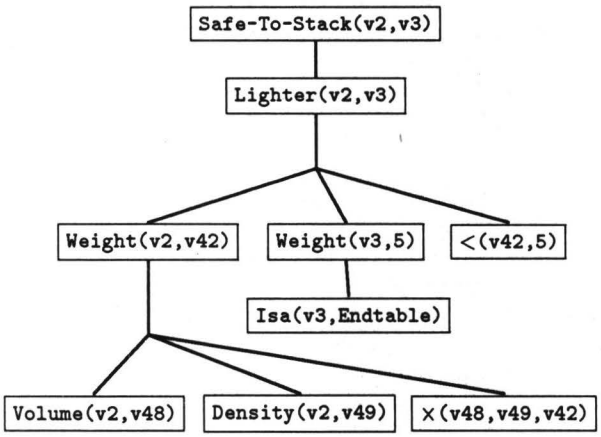

Figure 2: Generalized explanation structure (proof tree) for **Safe-To-Stack(Obj1,Obj2)**

base), and generates the subtask **Not(Fragile(Obj2))** by unifying the consequent of the rule with **Safe-To-Stack(Obj1,Obj2)** and applying the binding list thus formed to the antecedent of the rule. The antecedent becomes the new subtask. The generalized subtask **Not(Fragile(v2))** is generated in the same way, by unifying the consequent of the rule with the generalized form of the original task, **Safe-To-Stack(v2,v3)***, and applying the resulting bindings to the antecedent of the rule, resulting in a new generalized subtask. Thus the generalized explanation structure is formed in parallel to the formation of the instantiated explanation structure.

Not(Fragile(Obj2)) fails (for lack of applicable rules), so the system backtracks to the earlier goal **Safe-To-Stack(Obj1,Obj2)**, and its generalization, **Safe-To-Stack(v2,v3)**. The second rule is now tried, resulting in **Lighter(Obj1,Obj2)** and **Lighter(v2,v3)**. The backward-chaining process continues, at each step unifying both the current task and its generalization with the consequents of rules and applying the resulting bindings to the antecedents to obtain new subtasks. Failure to prove a subtask causes backtracking to an earlier subtask for a different rule selection or variable binding. Backtracking also occurs if a subtask is proved, but yet the subtask's generalization is not operational, since EBG must generate an operational definition of the concept. Note that backtracking removes all record of failed attempts to prove the task, and thus erroneous proof paths will affect neither the explanation structures, nor EBG's final learned rule.

*The generalized form of the first goal (**Safe-To-Stack(v2,v3)** here) is created by simply taking the non-unified consequent of the rule used to backward chain from the instantiated initial goal.

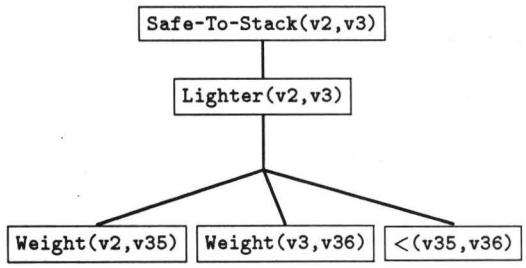

Figure 3: Generalized explanation structure (proof tree) for **Safe-To-Stack(Obj1,Obj2)** when **Weight(p,v)** is operational

The above description of backward-chaining on generalized goals while creating the proof has one exception. If a generalized subtask is ever operational, but the instantiated subtask requires further proof, the system merely continues normal proof of the subtask alone. This causes EBG to find more general operational definitions than it might otherwise form. The current generalized subtask becomes a terminal node in the developing generalized explanation structure, without inclusion of any of the subtasks that occur below it. For example, if **Weight** is operational (i.e., **Operational(Weight(p1,w1))** is in the database for **Safe-To-Stack**), the resulting generalized explanation structure will be that of Figure 3.

The final step of EBG is to form the conjunction of terminal nodes of the generalized explanation structure, and create a rule with this conjunction as the antecedent and the top goal of the generalized explanation structure as its consequent. All bindings formed during creation of the generalized explanation structure are applied to this rule, to handle interacting subgoals correctly. The resulting rule is the final result of EBG. Thus

Weight(v2,v35)∧Weight(v3,v36)∧<(v35,v36)
→Safe-To-Stack(v2,v3)

would be the rule learned for the preceding example in which **Weight** is operational.

DeJong and Mooney (1986) point out that there is often more than one way to prove that an example is indeed an instance of a concept, and each such differing proof can result in a different rule from EBG. Thus, for example, if the training-instance data for **Obj2** included knowledge for concluding **Not(Fragile(Obj2))**, there would be two ways to prove **Safe-To-Stack(Obj1,Obj2)**, and thus two rules could be created by EBG, depending on which proof was chosen. As discussed in Section VII, such selection is deterministically specified by the order of rules in the

domain theory—whichever knowledge came first, fragility or weight, would succeed in creating a proof. However, logic programming systems often provide a means for finding multiple results due to alternative inferential paths. In MRS, if the plural form of an inference method is selected, MRS finds *all* conclusions that can be found with the specified inference method. Since EBG has been implemented as an additional form of inference for MRS, EBGS was also implemented, to allow creation of all rules that can be formed from all possible proofs of the training instance. Furthermore, meta-level knowledge can be used to specify which rule to select at any point, thus determining which explanation EBG will form, and hence what the final learned rule will be.

V Operationality

Operationality criteria are one of the inputs that Mitchell, Keller, and Kedar-Cabelli specify for their EBG, but the only operational definition they give for it is "The concept definition must be expressed in terms of the predicates used to describe examples ... or other selected, easily evaluated, predicates from the domain theory." It is only a small extension to allow specification of any set of predicates, such as including `Weight` in the example above. However, DeJong and Mooney (1986) and Mooney and Bennett (1986) point out that such simple selection of operational predicates is insufficient, and provide examples in which operationality is a function of the proof structure. In one such example they define operationality to be predicates at terminal nodes of the generalized explanation structure after all predicates that only support `Isa` facts are removed. This dynamic operationality cannot be supported in EBG as described by Mitchell, Keller, and Kedar-Cabelli.

Another example of dynamic operationality not included in Mitchell, Keller, and Kedar-Cabelli (1986), nor in DeJong and Mooney (1986) or Mooney and Bennett (1986), is applying a theorem-prover to decide operationality dynamically. Since explanation is interleaved with generalization, it becomes easy to prove operationality while the actual goal is being proved. For example, an axiomatization of Keller's (1987) definition of operationality could be used to determine operationality dynamically, with predicate definitions changing over time as problem solving occurs. The axiomatization could even perform experimentation or look-ahead search to determine whether a specific predicate is operational. Finally, since operationality is now a problem-solving task, EBG can be applied to it as well, resulting in better operationality criteria, suited to the particular task at hand.

Deliberate reasoning about operationality allows the user to specify rules about operationality rather than rigidly listing a set of operational predicates. Such application of logic programming allows operationality to be defined in terms of proof progress, rather than of the final explanation structure. Thus operationality can change while doing explanation. For example, if the logic-programming system provides caching of intermediate results, a rule such as "If a predicate succeeds n times, make it operational" (for some n) could be used to specify operationality.

The motivation for making operationality a provable property is the notion of meta-level reasoning in MRS. In MRS, proof strategies can be specified in a meta-level theory. Thus, for example, a meta-level rule can specify to try one branch of a proof over another if it uses an arithmetic predicate. The vocabulary already exists within MRS to specify such properties, and hence it becomes easy to specify operationality rules such as

$$\texttt{Arithmetic-Predicate(pred(arg}_1\texttt{,...,arg}_n\texttt{))} \rightarrow \texttt{Operational(pred(arg}_1\texttt{,...,arg}_n\texttt{))}.^{*}$$

As another example, the rule specifying "If only one potential path is possible at this point, make the current subtask operational" would use the same meta-level predicates used by meta-level proof strategies in MRS. This notion of operationality is more robust than operationality as specified by Mitchell, Keller, and Kedar-Cabelli.

This also highlights the difference between the EBG described here and that of DeJong and Mooney (1986) and Mooney and Bennett (1986). In their problem-space framework, they simultaneously generate both the instance's explanation structure and the generalized explanation structure. However, during creation of the generalized explanation structure they do not reason about operationality; only later, during the subsequent generalization stage, do they determine operationality and modify the explanation structure.** The generalization stage prunes the generalized explanation structure using operationality, and then uses the resulting abridged structure to form the rule. Thus, although much of the work of generalization is accomplished during explanation, they still separate explanation and generalization.

One difficulty with ignoring operationality during explanation generation is that situations can arise in which an explanation is generated from which no operational concept definition can be created, yet other explanations exist from which operational definitions can be formed. Since in this work operationality is determined during the explanation process, such situations do not occur. As soon as a branch terminates without an operational definition, backtracking will occur to try to find an alternate proof for the branch from which an operational concept definition can

*This uses the ability to quantify over predicates in MRS.

**The generalized explanation structure is actually represented as an overgeneral explanation structure, with a binding list to sufficiently specialize the structure. Since operationality may change the binding list as well, it must also be modified during generalization.

be generated. It is impossible to generate a complete explanation structure without simultaneously ending with a generalized explanation structure that has an operational definition of the goal concept.

VI Search Control

Mitchell, Keller, and Kedar-Cabelli include an example of learning search control for integration operators. Their domain theory includes the following rules:*

```
Useful(op,x)↔Not(Solved(x))∧Solvable(op(x))
Solvable(x)↔
    ∃op[Solved(op(x))∨Solvable(op(x))]
Matches(x,"∫<any-fn>dx")→Not(Solved(x))
Matches(x,"<any-fn> ")→Solved(x)
Matches(op(x),y)↔Matches(x,Regress(y,op))
```

where **Regress** in the last rule propagates y through op. The goal concept is **Useful(Op3,x)**—to learn search control for **Op3**. However, the actual definition for **Op3** is never given. The hidden definition of **Regress** is the source of all resulting knowledge of **Op3**.

A footnote of Mitchell, Keller, and Kedar-Cabelli (1986) remarks: "Notice that the regression here involves propagating constraints on problem states through problem solving operators. This is a different regression step from the second step [generalization] of the EBG process, in which the goal concept is regressed through the domain theory rules used in the explanation structure." They are saying that for this task the **Regress** predicate is necessary in addition to EBG's own regression, since it is a different form of regression. The correct distinction, however, is not that they are different regressions, but rather that they are regressing through different knowledge representations. Rosenbloom and Laird (1986) point out that "the EBG implementation of search-control acquisition requires the addition of general interpretive rules to enable search with the task operators and the regression of the solution property through them, while Soar makes use of the same goal/problem-space/chunking approach as is used for the rest of the processing." Basically, they are saying that Soar has only one way to represent knowledge, and thus only has one form of regression possible. Likewise here, the strict enforcement of a logical representation forces representation of all knowledge, including operators, as rules in logic, with only a single form of regression.

Thus the distinction is not that Soar uses problem spaces, but rather that Soar consistently uses a single representation. Mapping this back to EBG, using a single representation scheme to represent all information uniformly allows use of a single form of regression. Here this means representing all information about operators, such as **Op3**, explicitly as rules in logic. A domain theory for integration satisfying this uniform representation constraint* is

```
Apply(op,problem,newproblem)
        ∧Solvable(newproblem)
    →Useful(op,problem)
Integrate(problem,answer)→Solvable(problem)
Any-Fn(problem)→Integrate(problem,problem)
∃op[Apply(op,problem,newproblem)
        ∧Integrate(newproblem,answer)]
    →Integrate(problem,answer)
Any-Fn(f1)∧Any-Fn(f2)∧Integrate(∫f2,answer)
    →Integrate(×(f1,∫f2),×(f1,answer))
Real(k)∧Unprovable(=(k,−1))∧+(k,1,m)
    →Apply(Op9,∫x^k,x^m/m)
Any-Fn(f)∧Real(r)
    →Apply(Op3,∫×(r,f),×(r,∫f)).
```

This database does two things. In addition to making the definitions of operators (such as **Op3** and **Op9**) explicit, it also makes the notion of state explicit. Rather than having functional application of operators within predicates (such as **Solvable**), **Apply** is used as a separate conjunct to create explicitly the state to which predicates will be applied.

Given that **Op3** is useful for $\int 7x^2 dx$ (i.e., **Useful(Op3,∫×(7,x²))**), as well as proper definitions for **Any-Fn**, operationality, etc., the rule learned by EBG, simplified by removing duplicate conjuncts, is

```
Any-Fn(x^v49)∧Real(v33)∧Any-Fn(v33)
        ∧Real(v49)∧Unprovable(=(v49,−1))
        ∧+(v49,1,v48)∧Any-Fn(x^v48/v48)
    →Useful(Op3,∫×(v33,x^v49)).
```

This is equivalent** to the simpler rule

```
Real(v33)∧Real(v49)∧Unprovable(=(v49,−1))
    →Useful(Op3,∫×(v33,x^v49)),
```

which is similar to the rule of Mitchell, Keller, and Kedar-Cabelli (1986).*** Note that the matching predicates of Mitchell, Keller, and Kedar-Cabelli (1986), **Matches(x,y)**, are done implicitly with unification, and that regression through operators is done by the same mechanism as regression through rules (since the operators *are* rules).

*Their domain theory has been modified a bit here for greater clarity.

*The domain theory has been simplified somewhat for presentation.

**All other conjuncts of the original antecedent will be true if the antecedent of this simpler rule is true. Prieditis and Mostow (1987) discuss how to use partial evaluation to generate such simplifications automatically.

***The difference, as in Rosenbloom and Laird (1986), is in the representation of integration problems and the applicability of operators to them. The difference is not significant.

VII Imperfect Domain Theories

Mitchell, Keller, and Kedar-Cabelli point out the reliance EBG has on the user-provided domain theory and identify the issue of imperfect domain theories as a topic for future work. They recognize three sources of imperfection in a domain theory: incompleteness, intractability, and inconsistency. Rosenbloom and Laird (1986) discuss two further sources, incorrectness and defeasibility of domain theories. The precision of a logical view of EBG greatly clarifies these issues, and suggests solutions to some of these issues.

As recognized by Mitchell, Keller, and Kedar-Cabelli, and Rosenbloom and Laird, the existence of a default rule to compute the weight of an endtable in the **Safe-To-Stack** example causes difficulties for application of EBG. Use of this default rule makes EBG learn an overgeneral rule, since the learned rule has hidden the fact that it was only relevant when the second object's weight was otherwise unprovable. The learned rule will thus allow concluding **Safe-To-Stack** for endtables that are not **Safe-To-Stack** using the original domain theory. This form of inconsistency raised by Mitchell, Keller, and Kedar-Cabelli is the same as the defeasibility issue raised by Rosenbloom and Laird.* In logic they become the issue of nonmonotonicity.

A theory is monotonic if adding new facts will never invalidate a previously proved fact. If new facts *can* invalidate previous results, the theory is nonmonotonic. Until now domain theories have been viewed as theories in a formal sense. However, when given to a theorem prover, the needs of efficiency outweigh the needs of theory, and extra-logical notions such as rule order and default rules must be added. These added computation devices cause monotonic theories to become nonmonotonic.

The **Safe-To-Stack** example has such nonmonotonicity. The order of rules in a database specifies what order the rules will be used in attempting a proof. By ordering the rules in the correct way, a rule can be made to serve as a default, to be used if other means of proof (tried earlier since they occurred earlier in the database) fail. Thus, for example, the endtable weight rule was placed after the volume times density weight rule to cause the endtable rule to serve as a default rule.

The cause of these difficulties is the fact that some rules conflict—that more than one rule can be used at the same time for some instances, and that different conclusions can be reached using these different rules. In such cases *conflict resolution* is necessary to decide which rule to select. The example of rule order above is one simple example of a conflict resolution strategy. The difficulty for EBG is that this extra information is not an explicit part of the domain theory, and is thus hidden from the EBG process, which therefore cannot incorporate this information into the learned rule. EBG is only as correct as the information given to it. Since knowledge of the default status of the endtable weight rule is kept hidden from EBG, it has no way to learn a correct rule.*

This view of the problem suggests a solution. If the difficulty is that information is hidden, the solution is to make this information explicit. There are two ways to do this. The first is to incorporate conflict resolution decisions into the explanation structure, so that EBG can form rules from all information used in verifying the instance. The domain theory would in effect contains all rules and facts for conflict resolution, and conflict resolution decisions will be incorporated into the explanation structure. Thus the explanation structure would include failing branches, which may be used in one of two ways. EBG could learn a rule with an added conjunct saying that **Weight(v3,v39)** is not provable using the volume-times-density **Weight** rule. It might alternatively learn a rule with an added conjunct corresponding to the specific reason for the failing branch in the explanation, **Unprovable(Volume(v3))**.

As discussed below, such addition of conflict resolution decisions into the explanation structure will often cause EBG to create over-specialized rules. Furthermore, conflict-resolution strategies such as rule order are often difficult to encode explicitly for use in the explanation structure. In these cases a second choice, reformulating the domain theory, becomes appropriate. The goal of such reformulation should be to remove all need for conflict resolution, namely, to eliminate overlap in conflicting rules. This can be done using predicates about provability. For example, the following reformulated domain theory for the **Safe-To-Stack** example makes the nonmonotonicity of the default rule explicit:

```
Not(Fragile(y))→Safe-To-Stack(x,y)
Lighter(x,y)→Safe-To-Stack(x,y)
Volume(p1,v1)∧Density(p1,d1)∧×(v1,d1,w1)
    →Weight1(p1,w1)
Isa(p1,Endtable)∧Unprovable(Weight1(p1,w))
    →Weight(p1,5)
Weight1(p1,w)→Weight(p1,w)
Weight(p1,w1)∧Weight(p2,w2)∧<(w1,w2)
    →Lighter(p1,p2).
```

(Note that the volume-times-density weight rule now concludes the new predicate **Weight1** rather than **Weight**.) Given a database with these rules, EBG will learn the correct rule:

*A theory is defeasible if the addition of knowledge can change results.

*Lewis (1987) has discussed a similar concept of the *safety* of production-system learning mechanisms. When priority schemes are used to order productions, the result of composing two productions may be inconsistent with respect to the original production system.

```
Volume(v2,v48)∧Density(v2,v49)
        ∧×(v48,v49,v42)∧Isa(v3,Endtable)
        ∧Unprovable(Weight1(v3,v51))∧<(v42,5)
    →Safe-To-Stack(v2,v3).
```

Failure to compute the endtable weight has become explicit in the `Unprovable(Weight1(v3,v51))` conjunct of the new rule.

Finally, note that conflict resolution schemes can be used to serve two different purposes. In the `Safe-To-Stack` example there are two cases of rule conflict. The first is the two different ways to conclude `Safe-To-Stack(x,y)`: `Unprovable(Fragile(y))` and `Lighter(x,y)`. In this case rule order has no impact on the correctness of the results of EBG; rule order will only influence the speed with which the system reaches conclusions. On the other hand, the rule order of the two `Weight` rules embodies domain knowledge: the second rule is a default rule, and it is only correct to try it if the first rule fails. It is only this second use of conflict resolution, when it embodies domain knowledge, that can cause EBG to overgeneralize. Thus only some conflicting rules need be reformulated to remove the conflict. Likewise, only some conflict resolution decisions need be included in the explanation structure. Although in both cases EBG learns correct rules when all conflicting rules are addressed, the learned rules will often be over-specialized.*

VIII Conclusion

This paper has described an implementation of explanation-based generalization (EBG) within a logic-programming environment, in which generalization is done in parallel with explanation generation. All aspects of EBG are viewed in logic, which clarifies issues of EBG. Operationality becomes a provable property requiring explicit reasoning. Additionally, logical representation of domain operators unifies regression of constraints through operators with regression of goals through domain rules. Finally, overgeneralization by EBG is better understood when viewed from a logic standpoint. Making those inconsistencies in a domain theory that are due to nonmonotonicity explicit, either by incorporating conflict resolution decisions into the explanation structure, or by making rules mutually exclusive, removes the cause of overgeneralization for EBG.

*Soar (Laird, Rosenbloom, and Newell 1987) has solved this by providing two forms of *preferences* (Soar's version of conflict resolution). Necessity preferences (Laird, Rosenbloom, and Newell 1986) are used to specify those preferences that embody knowledge of the goal test. The backtracing mechanism used in Soar ignores all preferences except necessity preferences, which are incorporated into the learned chunks.

References

1 DeJong, G.F., and R. Mooney, "Explanation-Based Learning: An Alternative View". *Machine Learning* 1:2 (1986) 145–176.

2 Finger, J. J., and M. R. Genesereth, "RESIDUE—A Deductive Approach to Design Synthesis", Technical Report KSL-85-1, Knowledge Systems Laboratory, Computer Science Department, Stanford University, January, 1985.

3 Keller, R. M., *The Role of Contextual Knowledge in Learning Concepts to Improve Performance.* Ph.D. Thesis, Department of Computer Science, Rutgers University, January, 1987.

4 Laird, J. E., A. Newell, and P. S. Rosenbloom, "Soar: An Architecture for General Intelligence", Technical Report KSL-86-70, Knowledge Systems Laboratory, Computer Science Department, Stanford University, December, 1986. To appear in *Artificial Intelligence.*

5 Laird, J. E., P. R. Rosenbloom, and A. Newell, "Overgeneralization During Knowledge Compilation in Soar". In *Proceedings of the Workshop on Knowledge Compilation*, Otter Crest, Oregon, 1986.

6 Lewis, C. "Composition of Productions". In Klahr, D., Langley, P., and Neches, R. (editors), *Production System Models of Learning and Development.* Cambridge, Mass.: MIT Press, 1987.

7 Mitchell, T. M., R. M. Keller, and S. T. Kedar-Cabelli, "Explanation-Based Generalization: A Unifying View". *Machine Learning* 1:1 (1986) 47:80.

8 Mooney, R., and S. Bennett, "A Domain Independent Explanation-Based Generalizer". In *Proceedings of the National Conference on Artificial Intelligence.* Philadelphia, PA, August, 1986.

9 Prieditis, A., and J. Mostow, "PROLEARN: Toward a Prolog Interpreter that Learns". In *Proceedings of the National Conference on Artificial Intelligence.* Seattle, Washington, July, 1987.

10 Rosenbloom, P.S., and J.E. Laird, "Mapping Explanation-Based Generalization onto Soar". In *Proceedings of the National Conference on Artificial Intelligence.* Philadelphia, PA, August, 1986.

11 Russell, S., "The Compleat Guide to MRS". Technical Report KSL 85-12, Computer Science Department, Stanford University, June, 1985.