The Utility of EBL in Recursive Domain Theories

Devika Subramanian* and **Ronen Feldman**
Computer Science Department
Cornell University
Ithaca, NY 14853

Abstract

We investigate the utility of explanation-based learning in recursive domain theories and examine the cost of using macro-rules in these theories. The compilation options in a recursive domain theory range from constructing partial unwindings of the recursive rules to converting recursive rules into iterative ones. We compare these options against using appropriately ordered rules in the original domain theory and demonstrate that unless we make very strong assumptions about the nature of the distribution of future problems, it is not profitable to form recursive macro-rules via explanation-based learning in these domains.

Introduction

The power of explanation-based learning (EBL) has been established, [TMKC86], questioned [Min88, Gre89] and subsequently re-affirmed [Sha90] in the machine learning literature. The specific objectives of this paper are to demonstrate the conditions under which it is useful to use EBL to learn macro-rules in recursive domain theories. In these theories the compilation options range from forming rules for a specific number of unwindings of a recursive rule (as generated by standard EBL) to ones that generalize-to-n [Tad86,Coh88,Sha90,SC89]. We study the cost and benefits of forming fixed-length and iterative/recursive macros and compare it with the costs of using the original domain theory and subsets thereof, with appropriately ordered rules.

The impetus for this research was provided by Shavlik's [Sha90] existence proof for the utility of EBL in some recursive theories – the objective of our work is to find a generalization of his empirical results and to provide a model that will predict not only his results but also prove optimality conditions for some compilations of recursive theories. A secondary objective is to evaluate some of the performance metrics chosen by previous researchers for the evaluation of EBL [Sha90,Min88,Moo89]. For instance, one of our findings is that the utility metric adopted in [Sha90] makes implicit assumptions about the distribution of future problems.

We begin with a logical account of the macro formation process in recursive theories with a view to understanding the following questions. What is the space of possible macro-rules that can be learnt in a recursive domain theory? How can we quantitatively estimate the cost of solving a class of goal formulas in a recursive theory with macro-rules? How should macros be indexed and used so as not to nullify the potential gains obtained by compressing chains of reasoning in the original domain theory? Under what conditions is using the original domain theory with the rules properly ordered, better than forming partial unwindings of a recursive domain theory?

We will answer these questions in turn in the sections below relative to our cost model and then provide experimental data that validate our theoretical claims. The main results are

1. For a special class of linear recursive theories, reformulation into an iterative theory is the optimal compilation for any distribution of future queries.
2. Self-unwindings of recursive rules generated by some approaches to the problem of generalization to n have a harmful effect on the overall efficiency of a domain theory for most distributions of queries.
3. By ordering rules in the domain theory and by only generating unwindings that are not self-unwindings of recursive rules, we can get the same efficiency advantages claimed by some generalization-to-n methods.

We prove these results using our cost model and experimentally verify our results on the domain of synthesizing combinational circuits. Our paper is structured as follows. First, we outline the compilation options for recursive domain theories. Then we explain the generalization-to-n approach to compiling recursive theories. The utility problem in EBL especially for recursive theories is outlined in the next section along with a cost model for evaluating the various compilations. The main results of this paper are then stated and proved. Subsequently, we describe a series of ex-

*This research is supported by NSF grant IRI-8902721.

periments conducted in the circuits domain that validate our cost model and provide evidence for our theoretical results. We conclude with the main points of this paper and suggest directions for future work in the analysis of the utility of explanation-based learning.

Compilation Options in Recursive Domain Theories

In non-recursive domain theories like the cup and the suicide examples in the literature, the macro rules that can be generated are finite compositions of the given rules. The number of such macro rules is also finite. Recursive domain theories are more complicated because we can form an infinite number of rule compositions each of which could be infinite. Correct (or deductively justifiable) macros in any domain theory can be generated by unwinding the rules in a domain theory. We will confine ourselves to Horn clause theories for this paper. This is not an unreasonable assumption, because the theories used by experimental researchers in EBL are Horn.

Unwinding Rules

Definition 1 *An 1-level unwinding of a Horn rule* `Head` $\Leftarrow$ `Antecedents` *consists of replacing some clause* `c` *in the list* `Antecedents` *by the antecedents of another rule in the domain theory whose head unifies with* `c`. *An n+1 level unwinding of a rule is obtained by a 1-level unwinding of a rule that has been unwound n levels (n≥ 1).*

Definition 2 *A 1-level* self-unwinding *of a recursive Horn rule r:* `Head` $\Leftarrow$ `Antecedents` *is a 1-level unwinding where the clause* `c` *that is chosen for replacement in the* `Antecedents` *list unifies with* `Head` *of the rule r. An n+1 level self-unwinding is defined in the same way that n+1 level unwindings are.*

We can specialize an unwound rule by substituting valid expressions for variables in the rule. The set of all macro rules that EBL can learn in a Horn theory is the closure of that theory under the unwinding and substitution operations.

Consider the non-linear recursive domain theory about kinship. In this theory, x, y and z are variables; *Joe* and *Bob* are constants.

1. $ancestor(x, y) \Leftarrow father(x, y)$
2. $ancestor(x, y) \Leftarrow ancestor(x, z) \wedge ancestor(z, y)$
3. $father(Joe, Bob)$

Assume that all ground facts in our theory are *father* facts. One 2-level unwinding of rule 2 is the rule for grandfather shown below.

4. $ancestor(x, y) \Leftarrow father(x, z) \wedge father(z, y)$

A self-unwinding of rule 2 is the following

5. $ancestor(x, y) \Leftarrow ancestor(x, x_1) \wedge ancestor(x_1, z) \wedge ancestor(z, y)$

EBL methods pick rule unwindings and specializations directed by a training example. They compress chains of reasoning in a domain theory: the cost of looking for an appropriate rule to solve an antecedent of a given goal or subgoal (paid once during the creation of the proof of the training instance) is eliminated by in-line expansion of that antecedent by the body of a rule that concludes it. If there is more than one rule that concludes a given clause, then EBL methods make the inductive generalization from a single training instance that the rule used for solving the training example is the preferred rule for future goals.

Consider the recursive theory below that describes how expressions in propositional logic can be implemented as combinational circuits. This is part of a domain theory described in [Sha90]. *Imp(x,y)* stands for x is implemented by y. In the theory below, x, y, a and b are variables. wC refers to the wire C. The Ci's are constants. We will assume that wx unifies with wC_1 with x being bound to C_1. Rule `s1`. states that a wire is its own implementation. A typical goal that is solved by this theory is $Imp(\neg((\neg wC_1 \vee \neg wC_2) \vee \neg wC_3), x)$.

`s1`. $Imp(wx, wx)$
`s2`. $Imp(\neg(\neg x), y) \Leftarrow Imp(x, y)$
`s3`. $Imp(\neg(x \vee y), a \wedge b) \Leftarrow Imp(\neg x, a) \wedge Imp(\neg y, b)$
`s4`. $Imp(\neg(x \wedge y), aNandb) \Leftarrow Imp(x, a) \wedge Imp(y, b)$

Note that this theory is rather different from the kinship theory. It is a structural recursive theory, where the arguments to the *Imp* relation are recursively reduced into simpler subexpressions. The base case occurs when the the expression to be implemented is a constant or a negation of a constant.

Standard EBL on the goal above will produce the following rule.

$Imp(\neg((\neg wx_1 \vee \neg wx_2) \vee \neg wx_3), ((wx_1 \wedge wx_2) \wedge wx_3))$

This rule can be seen as being generated by first substituting the variablized goal expression to be synthesized into the first argument of the head of rule `s3`, and then unwinding the specializations of the antecedents of `s3` thus generated. Note that this is a very special purpose rule applicable only to those cases that share this sequence of substitutions and unwinding. For structurally recursive theories, an extension to standard EBL, called *generalization-to-n* has been proposed, that generalizes the *number* of unwinding steps.

Generalization to n

One such method, BAGGER2 [Sha90] produces the following rule from this example.

`s5`. $Imp(\neg(x1 \vee x2), y1 \wedge y2) \Leftarrow$
$(x1 = \neg wC1) \wedge (x2 = \neg wC2) \wedge$
$(y1 = wC1) \wedge (y2 = wC2)$
$\vee \quad (x1 = z1 \vee z2) \wedge (x2 = \neg wC2) \wedge$
$(y2 = wC2) \wedge Imp(\neg x1, y1)$
$\vee \quad (x1 = \neg wC1) \wedge (x2 = z1 \vee z2) \wedge$
$(y1 = wC1) \wedge Imp(\neg x2, y2)$
$\vee \quad (x1 = z1 \vee z2) \wedge (x2 = z3 \vee z4) \wedge$
$Imp(\neg x1, y1) \wedge Imp(\neg x2, y2)$

This can conceptually[1] be treated as 4 rules that can be generated by substituting in the values shown for the variables *x1*, *y1*, *x2* and *y2*. Note that the first disjunction can be seen as the rule
s6.$Imp(\neg(\neg wx1 \vee \neg wx2), wx1 \wedge wx2)$

s6 can be generated by unwindings that unify both antecedents of **s3** with the head of **s1** and propagating the bindings generated throughout the rule **s3**.

One way of understanding rule **s5** is that it unwinds a recursive rule like **s3** into cases based on examining the structure of the input expression which will be synthesized into a circuit. **s5** unrolls the input expression one more level than the original domain theory does. [SF90] examines how the other rules can be viewed as unwindings, our focus in this paper is in evaluating the utility of such rules over wide ranges of problem distributions which vary the percentage of problems solvable by **s5** and which vary the depth and structure of input expressions to be implemented. Note that we could have formed a similar rule for **s4**.

Another class of methods work by explicitly identifying rule sequences that can be composed, using a training example as a guide. One such method is due to Cohen [Coh88]. Instead of generating a new redundant rule in the manner of EBL he designs an automaton that picks the right domain theory rule to apply for a given subgoal during the course of the proof. We have modified his algorithm to actually generate a macro rule that captures the recursive structure of the proof. Here is the algorithm.

Algorithm R1

Inputs: Proof for a goal formula and a domain theory

Output: The simplest context-free (CF grammar) that generates the rules sequence in the proof of the goal formula.

1. Mark the ith node in the proof tree by a new symbol A_i. The content of this node is denoted by $T(A_i)$.
2. Re-express the proof tree as a list of productions: i.e. for proof node A_i with children $A_{i_1} \ldots A_{i_m}$, create a production $A_i \Rightarrow A_{i_1}, \ldots A_{i_m}$. Label the production by the name of the domain theory rule used to generate the subgoals of this proof node. This label is also associated with the head of the production A_i, and we will call it $L(A_i)$. We will denote the resulting CF grammar as G. This CF grammar is special because every symbol in the grammar appears at most twice.
3. G' = Minimize(G)
4. D' = Build-Rules(A_0) where A_0 is the start symbol for G'.

Algorithm Minimize

Inputs: A labeled context-free grammar of rule sequences.

Output: A minimal (in the number of non-terminals) context-free grammar that is behaviourally equivalent to the input grammar on the given goal.

1. $\forall$ A_i,A_j such that $L(A_i) = L(A_j)$ put A_i and A_j in the same equivalence class.
2. For each pair of productions in the input grammar $A_i \Rightarrow A_{i_1}, \ldots A_{i_m}$ and $A_j \Rightarrow A_{j_1}, \ldots A_{j_m}$, such that A_i and A_j are in the same equivalence class, place A_{i_k} and A_{j_k} in the same equivalence class, if $L(A_{i_k}) = L(A_{j_k})$ or if $L(A_{i_k})$ and $L(A_{j_k})$ are both recursive productions for the same predicate (as **s3** and **s4** are), for $1 \leq k \leq m$.
3. Eliminate structurally equivalent productions. Two productions $A_i \Rightarrow A_{i_1}, \ldots A_{i_m}$ and $A_j \Rightarrow A_{j_1}, \ldots A_{j_m}$, are structurally equivalent, if $L(A_i)$ is the same as $L(A_j)$ and if terms associated with corresponding symbols X and Y (X and Y could be A_i and A_j or A_{i_k} and A_{j_k}) in the production, namely T(X) and T(Y) are structurally equivalent. T(X) and T(Y) are by definition derived by substitution from the head of the same domain theory rule. For structural equivalence we require that the substitutions have the same structure.

Algorithm Build-rules(X)

Inputs: A labeled minimal CF grammar in which X is a non-terminal symbol.

Outputs: A rule for T(X), the domain theory term corresponding to X.

Collect all productions in the grammar $X \Rightarrow X_1, \ldots X_m$ that have X at their head. Let L_i be the set of labels of domain theory rules associated with the symbols that are in the same equivalence class as the symbol X_i. The rule corresponding to the production above is $T(X) \Leftarrow \bigwedge_{i=1}^{m} \text{check} - \text{unwind}(X_i)$.

Check-unwind does the following: If the cardinality of $L_i > 1$ then X_i is not unwound (since we have then more than one option). If N = 1 then we perform unwinding (and propagate the constraints found in the head to the body).

An example makes this algorithm clear. The proof tree for a goal constructed using the original domain theory is shown in Figure 1.

The CF grammar of rule sequences we get from this proof are the following.

$$
\begin{array}{llll}
A_1 & \Longrightarrow & A_2 \quad A_3 & \{\mathbf{s3}\} \\
A_2 & \Longrightarrow & A_4 \quad A_5 & \{\mathbf{s3}\} \\
A_3 & \Longrightarrow & A_6 & \{\mathbf{s2}\} \\
A_4 & \Longrightarrow & A_7 & \{\mathbf{s2}\} \\
A_5 & \Longrightarrow & A_8 & \{\mathbf{s2}\} \\
A_6 & \Longrightarrow & A_9 & \{\mathbf{s1}\} \\
A_7 & \Longrightarrow & A_{10} & \{\mathbf{s1}\} \\
A_8 & \Longrightarrow & A_{11} & \{\mathbf{s1}\}
\end{array}
$$

The final equivalence classes generated by the minimizing phase are $\{A_1, A_2, A_3, A_4, A_5\}$, $\{A_6, A_7, A_8\}$, and $\{A_9, A_{10}, A_{11}\}$. The minimized grammar is

[1] In our experiments we implemented the internal disjunction to prevent the overhead incurred due to backtracking by treating this as 4 separate rules

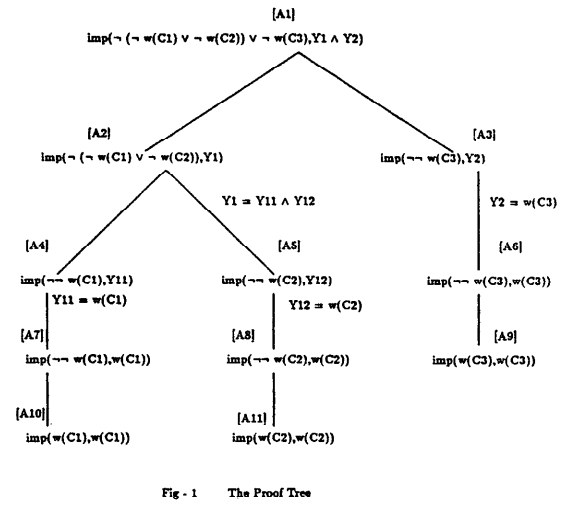

Figure 1: Proof Tree for Example Synthesis

$$\begin{array}{lcll} A_1 & \Longrightarrow & A_1\ A_1 & \{\texttt{s3}\} \\ A_1 & \Longrightarrow & A_6 & \{\texttt{s2}\} \\ A_6 & \Longrightarrow & A_9 & \{\texttt{s1}\} \end{array}$$

The only rule that is learned is the compression of s2 and s1 to one rule: s7. $Imp(\neg(\neg wx), wx)$ which is simply the compression of the base cases of the recursion. Rules which are self-unwindings of the original domain theory rules are not learned. R1 is an algorithm that generates useful generalizations-to-n. When applied to the blockworld example formulated in [Etz90], this algorithm learns the tail-recursive formulation of the rule for unstacking towers of height n.

The enumeration of the space of possible macro rules that can be learned by EBL as well as augmentations like generalization-to-n methods in terms of unwindings and specializations allows us to study two questions: what subset of these possible macros are *useful*, and how hard is it to learn that class of macros. Elsewhere [SF90] we study the former question, here we focus on the latter for the following four classes of unwindings/specializations for compiling a domain theory. These four classes are chosen because they have been experimentally tested in the EBL literature.

1. Keep the original domain theory (or a subset of the domain theory), ordered in such a way to optimise the set of goals we are interested in.
2. Augment domain theory by BAGGER2 style rules and order all the rules appropriately.
3. Augment domain theory by BAGGER2 style rules where the cases are indexed as a tree and order all rules appropriately.
4. Add only the rules learned by R1.

To better explain the four choices for compilation, we continue with the circuits example introduced earlier. For Option 1 we have a theory with rules s1, s2, s3 and s4 in that order. For Option 2 we have a theory with the rules s5, s1, s2, s3 and s4. For Option 3 we use the rules s5 with the cases indexed appropriately, s1, s2, s3 and s4. And for Option 4, we use the rules s7, s3, s1, s2 and s4.

Converting Recursion to Iteration

There is another possibility for optimising recursive theories: some linear recursions can be reformulated to a more efficient iterative form by the introduction of new auxiliary predicates (a classic example is the optimisation of the standard presentation of the Factorial function to a tail-recursive or iterative form). This class is interesting because optimatility results for this style of compilation under arbitrary distributions of future queries has been proven [War80].

An analysis of compilation strategies

In this paper we seek a dominance result: i.e., we wish to compare the costs of deriving a class of goals in the 4 compilations of recursive theories generated by EBL and generalization-to-n methods, against using the original domain theory (or a subset of it) appropriately ordered. This seeks to formalize and experimentally validate the intuition that for most cases, the cost of using an unwound recursive rule outweighs the cost of using the domain theory rules directly.

This work is part of a larger effort [SF90] in identifying cases in which macro rules provably reduce the effort a problem solver expends to solve a distribution of queries.

The Utility of Macro Rules

The utility problem in EBL is to justify the addition of macro rules by proving that for certain future query distributions and with some problem solvers, the benefit accrued by not having to perform a backtracking search through the domain theory for the right rule, is greater than the unification overhead incurred in establishing the antecedents of the macro rule. As Tambe and Newell [TN88] point out, EBL converts search in the original problem space to matching: we trade off time to pick the right rule to solve a sub-problem plus the time needed to solve the subproblem, against the time to establish the antecedents of the macro rule.

One way of casting this problem is to conceptually attach with every macro rule, two classes of antecedents: one elaborates the conditions under which it is correct to use that rule (this is phrased entirely in terms of the vocabulary of the domain), another class of antecedents elaborates the conditions under which it is computationally beneficial to use that rule (this is phrased entirely in terms that describe a problem solver, the distribution of future queries, the distribution of ground facts in the domain theory, as well as

performance constraints: time and space limitations). Learning is not mere accretion of new rules, we have to learn *when* to use the new rule. Most EBL systems learn just the correctness conditions and learn the default goodness conditions: every rule is unconditionally good. Minton's thesis work took a step toward identifying the goodness conditions by using axioms about the problem solver. His later empirical results [Min88] showed the negative effects the extra matching had on the overall problem solver performance on an interesting recursive domain theory.

Our analysis has the following form: we compute the cost of establishing a given class of goals in the original domain theory and compare it with the cost of doing the same in a domain theory augmented by a macro rule generated by standard EBL or generalization-to-n methods. We construct a cost model of exactly how rules are used by a problem solver. Since most of the empiricists use depth-first backward-chaining problem solvers, the Prolog model is quite adequate. In fact, we extend the Greiner and Likuski model to cover conjunctive and recursive domain theories [Gre89].

We illustrate our cost model in the context of the kinship example. We will assume that it costs i units to reduce a goal to its subgoals using a rule in the domain theory, and that it costs d units to check if a fact occurs in the domain theory. Let $C_{A_{xy}}$ denote the cost of solving an *ancestor* query with variable bindings x and y. Let the probability that q is in the domain theory be L_q. We abbreviate *ancestor(x,y)* by A_{xy} and *father(x,y)* by F_{xy}. Here x, y, z stand for variables.

$$C_{A_{xy}} = i+d+(1-Pr(L_{F_{xy}})[i+C_{A_{xz}}+Pr(A_{xz})*C_{A_{zy}}]$$

The probability that we succeed in proving a given *ancestor(x,y)* goal is

$$Pr(A_{xy}) = Pr(L_{F_{xy}})+(1-Pr(L_{F_{xy}}))[Pr(A_{xz})*Pr(A_{zy})]$$

Note that the cost equation is a recurrence equation. The base case occurs when when *father(x,y)* is in the domain theory: the cost then is $i + d$.

Assume now, that we have the following ground facts about *father* in theory T_1 that includes rules `1.` and `2.` from a previous section.
$father(c, d)$
$father(d, e)$
$father(e, f)$

The cost in T_1 for answering *Ancestor(c,f)* calculated using the equations above is is $7i + 5d$. This was a particularly simple case because we know that this computations terminate deterministically for the above database. and thus we can replace probabilities in the above equations by 0 or 1. The worst case occurs when the probabilities (0 or 1) work to our disadvantage and we explore the full search tree and the best case occurs when there is no backtracking over the rules. To analyze average case performance, we need to get estimates of the lookup probabilities above based on distributions of ground facts in the database, and solve the rather complex recurrence relations. In [SF90] we pursue simplifications of these recurrence relations that allow us to get qualitative cost measures. For this paper, we will analyse best and worst cases using our model to make the arguments for or against the various strategies.

Simplified Cost Models for Analyzing Recursive Compilations

Option 1: Keep the original domain theory

Let us calculate the cost in the original theory of proving a goal of the form $Imp(\neg(x \vee y), a \wedge b)$ where x and y are bound. The objective is to find bindings for a and b. Again, since the goal class is specified and since all recursion terminates on `s4`, we can analytically calculate the cost without doing the average case analysis that the equations allow us to compute.

We will have a finer grained accounting of i which is the cost to match against the head of a rule and generate the subgoals. For goals of the form $Imp(X, Y)$ let t be the cost of determining thata given rule rule applies. In our rule set, $t = 2$ because we only need to read the first two symbols in the input expression to determine if a rule applies. Since there are no ground facts here, we let d be zero. The non-linear recursive rule `s3` is decomposed into two Imp subgoals, we will say its degree is 2. Let m be the degree of rule with the highest non-linearity in the rule set. In a proof of goals of the form $Imp(\neg(X1 \vee X2), Y)$ where the $X1$ and $X2$ themselves are $\neg(... \vee ...)$ expressions or wires, there are two classes of nodes in the proof tree: the terminal and the interior nodes. Each terminal node is solvable by `s1` and/or `s2` and costs $3t$. The interior nodes which are reduced by `s3` each cost $2t$ for the failed matches on `s1` and `s2` and then t for the reduction by `s3` itself. We add a cost of m for extracting the m components in the input to unify with the m subgoals that are generated. The cost is

$$3t * n + (3t + m) * \frac{n-1}{m-1}$$

.

Note that the costs here are linear in the length of the expression to be synthesized. This has to do with the fact that this theory has a low branching factor and that the unification costs to determine if a rule applies is constant!

Option 2: Add BAGGER2 style rules in front

In the best case, we only need to consider the first two disjunctions of a rule like `s5`. Let n be the number of leaf nodes in the proof tree and m the degree of the rule with the highest non-linearity. In the best case, we use the second disjunct $\frac{(n-m)}{(n-1)}$ times. The total cost is

$$t * (m + 1) + m + (t * (m + 2) + 2 * m) * \frac{n-m}{m-1}$$

The worst case for BAGGER2 style rules occurs when the internal nodes in the proof tree are instances

of the last disjunct in a rule like **s5** and the leaf nodes are handled by the first disjunct. The total cost for this scenario is

$$\frac{n-m}{(m-1)*m}((t+m)*(2^m+m+1)+t*(2^m-1))$$

From this expression we can see that we have exponential growth in m as opposed to the cost in the domain theory which is linear in m.

Option 3: Add tree-indexed BAGGER2 style rules in front

As in the previous option, we will do best case and worst case analysis. The best case for R1 is also the best case for BAGGER2, the worst case for R1 is the worst case for BAGGER2. The cost for the best case computed just as before but with the tree rule (with 2 disjunctions) is

$$C = \frac{n-m}{m-1} * (2m + 2t + m * t) + t + (t+1) * m$$

In the worst case, we obtain the following cost formula.

$$C = \frac{n}{m}(t + m * (t+1)) + \frac{n-m}{(m-1)*m}(t + m + m * (2t+m))$$

Option 4: Keep rules generated by R1

The analysis is similar to that done for the first option. The total cost is

$$t * n + (2t + m) * \frac{n-1}{m-1}$$

Note that Option 4 is the best for this class of structural recursive theories where the unification cost to determine the domain rule to apply is bounded. In fact, Mooney ensures that this is the case, by never doing theorem proving to establish the antecedents of a learned rule, and Tambe and Rosenbloom ensure this by only learning macro rules whose antecedents have a bounded match cost. We have only computed the cost of solving the $\neg(\ldots \vee \ldots)$ type of synthesis goals because that is what these macros were designed to solve. For expressions that contain operations other than $\neg$ and $\vee$, the cost of determining that the macro rule fails to apply will be exponential in the depth at which operations other than the ones above appear in the input expression. The total cost of solving such goals will be the failure cost (worst case costs computed above) plus the regular cost of solving it in the original domain theory.

Experimental Verification of results

We experimentally verify the performance characteristics of strategies 1 through 4 given below for a variety of problem distributions.

1. Use domain theory to solve goals.
2. Use domain theory augmented by rule generated by Bagger2 to solve goals.
3. Use domain theory augmented by tree indexed Bagger2 rules.
4. Use domain theory augmented by rules generated by R1.

We test these specific hypotheses.

Hypothesis 1 *Only when we have a priori knowledge about problem distribution is it effective to learn macro rules a la BAGGER2 or R1. As the percentage of the problems that can solved by the added macro rule is decreased, the overall performance will decrease till we get to a point when the original ordered domain theory does better than the augmented theory with the macro rules. That is, Strategy 1 outdoes Strategies 2 and 3 as the problem distribution is skewed away from the original training instance.*

The role of the experiments is to find the exact cutoff points for the domination of Strategy 2 and 3 by Strategy 1 as a function of the problem distribution.

Hypothesis 2 *As we increase the degree of nonlinearity of the recursive rules, there is exponential degradation in performance upon addition of macro rules a la BAGGER2. I.e., Strategy 2 is dominated by all others as structural recursive theories get more complex.*

Hypothesis 3 *In domains where the recursion is structural and the matching cost for choice of rule to apply is bound, it is better to learn the unwinding of the non-recursive rules as in Strategy 4 instead of self-unwindings of the recursive rules as in Strategies 2 and 3.*

In order to check our hypotheses we built (using C-prolog on a Sun-4) a program which given the arity of operations in the boolean algebra and the rules for synthesizing binary expressions generates the domain theories for strategies 2,3 and 4. A goal to be solved by these strategies is randomly generated by a program that accepts a depth limit (this limits the depth of the proof of the instance) and the allowed boolean operations in the expression. Problem or goal distributions are produced by first picking a percentage of problems that can be solved by the macro rule alone. If that percentage is 80, say, and the problem set has 10 problems, we call the individual goal generator 8 times with a depth limit and with the operators $\neg, \vee$ and then 2 times with the same depth limit and with the whole range of boolean operations that the domain theory can handle. We solve the problem set using each one of the 4 strategies and estimate the cost using the cost model developed in the previous section. To do the cost counting, we use a meta-interpreter also written in C-Prolog. In all these cases, we experimented with various orderings of the rules in the theory. The results

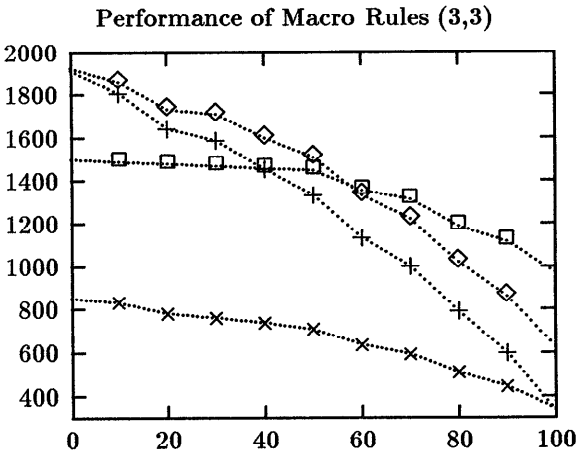

Fig-2

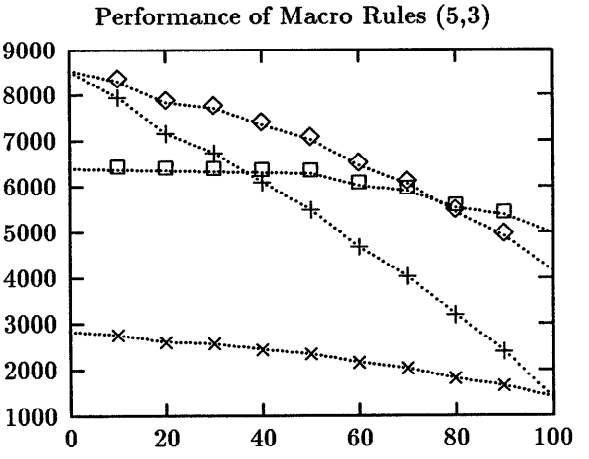

Fig-3

[t]

reported for each method are for the best orderings for that method.

Interpretation of Results

All our experimental hypotheses are verified in our domain. Figure 2 shows that unless the percentage of problems in the future queries that are solvable by the macro rules alone exceeds 60 percent for input expressions of arity 3 (e.g. $(x \vee y \vee z)$ and 80 percent for arity 5, BAGGER2 style macro rules are not useful. The key for interpreting the symbols in the figures is given below.

□	domain theory
◇	BAGGER2
+	Tree-Rule
×	R1

The utility cutoff point for tree indexed BAGGER2 rules is better, but the overall trend is similar to the BAGGER2 rules. These two styles of compilation suffer because they unwind recursion into cases: they save on unifications when they are the right rule to use, however for problems for which they are not the right rule – the instantiated recursive rules increase cost of problem solving. Thus, in the absence of strong guarantees about the repetition of problems like the initial training instance, macro rules that unwind recursion should not be learned.

In Figure 3 we note the degradation of performance of BAGGER2 type rules in synthesizing expressions of arity 5: learning BAGGER2 style rules in structural recursive domain theories with bounded rule selection cost is not useful. This is consistent with our observation before that BAGGER2's performance is exponential in m. In both these figures, the best performance is observed for Strategy 4 where a subset of the non-recursive portion of the original domain theory has been compressed to a single rule, and where the recursive rules are not unwound at all. This observation was also arrived at independently by [Etz90]. Macro rules that decompose cases of a recursive rule to a finer grain than what is available in the domain theory to begin with are more expensive than the original domain theory.

We wanted to check how these strategies react to exponential growth in the the problem size: i.e. we ask the theories to synthesize boolean expressions containing 2^i wires ($2 \leq i \leq 8$). The problem set solved by each strategy was such that 50 percent of the problems could be solved by BAGGER2 style macro rules and the rest were solvable by the other rules in the domain theory. As expected, we get exponential behaviour from all strategies because n is exponential in all cases. However, Strategy 4 has the slowest rising exponent. The domain theory has a higher exponent than Strategy 4 because it does not have the benefit of the unwinding of `s2` by `s1`. This compression which can be generated by ordinary EBL methods is the source of the performance improvement see in Figure 5.

Another interesting experiment is to see how the strategies react to inputs that cause extensive backtracking. We observe that the Strategy 2 has the worst behaviour. And that strategies 1 and 4 do much better than the others. The reason for this is the heavy backtracking cost that is incurred for inputs where the inapplicability of the macro rule is discovered several levels into the recursion in the internally disjunctive rule. All other disjuncts in that rule are tried before the other regular domain theory rules are tried. Strategy 3 suffers from a similar drawback, but because fewer cases are unwound, the exponent is smaller.

Conclusions

The overall message is that for structural recursive domain theories where we can find if a rule potentially

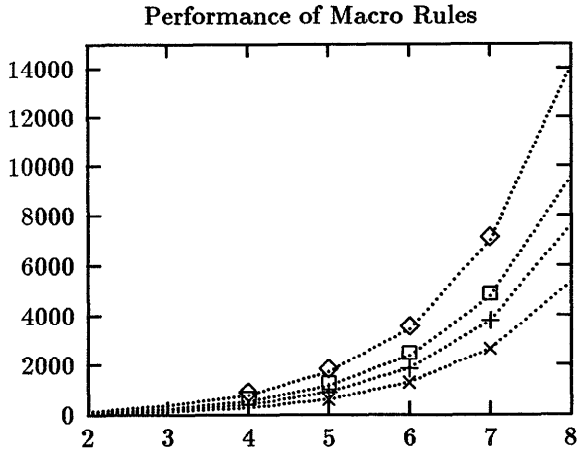

Fig-4

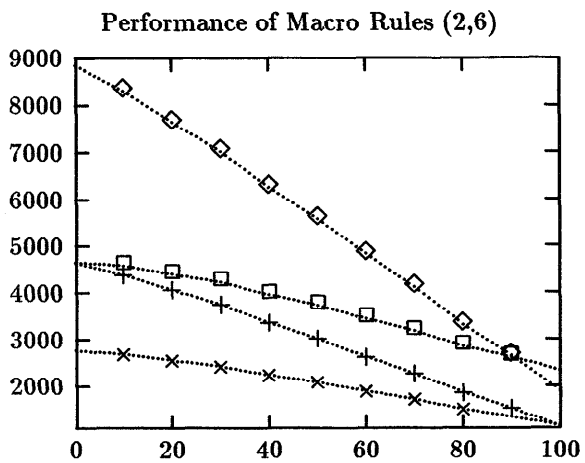

Fig-5

applies by a small amount of computation, forming self-unwindings of recursive rules is wasteful. The best strategy appears to be compressing the base case reasoning and leaving the recursive rules alone. We proved this using a simple cost model and validated this by a series of experiments. We also provided the algorithm R1 for extracting the base case compressions in such a theory.

Acknowledgments

Discussions with Stuart Russell, K. Sivaramakrishnan, Alberto Segre, and John Woodfill were valuable. Our thanks to Jane Hsu for providing careful feedback on an earlier draft.

References

W. Cohen. Generalizing number and learning from multiple examples in explanation-based learning. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 256–269. Morgan Kaufmann, 1988.

O. Etzioni. Why prodigy/ebl works. Technical report, Computer Science Department, Carnegie-Mellon University, January 1990.

R. Greiner. Finding the optimal derivation strategy in a redundant knowledge base. In *Proceedings of the Sixth International Workshop on Machine Learning.* Morgan Kaufmann, 1989.

S. Minton. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 564–569. Morgan Kaufmann, 1988.

R. Mooney. The effect of rule use on the utility of explanation-based learning. In *Proceedings of the Eleventh International Conference on Artificial Intelligence.* Morgan Kaufmann, 1989.

P. Shell and J. Carbonell. Towards a general framework for composing disjunctive and iterative macro-operators. In *Proceedings of the Eleventh International Conference on Artificial Intelligence*, pages 596–602. Morgan Kaufmann, 1989.

D. Subramanian and R. Feldman. The utility of ebl in recursive domain theories (extended version). Technical report, Computer Science Department, Cornell University, 1990.

J. Shavlik. Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 1990.

P. Tadepalli. Learning approximate plans in games. Thesis proposal, Computer Science Department, Rutgers University, 1986.

R. Keller T. Mitchell and S. Kedar-Cabelli. Explanation-based learning: A unified view. *Machine Learning*, 1(1):47–80, 1986.

M. Tambe and A. Newell. Some chunks are expensive. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 451–458. Morgan Kaufmann, 1988.

D.H.D. Warren. An improved prolog implementation which optimises tail recursion. Technical report, Dept. of Artificial Intelligence, University of Edinburgh, 1980.