1 Introduction

"Explanation-based" learning (EBL) is a semantically-driven, knowledge-intensive paradigm for machine learning which contrasts sharply with syntactic or "similarity-based" approaches. It provides a way of generalizing a machine-generated explanation of a situation into a rule which applies not only to the current situation but to similar ones as well. The original explanation is normally created by a problem-solver, which relates the given situation to a stated goal using domain knowledge that is provided to it. EBL takes the output of the problem-solver — in the form of an explanation tree — and processes it to discover an appropriate rule. In fact, it is more perspicuous to talk in terms of "problems" rather than "situations". In this sense, EBL provides a mechanism for learning something from the solution to a problem that will expedite the solution to similar problems in the future.

While the EBL operation is generally seen as being quite distinct from problem solving, we find it fruitful to integrate the two closely. To motivate this approach, imagine a problem-solver that is used quite often. Over the months, the same or similar problems are repeatedly given to it to solve. Since searching for solutions is computationally very demanding, it would seem attractive to remember the solutions to previously-solved problems. But a simplistic approach to adding such a "memo function" suffers from three drawbacks. First, space may be wasted in storing a host of solutions to specific problems that will never be repeated. Second, as the memory grows, sophisticated indexing structures will become necessary to access past history — for it would be unfortunate if the cost of discovering whether a problem has been solved in the past exceeded that of re-solving it! Third, the mechanism will not help with a particular problem unless *exactly* the same one has been encountered before. Insignificant changes which do not alter the solution will nevertheless have a great effect on the solution time. What is really needed is a way of learning something more general from solving a problem. This is just what EBL offers.

The present paper is a foundational review of EBL which stresses its intimate connection with problem-solving. By re-examining the operations involved in conventional implementations, it becomes clear how to integrate learning elegantly and invisibly into the inference engine that constitutes the heart of a problem-solver. The paper proceeds as follows. The next section reviews the idea of EBL and the principal structures that have been proposed to accomplish it; we also briefly summarize current research directions. Then a formal characterization is presented and used to explain how to embed the necessary generalization into the problem-solver itself. Resolution theorem provers and production systems are examined as examples of backward- and forward-chaining problem-solvers respectively, and the algorithms, with and without EBL, are illustrated in PROLOG. Finally, more complex and realistic examples of problem-solvers are considered. A detailed analysis of the planning problem shows that generalization of plans can be accomplished quite naturally within the paradigm of learning as problem-solving.

2 Explanation-based learning

It seems clear that a great deal of human learning is accomplished by examining particular situations and relating them to background knowledge in the form of known general principles. Such analysis often results in new rules that can be applied in other situations. This kind of learning, called "explanation-based learning" (EBL), contrasts with approaches such as "similarity-based learning" that analyze a large number of examples to discover their structural similarities and differences. Rather than acquiring new knowledge inductively, EBL operationalizes knowledge that is already latent in general principles, through a process of deduction.

Figure 1 shows a graphical representation of these two styles of learning. Similarity-based methods take several examples (and, perhaps, counterexamples) of a concept and apply structural analysis to determine a generalized concept description. EBL is more involved. Given an example, an "explanation" is first constructed by applying a problem solver to relate the example to some general domain theory. The result of this operation is a trace or "proof" of the example with respect to the theory. This is then generalized, by discarding irrelevant parts of it, into an explanation that applies to other examples too. Finally, operational rules are extracted from the general explanation.

2.1 The external view

Four inputs to an EBL system can be distinguished: the *goal concept*, *training example*, *domain theory*, and *operationality criterion* [Mitchell 86]. The output is a rule that applies to the training example and to others like it. Figure 2 illustrates these constructs in the simple domain of cup recognition. (In this Figure and elsewhere in the paper, upper-case letters denote uninstantiated variables.)

The goal concept defines what is to be learned in terms of high-level (non-operational) predicates. For example, the definition of a cup in Figure 2 is high-level in Winston's function-structure system [Winston 83] because it involves functional properties (liftable, stable, open-vessel ...) rather than structural features (light, partof, pointing ...). The training example is a particular instance of the target concept. The domain theory represents the facts and rules that constitute what the learner knows. The cup domain includes facts about concavities, bases, and lugs, as well as rules about liftability, stability and what makes an open vessel. These are used to explain why the training example is an instance of the target concept.

The fourth input, the operationality criterion, specifies which concepts can be used in rules created by EBL. If all predicates were operational, the definition of cup as given by the goal concept would be acceptable as it stands and there would be nothing to learn. The whole point of the exercise is to characterize that concept in terms of attributes that are easy to test. For example, in the function-structure system, learned concepts are to be

represented in terms of structural rather than functional predicates.

The operational rule learned from this example is also shown in Figure 2. This rule is better than the original definition because it involves one less level of processing, allowing cups to be recognized more quickly[1]. However, it is less general.

2.2 The internal view

An explanation-based learner works as follows [Mitchell 86]. It first constructs an explanation, in the form of a proof tree, which describes why the training example is an instance of the goal concept (see Figure 3a). Next it generalizes the explanation using the method of goal regression [Waldinger 77]. This involves traversing the tree from top to bottom, replacing constants by variables, but just those constants that were not embedded in the rules or facts used to create the proof. This produces the generalized proof tree of Figure 3b. The tree is then pruned by removing leaf nodes until no operational predicates appear at internal nodes. Finally, the operational definition of the target concept is simply the conjunction of the remaining leaf nodes.

What we have described is a three-pass explanation-based generalizer. However, it was pointed out in [DeJong 86] that one-pass goal regression can over-generalize the proof tree. To avoid this requires two passes. The first propagates constraints down through the proof tree, and the second propagates them back up. This modified technique is a four-pass learning method.

2.3 Research directions

Perhaps the best-known example of EBL is LEX2 [Mitchell 83], which learns heuristics for performing symbolic integration. Given an expression to integrate, it first searches for a solution using the standard repertoire of integration transformations. Then it examines each operator in the solution chain and generalizes the expression as much as possible without violating the preconditions for that operator. This enables it to learn a new operator which can be applied to this and other problems, sidestepping the normal trial-and-error search procedure.

A rather different application is GENESIS [Mooney 85], which acquires schemata for use in story understanding. For example, given a tale involving kidnapping, GENESIS learns a schema that summarizes its important elements. It can use the schema to recognize instances of kidnapping in future stories. Winston's "function-structure system" [Winston 83] operationalizes functional concept descriptions to obtain structural descriptions. It learns rules for recognizing objects such as cups and chairs based on a domain theory of how their structure (handle and concavity; legs and seat) fulfils their function (drinking; sitting upon). Other EBL projects address the problems of generalizing software [Hill 87], circuit designs [Ellman 85] and floor plans [Mostow 87]; while still others

learn about momentum [Shavlik 85], causal relationships [Pazzani 87], and equation solving [Silver 83].

Some recent research strives to create hybrid methods that work with similarity- and explanation-based techniques together. The aim is to blend the (synthetic) ability to generalize from empirical observation with the (analytic) ability to deduce generalizations from a known domain theory [Silver 88]. One important goal is to be able to learn domain principles from painstaking observations and then use them to make quick one-shot generalizations. Other work investigates the application of explanation-based learning techniques to incomplete [Hunter 88] or intractable [Ellman 88] domains. One way of doing this is to use explanations to learn from failures [Clancey 88]. For example, one kind of failure can be caused by reliance on simplifying assumptions in an intractable domain, which can lead to the creation of underconstrained concepts. These later show up as unexpected failures, which can be explained and the explanation used to further constrain the concept [Chien 88]. Another line of research is to strengthen the generalization of explanations by extending it from constants to operators, in an attempt to bridge the gap from first- to second-order quantification. For example, if an operator is used several times in a row, the explanation can be generalized to include arbitrary repetition of that operator [Shavlik 88].

The present paper aims at a rather different target. Current research tends to try and split off the "learning" component and examine it in isolation, separated from the rest of the system. But in reality it is grossly misleading to imagine learning as a sort of module that can be demarcated, placed under the microscope, and studied in its own right. Instead, we reappraise the paradigm of EBL, emphasizing the integrity and unity of the problem-solving system as a whole. The method of producing the explanation should not be glossed over; indeed, it deserves to be the focus of any study of learning. Learning can be considered more as a side effect than as an end in itself. This orientation produces novel and illuminating insights into the nature of EBL. The fact that some recent research finds it necessary to perform convoluted manipulations of the explanations (eg [Shavlik 88], [Mooney 88], [Cohen 88]) does not subvert the paradigm but merely highlights the inappropriateness of current knowledge representation schemes and problem-solving methods [Gupta 88]. Our perspective also promotes the use of EBL in practice, for it strips away the mystique by showing how to modify general problem solvers in order to make them learn.

3 Extending problem solvers

The previous section's characterization of EBL as a three- or four-pass process, while commonly accepted in the literature, is overly complicated and betrays the technique's essential simplicity. Instead of tacking a generalization filter on to the problem solver's output, it is easier and more perspicuous to modify the problem solver to learn whilst solving its problems. This section shows how.

First a more formal characterization of EBL is presented which clarifies the technique and pinpoints what is needed to augment problem solvers with the capability to generalize examples they are given. Next we look at how to represent the domain theory. Then we see how a basic problem solver can be converted into a learning problem solver in the four stages summarized in Figure 4. The procedure is developed for both backward- and forward-chaining problem solvers, and for concreteness skeleton implementations are given in PROLOG. The final system is called MEL, for "model for explanation-based learning", and is derived in both backward-chaining (MEL/B, Figure 10a) and forward-chaining (MEL/F, Figure 10b) forms. These two programs successfully and, we believe, elegantly, incorporate EBL into the heart of a backward-chaining resolution theorem prover and a forward-chaining production rule interpreter.

3.1 Functional Decomposition of EBL

As Figure 1 illustrates, traditional EBL involves three operations which we write as mathematical functions. $\sigma(\mathcal{G}_s, \mathcal{D})$ is the operation performed by the problem solver. It takes the goal $\mathcal{G}_s$ and domain theory $\mathcal{D}$, and generates a specific explanation E_s of the goal with respect to the domain theory. $\gamma(E_s, \mathcal{G}_g)$ is the mapping performed by the generalizer. It takes the explanation produced by the problem solver, together with the generalized goal $\mathcal{G}_g$, and returns a generalized explanation E_g. As will be explained below, the goal has two components: a general one $\mathcal{G}_g$, used by γ, which indicates the concept to be learned, and a specific one $\mathcal{G}_s$, used by σ, which in effect specifies the training example. $\alpha(E_g, \Omega)$ is the mapping performed by the operationality pruner, which creates an operational rule from the generalized explanation and operationality criterion Ω.

The entire EBL operation is fully characterized by the composition of these functions:

$$\mu(\mathcal{G}_s, \mathcal{G}_g, \mathcal{D}, \Omega) \stackrel{\text{def}}{=} \alpha(\gamma(\sigma(\mathcal{G}_s, \mathcal{D}), \mathcal{G}_g), \Omega).$$

The equivalent single function μ describes EBL more compactly and clearly than the three components. Moreover, as will be seen shortly, it is easy to implement μ directly by augmenting a problem solver. This has the advantage that the computation is conceptually more economical because no intermediate representation of the explanation is needed.

In order to preserve functionality, the essential contribution of each of σ, γ, and α will be abstracted, although the packaging — in particular, the explicit passing of arguments from one to the next — will be stripped away. The function σ contributes the idea of *remembering* items, γ that of using *general* items, and α that of *selecting* items to participate in the final rule. This breakdown is mirrored in sections 3.3 to 3.6, which first describe the basic problem solver, then have it remember items, next have it remember general items, and finally have it remember only those general items that are necessary. These steps result in a problem solver which computes μ as it goes along.

3.2 Domain theory for problem solvers

The EBL technique can only be applied to "general" problem solvers, not to those which incorporate procedural knowledge that is specially tailored to particular problem domains. At the core of a general problem solver[2] is a small, simple inference engine that uses explicitly-represented domain knowledge to guide its search for solutions. As the burgeoning technology of expert systems demonstrates, this is a powerful architecture which can accommodate a great variety of problems from wide-ranging domains.

With a general problem solver, it is necessary for the user to specify knowledge in the form of a declarative "theory" that supports the solution of problems in the target domain. This background knowledge or domain theory comprises two parts, operators $\mathcal{O}$ and facts $\mathcal{F}$, and can be written $\mathcal{D} = (\mathcal{O}, \mathcal{F})$. Operators work in the domain to draw inferences or change the state of the problem solver. They may be production rules, schemata, etc, and have two principal components, an *antecedent* which determines the conditions under which the operator can be applied, and a *consequent* which specifies the result of applying the operator. In the simple problem solvers considered in this section operators have only these two components; in this case we usually call them "rules" and write A $\Rightarrow$ B, where A is the antecedent and B the consequent. More complex operators are discussed in Section 4.

Facts are of two kinds: *primitive* and *derived*. The former include those explicitly defined within the domain theory (like **isa(lug, handle)** and **isa(base, bottom)** in Figure 2), which are denoted by F_d, as well as input facts (like **owner(obj1, edgar)** and **partof(obj1, hollow)**), denoted by F_i, which are supplied by the user to guide problem solving. These are treated equally by the problem-solver, and for present purposes it is convenient to regard input facts as part of the domain theory. Derived facts also split into two types: *specific* ones F_s (eg **cup(obj1)**, **stable(obj1)**), which are created by the application of operators during the course of solving a particular problem, and *general* ones F_g (eg **cup(X)**, **stable(X)**), which are non-ground versions of elements of F_s[3]. Only the first kind are relevant to the discussion of subsections 3.3 and 3.4; general derived facts will appear later when generalizing explanations. Note that F_s includes some of the domain and input facts, namely those relevant to the problem at hand.

To guide its work a problem solver needs to be given something to seek out: a *goal*. It receives this either from the user or from another part of the system. There are two types of goal: specific ones $\mathcal{G}_s$ and general ones $\mathcal{G}_g$. They differ in that only the latter may contain uninstantiated variables. For example, the general goal in Figure 2 is **cup(X)**, while the specific one is **cup(obj1)**, and facts about **obj1** are included in the domain theory. "Basic" and "tracing" problem solvers use the specific goal $\mathcal{G}_s$, while "generalizing" and "learning" problem solvers use the general goal $\mathcal{G}_g$ as well. Note that goals are also derived facts: $\mathcal{G}_s \in F_s$ and $\mathcal{G}_g \in F_g$. If $\mathcal{G}_s \notin F_s$, then the problem would be insoluble, for the solver would not be able to derive the goal.

It is often possible to convert a specific goal into a general equivalent by uninstantiating all arguments[4]. Conversely, a specific goal may be derivable from a general one by invok-

ing a problem solver to search for any domain objects that satisfy the general predicate. Consequently, whether an EBL system is provided with a general or a specific goal, or both, depends more on the application environment and desired mode of operation than on any fundamental considerations. In this paper we assume that both are provided.

3.3 Implementing basic problem solvers

During operation of the problem solver, operators $Op_s \in \mathcal{O}$ are applied to facts $F_i \cup F_d$ and previously derived facts F_s to generate new members of F_s. The stream of facts that is produced is called the *specific derivation stream.* Problem solvers that match consequents of rules with facts in F_s are called *backward-chaining,* while those that match antecedents with facts in F_s are *forward-chaining.* We deal with each type in turn.

Backward chaining. A resolution theorem prover is an important kind of problem solver which represents background knowledge in terms of first-order logic and proves theorems using the resolution principle. The PROLOG interpreter is a particular incarnation of such a scheme. Background knowledge is represented in the form of implications with conjunctive (or empty) antecedent and unit consequent. The problems themselves are given to the system as goals to prove. PROLOG is a backward-chaining interpreter because it works back from the goal, applying rules recursively to create subgoals until the subgoals are trivially solvable.

Figure 5a shows how to write a problem-solver, in the form of a meta-interpreter, in PROLOG. The first rule decomposes sequences of conjoined terms in a rule (in PROLOG, comma denotes conjunction) and executes each in turn. Here and elsewhere a prime denotes a sequence of terms of the type indicated; thus F_s denotes something that is or will be instantiated to a specific fact, while F_s' denotes a sequence of such items. The predicate **fact(A)** unifies **A** with a fact in the domain theory database, if this is possible[5]. **rule(A ⇒ B)** finds rules in the domain theory of the form **A ⇒ B** (where **A** may be a sequence of conjoined terms), so the third rule recursively evaluates domain rules.

Forward chaining. A second important class of problem solvers comprise forward-chaining production systems such as OPS-5. In many ways these can be viewed as reverse PROLOGs. Instead of working from the goal progressively back to the facts, they work forward from a set of input facts towards a goal. Matching is done on antecedents of rules rather than on consequents.

Figure 5b shows a rudimentary forward-chaining problem solver written in PROLOG. It is given a set of facts in the domain theory database, and a goal to seek. Domain rules are applied to the facts and to derived facts until the goal is reached. Whenever a rule is triggered, its consequent is added to the database, thereby augmenting the domain theory. The new facts may trigger other rules later. Rule firing continues until the goal is generated or until no further rules can be fired, in which case the goal is not solvable.

The first rule in Figure 5b checks to see whether the specific goal is contained within the facts currently in the database; if so, the problem is solved. As before, predicates **fact** and **rule** find facts and rules of the domain theory; **matches** just checks whether a conjunction of facts is supported by the theory. The second rule chooses a production rule and tries to apply it to the current database. If it matches, its consequent is asserted into the database using **save-fact**, and the cycle continues. If the rule does not trigger, a new one is tried.

3.4 Problem solvers which record a trace

During operation of a problem solver, many facts are created in the specific derivation stream F_s, some of which may turn out not to contribute to the solution of the problem. The $f \in F_s$ that are used in the solution of the goal form the basis of an *explanation* of how the example satisfies the goal. Explanations take the form of a tree whose interior nodes are derived facts, with primitive facts at the leaves and the goal as the root. Each arc in the tree corresponds to an operator. Because there are two types of derived fact, two types of explanation are distinguished: specific ones E_s with nodes in F_s, and general ones E_g with nodes in F_g (in the present section we encounter only the first kind). Figure 6 illustrates the relationship between different kinds of specific fact, specific goal, and specific explanation.

During the course of its work, any problem solver must derive all facts that are needed in a specific explanation E_s, since all nodes in E_s are drawn from F_s and it certainly derives all of F_s. Consequently if a basic problem solver is modified to trace its own operation, saving the facts it derives while creating a solution, the explanation can be reconstructed from this record. We call such a mechanism, which calculates σ during the course of solving a problem, a "tracing" problem solver.

It is not necessary to store facts derived in failed branches of the problem-solving process. In backward chainers, this corresponds to suppressing all memory of branches that fail and cause backtracking. However, forward chainers cannot know whether the facts they produce will be relevant to the deduction until the final goal is reached, and so an explanation needs to be recorded with each fact.

Backward chaining. In a backward-chaining problem solver implemented by backtracking, a second argument can be added to the definition of **solve** to collect the explanation. This records all subgoals that are solved during the course of the proof. Figure 7a shows how to do this. The first rule maintains conjunctions of subgoals as they are executed, while the last two add facts and rules to the explanation whenever they are retrieved from the database.

Forward chaining. As in the backward chainer, a second argument must be added to collect the explanation, and Figure 7b illustrates what is required. Notice that in this case explanations are collected as part of the fact database. Instead of simply storing the specific derived fact

```
fact( stable( obj1 ) ),
```

the explanation is stored as well:

```
fact( stable( obj1 ), ( partof( obj1, base ),
                        isa( base, bottom ),
                        flat( base ) ) ).
```

Since these explanations are incrementally compounded, the one associated with the goal fact will be E_s, the complete explanation tree. This is returned by the first rule of **solve**, and the predicate **matches** is augmented to give the explanations of all facts that are matched.

3.5 Problem solvers which generalize

The explanation that is returned by a "tracing" problem solver, from which a general rule must be constructed, is specific. The function $\gamma(E_s, \mathcal{G}_g)$ produces a generalized explanation from it. The second argument, the general goal $\mathcal{G}_g$, forms the root of the general explanation. Simply substituting variables for constants in E_s will over-generalize the explanation tree, and $\mathcal{G}_g$ must be used to further constrain the generalization. The standard technique of *goal regression* provides a way of computing γ. Basically, this constructs an appropriately generalized fact F_g corresponding to each specific fact F_s of E_s, and creates a general explanation E_g by exchanging the F_s for these F_g.

In order to calculate the generalized explanation on the fly, the problem solver should remember the general facts F_g corresponding to those specific facts F_s necessary to create a specific explanation. The facts F_g are calculated explicitly, in parallel with the normal computation of F_s. The simplest way to do this is to include a *general* derivation stream along with the specific derivation stream used before. Recall that previously, operators $Op_s \in \mathcal{O}$ were applied to facts $F_i \cup F_d$ and previously derived facts F_s to obtain new members of F_s. Now a parallel general derivation stream is needed to create general facts in F_g, as illustrated in Figure 8.

To accomplish this, the specific operator Op_s that is to be applied is copied (before being instantiated) to create Op_g. Op_g then is applied to existing members of F_g. Since if it succeeds Op_s matches some subset of F_s, Op_g will likewise match the corresponding subset of F_g. Note that the mapping from F_s to F_g may be many-to-one, and so the correspondence between related members of F_s and F_g must be stored explicitly.

Thus backward- and forward-chaining problem solvers which generalize can be created by supplementing the basic problem solver with a general derivation stream and recording those members of F_g that lead to the solution of the goal. In the backward-chaining case, F_g is initialized to contain $\mathcal{G}_g$ alone. For forward-chaining, the first members of F_g are the antecedents of the Op_g corresponding to those Op_s that are applied to members of $F_i \cup F_d$.

Backward chaining. A "generalizing" backward-chaining problem solver maintains two goals: a specific one $\mathcal{G}_s$ and a general one $\mathcal{G}_g$, along with a generalized explanation tree E_g. This effectively adds the general derivation stream alongside the specific one. All unification can be performed by the PROLOG interpreter, and Figure 9a shows how to do this. (An independently-created system called PROLOG-EBG [Kedar 87] bears a strong resemblance to this program.)

The first rule takes care of conjunctions in the usual way. The second checks the *specific* goal against the database to see if the proof has terminated. The third finds a rule which matches the *general* goal regardless of specific variable instantiations. The procedure **matched-copy** creates a copy of this rule and unifies the copy with the specific goal, while leaving the general form of the rule untouched. The procedure then recurs, with the specific (instantiated) and general (uninstantiated) body of the rule as its first two arguments.

To use the program, **solve** should be called with both specific and general goals; the third argument returns the correctly generalized proof tree. For instance, in the "cup" domain one could invoke

```
solve( cup(obj1), cup(A), E )
```

and execution would bind **E** to a tree like

```
cup(A)
    liftable(A)
        light(A)
        partof(A, B)
        isa(B, handle)
    stable(A)
        partof(A, C)
        isa(C, bottom)
        flat(C)
    open-vessel(A)
        partof(A, D)
        isa(D, concavity)
        pointing(D, upward).
```

Forward chaining. Augmenting forward-chaining systems to return a general explanation is not so straightforward. As Figure 9b shows, a general derivation stream is included along with the specific one. As before, the general goal is included as an argument and will form the root of the explanation tree. However, in the forward chainer this argument is not decomposed, but is used to constrain E_g once the specific goal is generated.

In forward chainers, explanations are recorded with individual facts. A fact, which is written **fact(** F_s, F_g, E_g **)**, has three components: the specific and general form, and the explanation tree E_g that led to its production. An example is

```
fact( stable( obj1 ), stable( X ), ( partof( X, Y ),
                                     isa( Y, bottom ),
                                     flat( Y ) ) ).
```

Given that facts are stored in this manner, a specific production rule can be applied to the specific derivation stream (the first component of the fact), and the corresponding general production rule, created by **functor-copy**, applied to the general derivation stream (the second component).

The **matches** predicate returns the conjunction of the explanations of all facts that contributed to the rule's triggering. If **matches**(F_s, F_g, E_g) is called and F_s matches some derived facts, an explanation will already be stored with the derived facts which can be returned as the new explanation for F_s and F_g. However, facts at the lowest level, namely those in $F_d \cup F_i$, will have no associated explanation, for example

```
fact(isa(lug, handle), _, none).
```

Then the explanation is just the corresponding fact in F_g which is given as an argument to **matches**, in this case, **isa(X, handle)**.

3.6 Problem solvers which learn

The final task of EBL, $\alpha(E_g, \Omega)$, is to prune the generalized proof tree and create a rule from it. To embed this within the problem solver it is necessary to consider the form of the operationality criterion Ω. We assume that operationality is defined by specifying which facts in the domain can participate in the final generalized rule. In the "cup" domain, for example, the operational predicates are simply the structural ones (light, partof, pointing, ...), and not the functional ones (liftable, stable, open-vessel, ...).

A "learning" problem solver prunes the proof on the fly and returns an operational clause on exiting at the top level. No explicit explanation tree need be maintained at all. The function α is applied by checking facts F_g in E_g against Ω to determine which ones should participate in the learned rule. Each fact is already considered by the previously-derived "generalizing" problem solver, and so it suffices to apply Ω to it before storing it in E_g. If it passes the test it becomes part of the antecedent of the learned rule, otherwise it is ignored. The consequent of that rule is the general goal.

Backward chaining. To embed α in a backward-chaining system, all goals that are generated are tested for operationality. The resulting system, called MEL/B, appears in Figure 10a. Operationality testing is done by the **operational** predicate. The **solve** function returns the rule $F'_g \Rightarrow F_g$ whose antecedent is the generalized goal $\mathcal{G}_g$, written here as F_g, and whose consequent is a sequence (hence the prime) of operational predicates in the form of general derived facts. The **osolve** function, which does the work and returns OF'_g, is very

similar to **solve** of Figure 9. However, when a rule is sought in the database it checks to see if the head clause is operational; if it is, it serves as part of the rule without further ado.

Forward chaining. Embedding α within a forward-chaining problem solver is just as easy. Instead of remembering general explanations, only the operational parts are remembered. As before, facts are written as **fact(** F_s, F_g, *OpJust* **)** with three components, the specific and general forms and an explanation part. The third part does not include the complete explanation, however, but only those operational items in E_g that might contribute to the final rule. We call such items ***operational justifications.*** An example fact is

```
fact( cup( obj1 ), cup( X ), ( ( partof( X, Y1 ), isa( Y1, concavity ), pointing( Y1, upward ) ),
                               ( partof( X, Y2 ), isa( Y2, bottom ), flat( Y2 ) ),
                               ( light( X ), partof( X, Y3 ), isa( Y3, handle ) ) ).
```

As it happens this is the goal fact for the cup example, and the final rule contains the second argument as consequent and the third as antecedent.

Figure 10b shows the implementation of MEL/F, a model for EBL using forward chaining. The **matches** function returns the operational justifications of all facts that contributed to the rule's triggering. These are used by **prune**, along with the Ω predicate and the antecedent (general version) of the rule just triggered, to determine the consequent's operational justifications. Ω is applied to each $f \in F_g$ in the antecedent, and if it fails the operational justifications for f are retained. Otherwise, f becomes part of the operational justification for the consequent.

Augmenting the facts in this way allows the final operator of the first clause to be pieced together simply by examining the second and third arguments of the goal fact. The second argument is the consequent and the third the antecedent of the rule.

4 More complex problem solvers

The rudimentary backward- and forward-chaining systems considered in the last section are somewhat simplistic. Real-world problem solvers tend to be more complex. More sophisticated domains — planners, story understanders, programming-by-example schemes — present new challenges because operators are more than just antecedent/consequent rules, and so it is not completely straightforward to build the final operator which encapsulates the result of learning. Moreover, our earlier decision-function model of the operationality criterion may be inappropriate or overly simplistic [DeJong 86]. Nevertheless, the same idea of embedding learning within the problem-solving mechanism can still be used to enhance these more sophisticated systems.

This section presents a case study of *planners*, as an example of a more complex type of problem solver. We first discuss the planning problem, then build a basic planner,

and finally show how to augment it with EBL. Although skeleton implementations are presented as before, space does not permit a detailed discussion of the code. The resulting program, called MEL/P and shown in Figure 13, supports our contention that only small modifications are required to endow general problem solvers with learning capabilities.

4.1 The planning domain

Whether we consider humans, robots, game-playing programs or search systems, the ability to plan ahead is a hallmark of intelligent behavior. Planning involves reasoning about how to achieve a desired state, based on knowledge of the current situation and the various ways that changes can be made to the world. More formally, given a starting state S, the planning problem is to find a plan Φ which leads to a state that satisfies a set of goals Γ. The plan will be a sequence of operators or actions that modify the world state:

$$\Phi = [O_1, O_2, O_3, ..., O_n].$$

For example, suppose that in the celebrated blocks world the original situation in Figure 11a must be transformed into that of Figure 11b. The starting state is

$$S = [\ \text{on(a,d)}, \text{on(b,e)}, \text{on(c,f)}, \text{clear(a)}, \text{clear(b)}, \text{clear(c)}\ ],$$

and the goal is to get a on b and b on c:

$$\Gamma = [\ \text{on(a,b)}, \text{on(b,c)}\ ].$$

Here is a plan which, when applied to S, produces the configuration of Figure 11b and thereby satisfies Γ:

$$\Phi = [\ \text{move(b,e,c)}, \text{move(a,d,b)}\ ]$$

— a move of block b (which is currently on e) to the top of c, followed by a move of block a (which is currently on d) to the top of block b.

Finding Φ for arbitrary Γ and S involves trying out meaningful combinations of operators O (O=move(b,e,c) for example) to see if they lead to states that satisfy Γ. Many of the original domains on which EBL was applied incorporated planning (eg LEX2 [Mitchell 83]). In most of these, domain knowledge was deeply embedded in the mechanism for doing the planning, and so a separate process was required to generalize explanations.

4.2 Basic planners

Our basic planner is derived from STRIPS [Fikes 72], the grandfather of all planners, and is simple enough to permit a full description here. Following STRIPS, operators $\mathcal{O}$ are not just antecedent/consequent pairs but have three parts. The first is a set of preconditions that must be met in order to apply the operator. The other two contain postconditions that specify the state of the world after application of the operator, effectively splitting the consequent in two. The *add list* dictates what new conditions will be added to the world, while the *delete list* gives those previously-existing conditions that will have been invalidated by the operator.

The three parts of an operator are specified by

$pre(\mathcal{O})$ — set of preconditions
$add(\mathcal{O})$ — set of positive (additive) effects on the world state
$del(\mathcal{O})$ — set of negative (subtractive) effects on the world state.

For example, one definition of the above-mentioned "move" operator is

pre(move(x,y,z)) = [on(x,y), clear(x), clear(z)]
add(move(x,y,z)) = [on(x,z), clear(y)]
del(move(x,y,z)) = [on(x,y), clear(z)]

The more complex operator structure affects the nature of the α function needed by a planner incorporating EBL.

Some STRIPS-like planners use a backward-chaining technique called *goal regression* [Waldinger 77], that we employ in our system, which starts with the goal and creates a series of subgoals, the last of which is completely satisfied by the initial state. The reduction of a goal to subgoals is done on the basis of operator descriptions. An operator is chosen and a subgoal generated such that when the operator is applied to any state satisfying the subgoal, the resulting state will satisfy the goal. Note that the goal typically contains several conjoined parts (recall the example given earlier), and any operator can be chosen so long as

- some part of the goal matches something in its add list
- no parts of the goal match anything in its delete list.

The first condition ensures that the operator aids in achieving the goal, while the second ensures that it does not simultaneously invalidate other components of it. Producing the appropriate subgoal is called "regressing" the goal through the operator, and involves including all of the operator's preconditions in the subgoal, as well as all predicates in the goal that do not match anything in the add list.

Unfortunately, doing simple regression through general operators may create disjunctive goals (see [Nilsson 80], pp 288-292). This is unacceptable since goals in our problem solvers must be conjunctive. One way to solve the problem is to instantiate operators before they are used. First, the general operator is retrieved from the database. Next, an instantiation of it is guessed on the basis of the initial state and the objects in the world. The guess is submitted to a consistency check and if it fails, a new one is tried. Once a valid instantiation of the operator is obtained, the goal is regressed through the operator to obtain a new subgoal as described above.

The resulting basic planner is given in regular PROLOG notation in Figure 12.

4.3 Planners that learn

The planner described above searches for a plan each time a goal state is requested. To avoid duplicating this work whenever a similar goal is sought, it can be extended to return generalized rules, or *macro operators* ("macrops" for short), which encapsulate the relevant parts of the plan. A macrop is created by composing other operators. Since the basic planner works by composing operators, it is not difficult to modify it to return a macrop as well as a plan. In fact, the macrop's precondition is just the last subgoal that needed to be solved, and its add and delete lists are just the union of the add and delete lists of the operators used in the plan[6]. The macrop thus created is stored and can be used in future plans, either by itself or with other operators.

Since operators are instantiated before use, a macrop formed by concatenating them would be fully instantiated. Unfortunately, this means that it could only be used in the same initial state as was used to create it. Actually, there is some latitude for variation even when all operators are fully instantiated. For example, adding an irrelevant block to the configuration would not render the macrop inapplicable, just as adding an irrelevant rule to a forward- or backward-chaining interpreter would not deny an existing conclusion. However, EBL seeks to generalize more deeply, for example by making the macrop insensitive to such things as names. This requires working with operators that are not fully instantiated.

Using the ideas developed in the previous section, the basic planner is first modified by augmenting it with a general derivation stream. This involves adding a general goal argument and regressing general goals and subgoals through a general copy of the current operator to create new facts (subgoals) in F_g. Care must be taken when regressing general goals because if operators and goals are both general, there is more opportunity for predicates to match than in the specific case. It is important to match and regress the same predicates as in the specific derivation stream, and so the **matches** and **regressed** predicates are modified to ensure that general matching proceeds exactly in parallel with specific matching.

Remembering general operators instead of the specific versions yields a general plan

E_g. This must be turned into a general rule or macrop using the analogue of the α function. Operationality testing is not an issue in this simple planner — all operators are operational — and so Ω is unnecessary. However, α still has the job of piecing together the final macrop's precondition, add, and delete lists. This can be done quite naturally within our implementation. MEL/P, as our learning planner is called, keeps general versions of each component of the macrop. One argument is included to record the add list. During goal regression, it is filled by collecting together the add lists of the general operators used in the general derivation stream[7]. Another argument is used to collect the delete list, and contains the union of the delete lists of the general operators used in the general plan E_g. Finally, the preconditions of the macrop are formed by remembering the last subgoal in the general derivation stream.

Augmenting the basic planner in this manner gives the program of Figure 13. If necessary, a notion of operationality of predicates could be incorporated simply by applying Ω before storing components of the macrop.

5 Conclusions

We have argued that problem solvers can easily be extended to give them learning abilities, and supported this contention with several specific illustrations. Of course, there are reasons why one might not want to augment a problem solver in the manner proposed. For one thing, modifying existing code is always very difficult. However, adding learning abilities even within the standard EBL paradigm requires modifying the problem solver to obtain a specific trace, and as we have seen it is only a small further step to go all the way and create a learning problem solver.

Another question concerns efficiency. At first glance, it seems that the method presented here may be more efficient than the traditional multi-pass technique since it requires only one pass and stores no intermediate explanations. However, maintaining the general derivation stream increases the computational load and this makes it hard to compare the resources required by the two approaches. In any case, the only differences are just constant overheads in time and space. Besides, the real need is for efficiency in understanding, not computation; this places emphasis on elegance and economy of ideas.

Finally, our strategy of making the problem solver do all the work seems to fly in the face of modern ideas of modularity. But another principle of software engineering is *localization*: putting things that belong with each other together. We believe that the issue of learning should not be sundered from problem solving and the performance system, but forms an intimate part of it.

Introspection — admittedly a risky business! — seems to indicate that people do not have to make special efforts to learn something from performing tasks. We do not normally undertake *post hoc* analyses of our problem-solving activities. Learning seems more like a byproduct of the problem-solving process itself. This research puts the learning back where it belongs, right in the heart of the problem solver.

Acknowledgements

We gratefully acknowledge the roles that David Hill and Bruce MacDonald have played in helping us to develop these ideas, and the stimulating research environment provided by the Calgary Machine Learning Group. Thanks also to Rosanna Heise who provided valuable comments on early drafts of this paper. This research is supported by the Natural Sciences and Engineering Research Council of Canada.

References

[Chien 88] Chien, S. "A Framework for Explanation-Based Refinement." *Proc AAAI Spring Symposium Series on Explanation Based Learning*, pp137-141, March, 1988.

[Clancey 88] Clancey, W.J. "Detecting and Coping with Failure." *Proc AAAI Spring Symposium Series on Explanation Based Learning*, pp22-26, March, 1988.

[Cohen 88] Cohen, W., Mostow, J., Borgida, A. "Generalizing Number in Explanation-Based Learning." *Proc AAAI Spring Symposium Series on Explanation Based Learning*, pp62-72, March, 1988.

[DeJong 86] DeJong, G., Mooney, R. "Explanation-Based Learning: An Alternative View." *Machine Learning*, Vol 1, No 2., pp145-176, Kluwer Academic Publishers, Boston, 1986.

[Ellman 88] Ellman, T. "Explanation-Directed Search for Simplifying Assumptions." *Proc AAAI Spring Symposium Series on Explanation Based Learning*, pp95-99, March, 1988.

[Ellman 85] Ellman, T. "Generalizing Logic Circuit Designs By Analyzing Proofs of Correctness." *Proc 9th Intl. Joint Conf. on A.I.*, Los Angeles, CA, pp643-646, 1985.

[Fikes 72] Fikes, R.E., Hart, P.E., Nilsson, N.J. "Learning and Executing Generalized Robot Plans." *Artificial Intelligence 3*, pp251-288, 1972.

[Gupta 88] Gupta, A. "Significance of the Explanation Language in EBL." *Proc AAAI Spring Symposium Series on Explanation Based Learning*, pp73-77, March, 1988.

[Hill 87] Hill, W.L. "Machine Learning for Software Reuse." *Proc 10th Intl. Joint Conf. on A.I.*, Los Angeles, CA, pp338-344, Morgan Kaufmann, 1987.

[Hunter 88] Hunter, L. "Explanation-Based Discovery." *Proc AAAI Spring Symposium Series on Explanation Based Learning*, pp2-6, March, 1988.

[Kedar 87] Kedar-Cabelli, S. T., McCarty L. T. "Explanation-Based Generalization as Resolution Theorem Proving." *Proc of the 4th Intl. Machine Learning Workshop*, Irvine, CA, pp383-389, Morgan Kaufmann, 1987.

[Mitchell 86] Mitchell, T., Keller R., Kedar-Cabelli, S. "Explanation-Based Generalization: A Unifying View." *Machine Learning*, Vol 1, No 1., pp47-80, Kluwer Academic Publishers, Boston, 1986.

[Mitchell 83] Mitchell, T. "Learning and Problem Solving." *Proc 8th Intl Joint Conf on AI*, Morgan Kaufmann, Karlsruhe, West Germany, pp1139-1151, 1983.

[Mooney 85] Mooney, R.J., DeJong, G.F. "Learning Schemata for Natural Language Processing." *Proc 9th Intl. Joint Conf on A.I.*, Los Angeles, CA, pp681-687, 1985.

[Mooney 88] Mooney, R. "Generalizing the Order of Operators and Its Relation to Generalizing Structure." *Proc AAAI Spring Symposium Series on Explanation Based Learning*, pp84-88, March, 1988.

[Mostow 87] Mostow, J., Bhatnagar, N. "Failsafe: A Floor Planner that uses EBG to Learn from its Failures." *Proc 10th Intl. Joint Conf. on A.I.*, Milan, Italy, pp249-255, Morgan Kaufmann, 1987.

[Nilsson 80] Nilsson, N.J. *Principles of Artificial Intelligence.* pp. 288-292, Tioga Publishing Company, Palo Alto, CA, 1980.

[Pazzani 87] Pazzani, M. "Inducing Causal and Social Theories: A Prerequisite for Explanation-based Learning." *Proc 4th Intl. Machine Learning Workshop*, Irvine, CA. pp230-241, 1987.

[Shavlik 88] Shavlik, J. "Issues in Generalizing to N in Explanation-Based Learning." *Proc AAAI Spring Symposium Series on Explanation Based Learning*, pp78-83, March, 1988.

[Shavlik 85] Shavlik, J. "Learning About Momentum Conservation." *Proc 9th Intl. Joint Conf. on A.I.*, Los Angeles, CA, pp667-669, 1985.

[Silver 88] Silver, B. "A Hybrid Approach in a Imperfect Domain." *Proc AAAI Spring Symposium Series on Explanation Based Learning*, pp52-56, March, 1988.

[Silver 83] Silver, B. "Learning Equation Solving Methods From Worked Examples." *Proc of the 2nd Intl. Machine Learning Workshop*, Urbana, Ill., pp99-104, 1983.

[Waldinger 77] Waldinger, R. "Achieving Several Goals Simultaneously." *Readings in Artificial Intelligence* N. Nilsson and B. Webber (Eds.), pp250-271, Tioga, Palo Alto, CA, 1981.

[Winston 83] Winston, P.H., Binford, T.O., Katz, B., Lowry, M., "Learning Physical Descriptions from Functional Definitions, Examples and Precedents." *Proc National Conference on Artificial Intelligence*, Washington, D.C., pp433-439, August, 1983.

Endnotes

1. In this simple example, the speed advantage of using the new rule is negligible. In other situations, the learned rule will bypass substantial search.

2. Not to be confused with the "generalizing" problem solvers discussed below.

3. In some cases, a fact in F_s will be as general as possible already and cannot contain variables; in this case the predicate in F_s is identical to its twin in F_g.

4. However, ambiguity can occur with multi-argument predicates. For example, it is not clear whether the general goal corresponding to goo(a, a) should be goo(X, Y), goo(X, X), or even goo(X, a) or goo(a, X).

5. In practice, fact and rule will likely both be implemented using the PROLOG built-in predicate clause.

6. In some planning problems this is overly simplistic, for add and delete lists may interact and this must be taken into account when obtaining the macrop's add and delete lists.

7. Again, we assume non-interacting add lists and delete lists. If necessary, interactions could be caught within the problem solver, but the issue is ignored for simplicity.

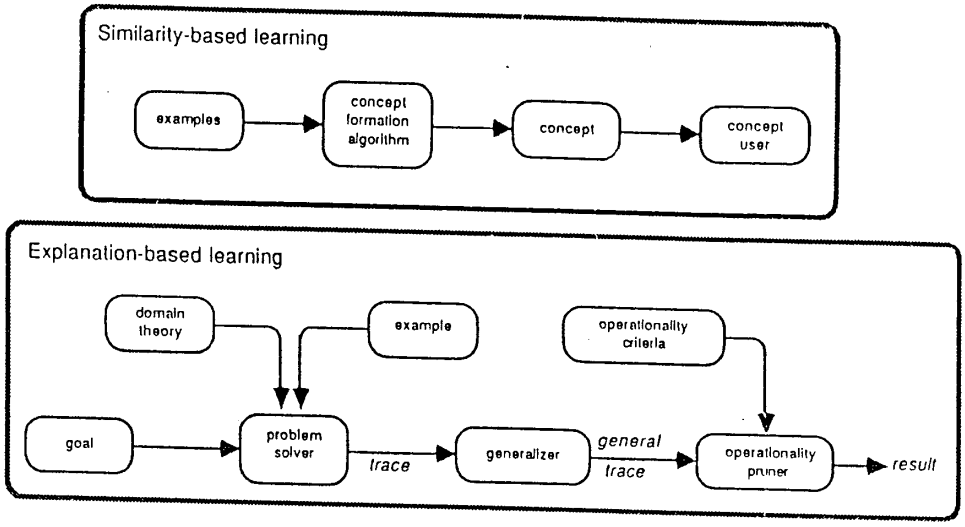

Figure 1: Similarity- and explanation-based learning

goal concept	cup(X) :- open-vessel(X), stable(X), liftable(X).
training example	owner(obj1, edgar). partof(obj1, hollow). pointing(hollow, upward). partof(obj1, base). flat(base). partof(obj1, lug). color(obj1, green). light(obj1).
domain theory: facts	isa(hollow, concavity). isa(base, bottom). isa(lug, handle).
domain theory: rules	liftable(X) :- light(X), partof(X, Y), isa(Y, handle). stable(X) :- partof(X, Y), isa(Y, bottom), flat(Y). open-vessel(X) :- partof(X, Y), isa(Y, concavity), pointing(Y, upward).
operationality criterion	operational(light). operational(flat). operational(partof). operational(pointing). operational(isa).
operational rule	cup(X) :- partof(X, Y1), isa(Y1, concavity), pointing(Y1, upward), partof(X, Y2), isa(Y2, bottom), flat(Y2), light(X), partof(X, Y3), isa(Y3, handle).

Figure 2: The "cup" domain

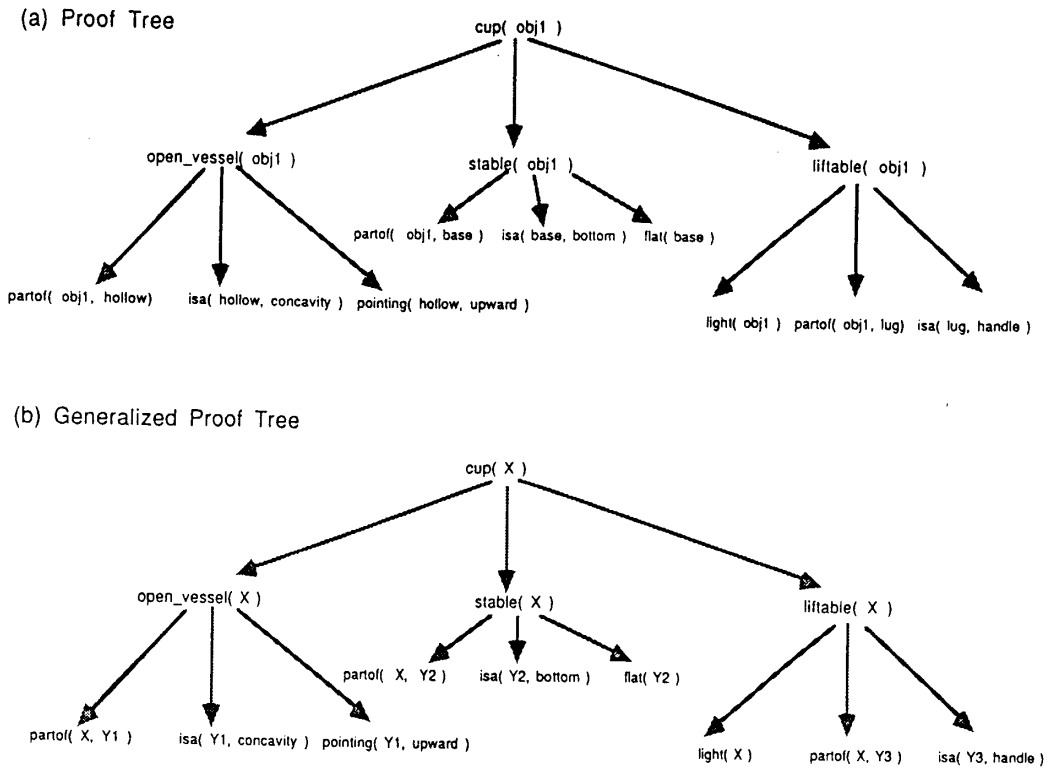

Figure 3: Explanation trees for the "cup" example

Type	Computes	Specific Derivation Stream	General Derivation Stream	Trace Type	Discussed in Section
Basic	(σ)	√		none	3.3
Tracing	σ	√		specific	3.4
Generalizing	$\gamma\sigma$	√	√	general	3.5
Learning	$\alpha\gamma\sigma = \mu$	√	√	none	3.6

Figure 4: Types of problem solver

```
solve((F_s,F'_s))   :- solve(F_s), solve(F'_s)
solve(F_s)          :- fact(F_s).
solve(F_s)          :- rule(F'_s ⇒ F_s), solve(F'_s).

fact(F_s)           unifies F_s with a fact from the domain theory database

rule(F'_s ⇒ F_s)    retrieves a rule from the domain theory database
```

(a) Backward chaining

```
solve(G_s)          :- matches(G_s).
solve(G_s)          :- rule(Op_s), apply(Op_s), solve(G_s).

apply(F'_s ⇒ F_s)   :- matches(F'_s), save-fact(F_s).

matches((F_s,F'_s)) :- fact(F_s), matches(F'_s).
matches(F_s)        :- fact(F_s).

save-fact(F_s)      adds the fact to the domain theory database
```

(b) Forward chaining

Figure 5: Implementing basic problem solvers in PROLOG

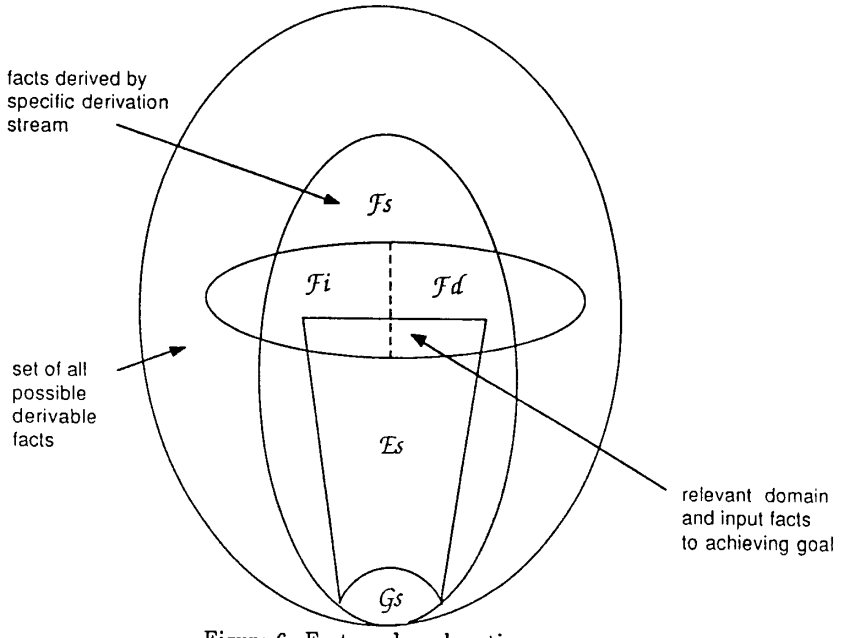

Figure 6: Facts and explanations

```
solve((F_s,F'_s),(E_s,E'_s))   :- solve(F_s,E_s), solve(F'_s,E'_s).
solve(F_s,true)                :- fact(F_s).
solve(F_s,(E'_s ⇒ F_s))        :- rule(F'_s ⇒ F_s), solve(F'_s,E'_s).
```

(a) Backward chaining

```
solve(F_s,E_s)                 :- matches(F_s,E_s).
solve(F_s,E_s)                 :- rule(Op_s), apply(Op_s), solve(F_s,E_s).

apply(F'_s ⇒ F_s)              :- matches(F'_s,E_s), save-fact(F_s,E_s).

matches((F_s,F'_s),(E_s,E'_s)) :- fact(F_s,E_s), matches(F'_s,E'_s).
matches(F_s,E_g)               :- fact(F_s,E_s).
```

(b) Forward chaining

Figure 7: Implementing "tracing" problem solvers

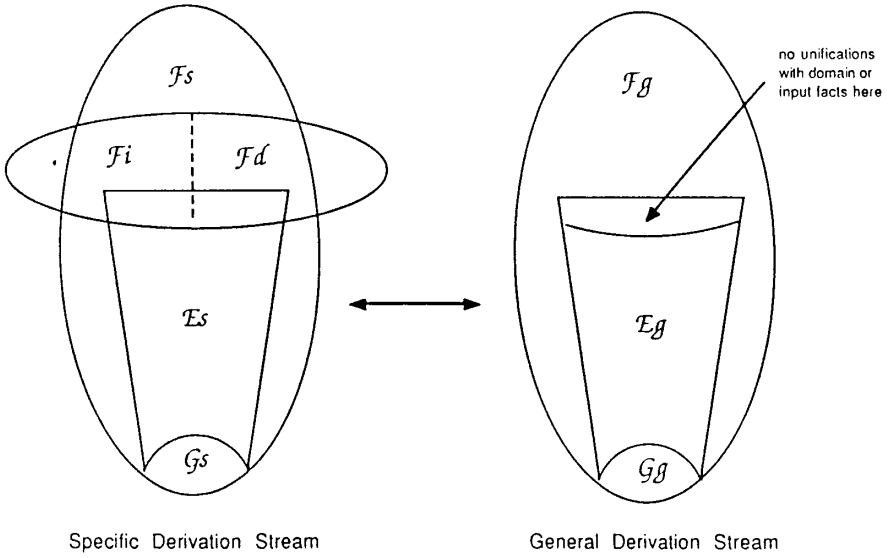

Figure 8: Specific and general derivation streams

solve(Specific goal $\mathcal{G}_s$,
General goal $\mathcal{G}_g$,
General explanation E_g)

solve($(F_s,F'_s),(F_g,F'_g),(E_g,E'_g)$)	:- solve(F_s,F_g,E_g), solve(F'_s,F'_g,E'_g).
solve(F_s,F_g,(true $\Rightarrow F_g$))	:- fact(F_s).
solve($F_s,F_g,(E_g \Rightarrow F_g)$)	:- rule($F'_g \Rightarrow F_g$), matched-copy($(F'_g \Rightarrow F_g),(F'_s \Rightarrow F_s)$), solve($F'_s,F'_g,E_g$).
matched-copy(A,B)	*creates a copy of A but with different variables*

(a) Backward chaining

solve(Specific goal $\mathcal{G}_s$,
General goal $\mathcal{G}_g$,
General explanation E_g)

solve(F_s,F_g,E_g)	:- matches(F_s,F_g,E_g).
solve(F_s,F_g,E_g)	:- rule(Op_s), functor-copy(Op_s,Op_g), apply(Op_s,Op_g), solve(F_s,F_g,E_g).
apply($(F'_s \Rightarrow F_s),(F'_g \Rightarrow F_g)$)	:- matches(F'_s,F'_g,E_g), save-fact(F_s,F_g,E_g).
matches($(F_s,F'_s),(F_g,F'_g),(E_g,E'_g)$)	:- fact(F_s,F_g,E_g), matches(F'_s,F'_g,E'_g).
matches(F_s,F_g,E_g)	:- fact(F_s,F_g,E_g).

(b) Forward chaining

Figure 9: Creating a generalized explanation tree on the fly

solve(Specific goal $\mathcal{G}_s$,
General goal $\mathcal{G}_g$,
Operationalized rule R)

solve(F_s,F_g,($OF'_g \Rightarrow F_g$))	:- osolve(F_s,F_g,OF'_g).
osolve((F_s,F'_s),(F_g,F'_g),(OF_g,OF'_g))	:- osolve(F_s,F_g,OF_g), osolve(F'_s,F'_g,OF'_g).
osolve(F_s,F_g,F_g)	:- fact(F_s).
osolve(F_s,F_g,F_g)	:- operational(F_g), rule($F'_g \Rightarrow F_g$), matched-copy(($F'_g \Rightarrow F_g$),($F'_s \Rightarrow F_s$)), osolve(F'_s,F'_g,_).
osolve(F_s,F_g,OF'_g)	:- not operational(F_g), rule($F'_g \Rightarrow F_g$), matched-copy(($F'_g \Rightarrow F_g$),($F'_s \Rightarrow F_s$)), osolve(F'_s,F'_g,OF'_g).

(a) MEL/B: Backward chaining

solve(Specific goal $\mathcal{G}_s$,
General goal $\mathcal{G}_g$,
Operationalized rule R)

solve(F_s,F_g,($F'_g \Rightarrow F_g$))	:- matches(F_s,F_g,F'_g).
solve(F_s,F_g,R)	:- rule(Op_s), functor-copy(Op_s,Op_g), apply(Op_s,Op_g), solve(F_s,F_g,R).
apply(($F'_s \Rightarrow F_s$),($F'_g \Rightarrow F_g$))	:- matches(F'_s,$OpJust'$), prune($OpJust'$,F'_g,$NewOpJust$), save-fact(F_s,F_g,$NewOpJust$).
matches((F_s,F'_s),(F_g,F'_g),($Just$, Js))	:- fact(F_s,F_g,$Just$), matches(F'_s,F'_g,Js).
matches(F_s,$F_g Just$)	:- fact(F_s,F_g,$Just$).
prune(OpJust, F_g, NewOpJust)	*keep old operational justifications, or make F_g the new ones*

(b) MEL/F: Forward chaining

Figure 10: Problem solvers which learn

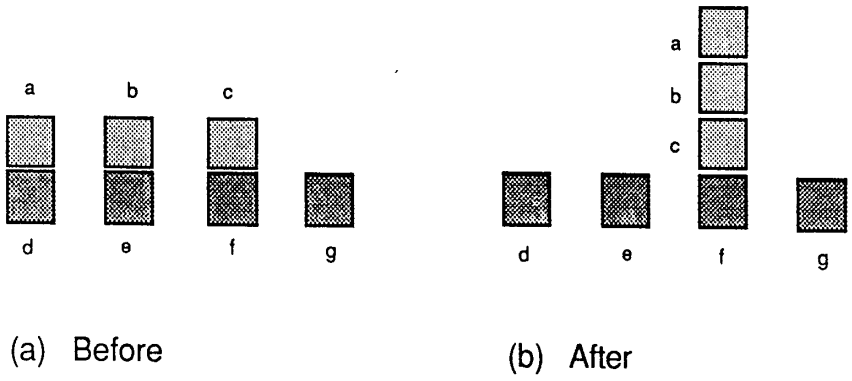

Figure 11: Blocks world: stack three

```
plan( Goal, Plan ) :-
        instantiation( Goal ),
        plan( Goal, [ ], Plan ).

plan( Goal, LastGoal, Plan ) :-
        instantiation( Goal ),
        consistent( Goal, LastGoal ),
        plan2( Goal, Plan ).

plan2( Goal, [ ] ) :-
        matches-initial-state( Goal ).

plan2( Goal, [Op | Plan] ) :-
        operator( Op ),
        matches( Goal, Op ),
        groundop( Op ),
        regressed( Goal, Op, NewGoal ),
        plan( NewGoal, Goal, Ops ).

regressed( Goal, Op, SubGoal ) :-
        pre( Op, PreConds ),
        add( Op, AddList ),
        del( Op, DelList ),
        intersection( Goal, DelList, [ ] ),
        pre( GenOp, GenPreConds ),
        add( GenOp, GenAddList ),
        setdifference( Goal, AddList, GoalLeft ),
        union( GoalLeft, PreConds, SubGoal ).
```

consistent(Goal,Lastgoal)	*checks for infinite loops and that Goal makes sense in current domain*
groundop(Operator)	*checks that the Operator is instantiated*
instantiation(Goal)	*instantiates Goal using knowledge from the asserted initial state*
matches(Goal,Operator)	*checks to see that the Operator is relevant to reducing the Goal*
matches-initial-state(Goal)	*checks that Goal is a subset of the initial state*
operator(Operator)	*retrieves an Operator from the domain theory database*
setdifference(A,B,C)	*set C contains the elements in A but not in B*
union(A,B,C)	*the two sets A and B are combined to form a single set C*

Figure 12: A basic planner

```
plan( Goal, GenGoal, Plan, Macrop ) :-
        instantiation( Goal ),
        plan( Goal, [ ], GenGoal, Plan, [ ], [ ], Macrop ).

plan( Goal, LastGoal, GenGoal, Plan, Add, Del, Macrop ) :-
        instantiation( Goal ),
        consistent( Goal, LastGoal ),
        plan( Goal, GenGoal, Plan, Add, Del, Macrop ).

plan2( Goal, GenGoal, [ ], Add, Del, Macrop ) :-
        matches-initial-state( Goal ),
        macrop( GenGoal, Add, Del, Macrop ).

plan2( Goal, GenGoal, [Op | Plan], Add, Del, Macrop ) :-
        operator( Op ),
        functor-copy( Op, GenOp ),
        matches( Goal, Op, GenGoal, GenOp ),
        groundop( Op ),
        regressed( Goal, Op, NewGoal, GenGoal, GenOp, GenNewGoal ),
        accumulated( GenOp, Add, Del, NewAdd, NewDel ),
        plan( NewGoal, Goal, GenNewGoal, Ops, NewAdd, NewDel, Macrop ).

accumulated( GenOp, Add, Del, NewAdd, NewDel ) :-
        add( GenOp, AddList ),
        del( GenOp, DelList ),
        append( AddList, Add, NewAdd ),
        append( DelList, Del, NewDel ).

regressed( Goal, Op, SubGoal, GenGoal, GenOp, GenSubGoal ) :-
        pre( Op, PreConds ),
        add( Op, AddList ),
        del( Op, DelList ),
        intersection( Goal, DelList, [ ] ),
        pre( GenOp, GenPreConds ),
        add( GenOp, GenAddList ),
        new-setdifference( Goal, AddList, GoalLeft,
                           GenGoal, GenAddList, GenGoalLeft ),
        union( GoalLeft, PreConds, SubGoal ),
        union( GenGoalLeft, GenPreConds, GenSubGoal ).
```

append(A,B,C) *appends lists A and B to form C*

macrop(Pre, Add, Del, Macrop) *pieces together a macrop given its preconditions, add list and delete list*

new-setdifference(A,B,C,GA,GB,GC) *same as setdifference but places general versions of elements in C into GC as well*

Figure 13: MEL/P: A learning planner