# Acquiring Recursive Concepts with Explanation-Based Learning

Jude W. Shavlik
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

## Abstract

Explanation-based generalization algorithms need to generalize the *structure* of their explanations. This is necessary in order to acquire concepts where a recursive or iterative process is implicitly represented in the explanation by a *fixed* number of applications. The fully-implemented BAGGER2 system generalizes explanation structures and produces recursive concepts when warranted. Otherwise the same result as standard explanation-based generalization algorithms is produced. BAGGER2'S generalization algorithm is presented and empirical results that demonstrate the value of acquiring recursive concepts are reported. These experimental results indicate that generalizing explanation structures helps avoid the recently reported negative effects of learning. The advantages of the new approach over previous approaches that generalize explanation structures are described.

## 1. Introduction

Explanation-based learning (EBL) systems acquire new concepts by generalizing explanations to specific solutions. It has been recognized that explanation structures that suffice for understanding a specific solution are not always satisfactory for generalizing the solution. Instead, the explanation *structure* must often be *augmented* if a useful generalization is to be produced lShavlik88]. This paper addresses the important issue in EBL of *generalizing to N* [Cheng86, Cohen88, Prieditis86, Shavlik85, Shavlik87]. This can involve generalizing such things as the number of entities involved in a concept or the number of times some action is performed. This type of generalization is necessary in order to acquire concepts where a general iterative or recursive process is implicitly represented by a fixed number of applications in the specific problem's explanation.

BAGGER2 is a fully-implemented system designed to generalize the structure of explanations. This system is the successor to an earlier structure-generalizing EBL system [Shavlik87] that learned iterative concepts (manifested as linear chains of rule applications). Unlike its predecessor, BAGGER2 is capable of acquiring recursive concepts involving arbitrary tree-like applications of rules, can perform multiple generalizations to *N* in one example, and can integrate the results of multiple examples.

The next section presents the BAGGER2 algorithm. Subsequent sections illustrate the new algorithm with a simple example, compare what it learns to the result of more standard EBL systems, present empirical evidence of the value of generalizing explanation structures, and compare BAGGER2 to other systems that generalize number.

This research was partially supported by a grant from the University of Wisconsin Graduate School.

## 2. The BAGGER2 Algorithm

BAGGER2 extends the EGGS algorithm [Mooney86J, a standard EBL algorithm. Both algorithms assume that, in the course of solving a problem, a collection of pieces of general knowledge (e.g., inference rules, rewrite rules, or plan schemata) are interconnected, using unification to insure compatibility. In EGGS, the resulting explanation structure is generalized by first stripping away the details of the specific problem and then determining the most general unifier that allows the general pieces of knowledge to be connected in the same way. This involves replacing the constants in the specific explanation with constrained variables. The result is a new composite knowledge structure that contains the unifications that must hold in order for the knowledge pieces to be combined in the given way. Assuming tree-structured explanations, if the leaf nodes can be satisfied, the root (goal) node will also be satisfied. There is no need to again reason about combining the pieces of knowledge together to achieve the goal. Since a substantial amount of work can be expended constructing the original solution, the new knowledge structure can lead more rapidly to a solution. Notice, though, that the structure of the explanation is not changed. If some process is repeated three times in the specific problem's explanation, it will be repeated exactly three times in the concept acquired by EGGS.

BAGGER2 generalizes explanation structures by looking for repeated intcr-dependent sub-structures in an explanation. Figure 1 schematically presents this process. Assume that in explaining how a goal is achieved, the same general sub-problem (P) arises several times. The full explanation can be grouped into several qualitatively different portions. First, there arc the sub-explanations where an instantiation of P is supported by the explanations of other instantiations of the general problem P. In the figure, these are the sub-explanations marked 1 and 4. Second, there are the sub-explanations where an

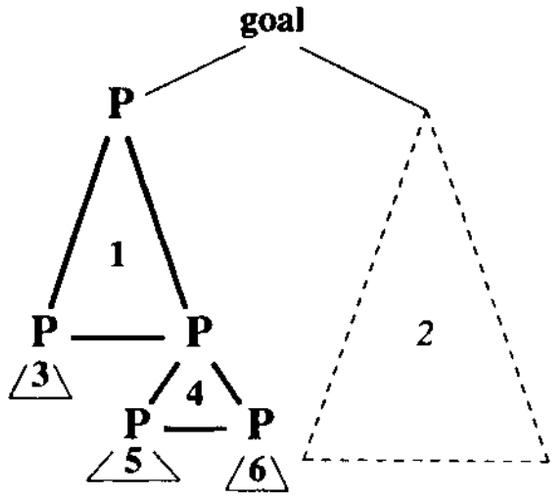


**Figure 1. Generalizing the Structure of an Explanation**

instantiation of P is explained without reference to another instantiation. These arc the sub-explanations labelled 3, 5, and 6. Finally, there are the portions not involving P (sub-explanation 2).

The explanation in figure 1 can be viewed as the trace of a recursive process. This is exactly what must be recognized in the explanation of a specific example if a recursive or iterative concept is to be learned. The generalizations of sub-explanations 1 and 4 form the recursive portion of the concept, while the generalizations of sub-explanations 3, 5, and 6 produce the termination conditions. BAGGER2 partitions explanations into groups as illustrated by figure 1, from which a new recursive concept is produced.

The BAGGER2 generalization algorithm appears in figure 2. This algorithm is expressed in a pseudo-code, while the actual implementation is written in Common Lisp. The remainder of this section elaborates the pseudo-code. In the algorithm back arrows (<—) indicate value assignment. The construct

for each *element* in *set* do *statement*

means that *element* is successively bound to each member *of set,* following which the *statement* is evaluated.

The BAGGER2 algorithm assumes explanations arc derivation *trees* (e.g., something that could be produced by a Horn clause theorem prover such as Prolog). As is standard in explanation-based algorithms, an *explanation structure* is first produced from the specific problem's explanation. To build the explanation structure, each instantiated rule in the explanation is replaced by a copy of the original general rule. (If the same general rule is used multiple times, each time it appears in the explanation structure its variables are renamed. This prevents spurious equalities among variables in the explanation structure.)

The algorithm starts at the root of the explanation structure. If something that unifies with the general goal appears elsewhere in the explanation structure, then a recursive rule (called a *recurrence)* is produced starting at the root node. Otherwise, the general version of the antecedents arc collected and a new rule produced. A recurrence can also arise within an explanation structure, and this discussion will assume the root node does not directly lead to a recurrence.

*Collect GeneralAntecedents* produces the necessary requirements for the consequent of a rule to hold. Ignoring for a moment the possibility of recurrences being constructed, this entails traversing through the explanation structure and stopping at *operational* [Keller88] nodes. Along the way all the unifications necessary to connect the rules in the explanation structure are collected (thus eliminating the need to check these when the acquired rule is later applied). Operational nodes are either antecedents satisfied by a problem-specific fact or antecedents somehow judged to be easily satisfied. This portion of the algorithm is merely a rehash of the EGGS algorithm. Hence, notice that when BAGGER2 detects no potential generalizations to TV it produces the same result as the EGGS algorithm.

More interesting is what happens when a potential recurrence is detected. This is done by seeing if, in the derivation of a general antecedent, a unifiable version of the antecedent appears (e.g., the P's in figure 1). If so, the explanation structure headed by the general antecedent is partitioned into two types of sub-explanations. Those *terminal proofs* where a version of the antecedent docs not appear in its proof and those *recursive proofs* where at least one does. In the recursive proofs, the recursive sub-explanations arc replaced by a call to the recurrence being constructed. These calls contain the term that must be unified with the consequent of the recurrence. Hence, in figure 1, when cutting out sub-explanation 1, sub-explanations 3 and 4 are removed. Notice, then, that the cut-out sub-explanations are non-overlapping.

Once the sub-explanations are produced, each is generalized by again calling the BAGGER2 algorithm. This means that another recurrence can be found within a sub-explanation, allowing multiple generalizations to *N* in a single example. When generalizing the sub-explanations, the necessary unifications between the root of the sub-explanation and the recurrence are collected. The generalizations of the sub-explanations are disjunctively combined and a recurrence produced. Since two sub-explanations may generalize to the same result, duplicate disjuncts arc removed from the acquired recurrence.

The recurrence is a separate entity from the rule produced for the full explanation. Because of this, they support transfer of the the results learned during one task to the performance of another, provided the two tasks involve common sub-tasks. Recurrences being separate entities also supports learning from multiple examples. If a new method for satisfying the consequent of a recurrence is encountered, it can be merged with the previous disjuncts.[1]

[1] This may lead to poor performance if too many disjuncts are learned. The user of BAGGER2 can decide when a concept is sufficiently learned and tell the system to "freeze" all of its recurrences. After that, new recurrences will be built even if they have the same consequent as an existing one.

```
procedure BuildNewBAGGER2rule (goalNode)
    "Generalize the explanation headed by this node."
    consequent ← Consequent(goalNode)
    if PotentialRecurrence?(consequent) then ProduceRecurrence(goalNode)
    else AddRule(consequent, CollectGeneralAntecedents(goalNode))

procedure CollectGeneralAntecedents (node)
    "Collect the generalized version of the antecedents of this explanation node."
    result ← ϕ
    for each antecedent in Antecedents(node) do
        if OperationalNode?(antecedent) then AddConjunct(result, antecedent)
        else if RecurenceCall?(antecedent) then AddConjunct(result, antecedent)
        else if PotentialRecurrence?(antecedent,node) then
            AddConjunct(result, ProduceRecurrence(antecedent))
        else if SupportedByRuleConsequent?(antecedent) then
            AddConjunct(result, CollectNecessaryEqualities(antecedent, SupporterOf(antecedent)))
            AddConjunct(result, CollectGeneralAntecedents(SupportingRule(antecedent)))
        else return fail
    return SimplifyConjunct(result)

procedure ProduceRecurrence (node)
    "Produce a BAGGER2 recurrence from the sub-explanation headed by this node."
    result ← ϕ
    for each proof in CutOutSingleApplicationProofs(node) do      ; terminal proofs
      AddDisjunct(result,CollectNecessaryEqualities(node,Consequent(proof)))
      AddDisjunct(result,CollectGeneralAntecedents(proof))
    for each proof in CutOutRepeatedApplicationProofs(node) do    ; recursive proofs
      AddDisjunct(result,CollectNecessaryEqualities(node,Consequent(proof)))
      AddDisjunct(result,CollectGeneralAntecedents(proof))
    return AddRecurrence(Consequent(node), SimplifyDisjunct(result))
```

**Figure 2. The BAGGER2 Generalization Algorithm**

**Table 1. Supporting Functions in the BAGGER2 Algorithm**

| Function | Description |
|---|---|
| *PotentialRecurrence?(Antecedent,Node)* | Is this antecedent supported by a term that unifies with node? |
| *OperationalNode?(Antecedent)* | Is this antecedent in the explanation considered operational? |
| *RecurrenceCall(Antecedent)* | Is this a call to a *recurrence* (see text)? |
| *CollectNecessaryEqualities(T1,T2)* | Collect the equalities that must hold such that these two terms unify. |
| *SupportedByRuleConsequent?(Antecedent)* | Is this antecedent supported by the consequent of another rule? |
| *CutOutSingleApplicationProofs(Node)* | Return a list of the *terminal* subproofs of this node (see text). |
| *CutOutRepeatedApplicationProofs(Node)* | Return a list of the *recursive* subproofs of this node (see text). |
| *AddConjunct(X,Y)* | Reset *X* to *X AND Y*. |
| *AddDisjunct(X,Y)* | Reset *X* to *X OR Y*. |
| *AddRule(Consequent, Antecedents)* | Add this rule to the rule database. |
| *AddRecurrence(Consequent, Antecedents)* | Construct a recurrence and return a call to it. |

Before BAGGER2 produces a new rule, it reorganizes the antecedents. This involves removing redundant antecedents and reordering them to increase the efficiency of future retrievals. In recurrences, if an antecedent (one independent of the variables in the recurrence's consequent) appears in every terminal disjunct, it can be removed from the recursive disjuncts.

Assuming that explanations are logical proofs, the BAGGER2 algorithm can be proved correct.

Theorem: The BAGGER2 algorithm is *sound.* That is, the rules it learns will never derive anything that cannot be derived by the initial domain theory (see [Shavlik89] for the proof).

There are several shortcomings of the BAGGER2 algorithm. One, explanations must be trees. Two,

recurrences may not terminate. Three, when a concept involves multiple recurrences, better performance often can be obtained by merging the recurrences together. Four, there can be redundant computation in some cases. Five, a recurrence can acquire too many disjuncts, thereby decreasing its utility. Approaches to these problems are discussed in [Shavlik89].

## 3. An Example

An sample application of the BAGGER2 algorithm appears in this section. Circuit design is the domain used. Rules (appearing in the appendix) determine how to implement a circuit depending on the type of gates available. Assume only *AND* and *NOT* gates are available. DcMorgan's law must be repeatedly applied in order to implement a circuit involving a collection of *OR* gates, in which the final output is negated. An explanation of how this task can be accomplished can be produced using the rules provided.

If the EGGS algorithm is applied to the resulting explanation, the rule in figure 3 results. Notice that this rule not only requires a fixed number of inputs, but also a fixed topology. Clearly the explanation structure needs to be generalized.

```
(Implement-By
  (not (or (or (or (not ?g6) (not ?g9))
                (or (not ?g16) (or (not ?g23) (not ?g26))))
           (or (or (not ?g37) (not ?g40)) (not ?g43))))
  (and (and (and ?g6 ?g9)
            (and ?g16 (and ?g23 ?g26)))
       (and (and ?g37 ?g40) ?g43)))
  ←
(and (have-ands) (wire ?g43) (wire ?g6) (wire ?g9) (wire ?g16)
     (wire ?g37) (wire ?g40) (wire ?g23) (wire ?g26))
```

**Figure 3. The Rule Acquired by EGGS**

```
Recurrence implement-by-1:
  (implement-by (not (or ?v1 ?v2)) (and ?v3 ?v4))
    ←
  (or (and (match ?v1 (not ?v3))      (match ?v2 (not ?v4))
           (have-ands)                (wire ?v3)
           (wire ?v4))
      (and (match ?v1 (or ?e1 ?e2))   (match ?v2 (not ?v4))
           (match ?v3 (and ?e3 ?e4)) (wire ?v4)
           (call implement-by-1
             (implement-by (not (or ?e1 ?e2)) (and ?e3 ?e4))))
      (and (match ?v1 (not ?v3))      (match ?v2 (or ?e1 ?e2))
           (match ?v4 (and ?e3 ?e4)) (wire ?v3)
           (call implement-by-1
             (implement-by (not (or ?e1 ?e2)) (and ?e3 ?e4))))
      (and (match ?v1 (or ?e1 ?e2))   (match ?v2 (or ?e3 ?e4))
           (match ?v3 (and ?e5 ?e6)) (match ?v4 (and ?e7 ?e8))
           (call implement-by-1
             (implement-by (not (or ?e1 ?e2)) (and ?e5 ?e6)))
           (call implement-by-1
             (implement-by (not (or ?e3 ?e4)) (and ?e7 ?e8))))))
```

**Figure 4. The Recursive Rule Acquired by BAGGER2**

The result produced by BAGGER2 appears in figure 4. In this problem, the full explanation leads to a single recurrence. The recurrence (which is given a "gensym'ed" name) involves four disjuncts. The first applies when only a single application of DeMorgan's rule is necessary. *AND* gates must be available if the resulting circuit is to be implemented. The remaining three disjuncts are recursive. The second and third disjuncts apply when one input is a wire. In this case, the rule recurs on the other input. In the final disjunct, recursion is needed for both inputs. (If the training example was simpler, all of these conditions may not have been encountered and multiple examples would be needed to learn the complete concept.)

A couple of points about the notation in figure 4 are necessary. The *match* predicate unifies its two arguments. The italicized *or's* and *and's* describe the meaning of the rule, while the others refer to gates in the circuit being designed. The special predicate *call* calls the recurrence named in its first argument (recall that there can be recurrence calls within recurrence calls, which is why the name is needed). The second argument is unified with the consequent of the recurrence upon the recursive call. Finally, BAGGER2 renames the variables in recurrences. Variables starting with v appear in the consequent, while the *e* variables arc "local" variables.

The rule learned by BAGGER2 can be viewed as a general version of DcMorgan's law. It converts the negation of an N-input *OR* gate into an N-input *AND* gate. Notice that it applies to a much larger class of problems than does the rule learned by EGGS.

Notice that the acquired recurrence does not refer to any of the initial rules. It is self-contained and is topologically similar to a recursive Lisp function. The consequent specifics the parameters and the antecedents form something like a Lisp *COND.* This "function" is produced from a collection of simple declarative Prolog-like rules. Rules are called explicitly rather than seeing which rules in a large rulebase may be applicable. Hence, BAGGER2 provides a way to transform a simple, but inefficient, logic program into a program in a more efficient language.

## 4, Empirical Analysis

A question arises. Is it worthwhile to generalize explanation structures? Generalizing explanation structures leads to acquiring more general rules, but because the resulting rules are more complicated, applying them entails more work. This question involves the relationship between the *operaiionality* and *generality* of acquired rules [Keller88]. Experiments reported in this section investigate whether it is better to the learn a more general recursive rule or whether it is better to individually learn the subsumed rules as they are needed.

Using the circuit design rules, three systems are compared: BAGGER2, EGGS, and no-learn (a system that does not learn any new rules). The two learning systems arc given some number of circuits to convert and if they use more than one rule to solve a problem, they generalize the resulting explanation and save the new rule. Following this training phase, all three systems try to solve a new collection of ten problems, with the learning systems giving

priority to their acquired rules. During this testing phase no learning occurs. All three systems use the same backward-chaining problem solver, which is basically a Lisp implementation of Prolog augmented to handle explicit calls to BAGGER2's recurrences. Each experiment is repeated ten times. Hence, each point plotted in the figures below is the mean of 100 measurements.

In the first experiment, training problems consistent of randomly generated implementations of 8-input *OR* gates using binary *OR}* The final output is negated. As in the last section, the task is to implement this gate using only *NOT* and binary *AND* gates.

The percentage of test problems solvable using each systems acquired rules is plotted in figure 5. Clearly, BAGGER2 needs many fewer training examples to learn the concept being taught.

The next issue, a more important one, is how long it takes each of the systems to solve a new problem. The learning systems are trained on *OR* circuit problems of various sizes. The number of randomly-generated training examples for each problem type equals the possible number of binary circuits with that total number of inputs (see the formula in the previous footnote), EGGS organizes its rules according to the number of inputs involved (i.e., in six groups) and only possibly relevant rules are checked during problem solving. Figure 6 contains the mean solution time on the test problems. (For EGGS, only the time spent on problems solved by a learned rule is recorded. Both this and the assumption about rule organization favor EGGS.)

Figure 6 shows that merely learning all possible cases is worthwhile if there are only a few possible cases. However, as the number of possible cases grows, it soon becomes worthwhile to learn recursive rules. Note that after awhile, it would be better to have not learned at all than to use EGGS. A structure-generalizing EBL algorithm such as BAGGER2 helps avoid the negative effects of learning recently reported [Minton88].

BAGGER2 has also been run on blocks-world problems (see [Shavlik89] for details). The task is to teach a system how to build towers of a range of heights. Figure 7 presents, on a logarithmic scale, the performance of the three systems on this task as the maximum tower height increases. (At each point, enough training examples are presented so that both learning systems completely learn the concept.) Again, as the complexity of problems increases, BAGGER2 begins to out-perform EGGS.

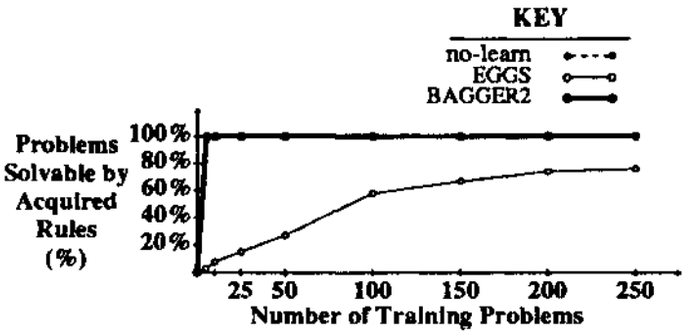


**Figure 5. New Problems Solvable with Acquired Rules as a Function of Amount of Training**

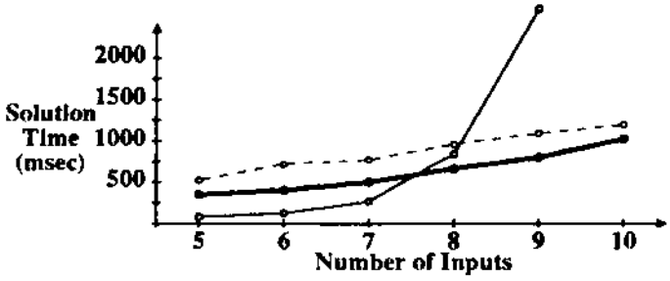


**Figure 6. Mean Solution Time as a Function of Problem Size**

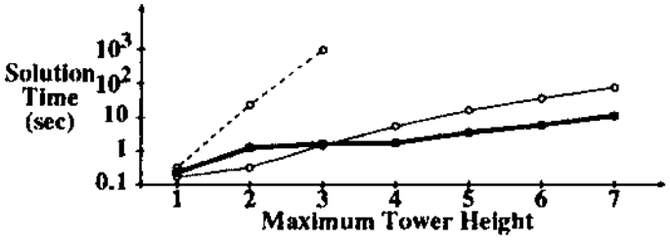


**Figure 7. Mean Solution Time as a Function of Problem Size for Tower Building**

## 5. Related Work

Besides BAGGER [Shavlik87] (which only learns iterative concepts) and BAGGER2, several other explanation-based approaches to generalizing number have been recently proposed.

Prieditis [Prieditis86] developed a system that learns macro-operators representing linear sequences of repeated STRIPS-like operators. Recursive rules are not learned, nor are disjunctive ones. In the FERMI system [Cheng86], cyclic patterns are recognized using empirical methods and the detected repeated pattern is generalized using explanation-based learning techniques. However, unlike the techniques implemented in BAGGER2, the rules acquired by FERMI are not fully based on an explanation-based analysis of an example, and so are not guaranteed to always work. Cohen [Cohen88] recently developed and formalized another approach to the problem of generalizing number. His system generalizes number by constructing a finite-state control mechanism that deterministically directs the construction of proofs similar to the one used to justify the specific example. His approach can acquire recursive and disjunctive concepts, as well as learn from multiple examples. However, his approach assumes that no new relevant facts or rules are added to the database after learning. This means that, unlike BAGGER2, a new concept cannot be learned in the presence of one set of facts and then applied under a new set of facts (e.g, from one blocks-

[2] The number of ways to implement an $N$-input *OR* gate with binary gates is $\frac{(2N-2)!}{N!(N-1)!}$ (page 18 of [Jacobson51]).

world scene to another). Finally, in Physics 101 [Shavlik88] the need for generalizing number is motivated by analyzing mathematical calculations.

The problem of generalizing to *N* has also been addressed within the paradigms of empirical (or similarity-based) learning (e.g., [Sammut86]) and automatic programming (e.g., [Summcrs77]). A general specification of number generalization has been advanced by Michalski [Michalski83]. He proposes a set of generalization rules including a *closing interval rule* and several *counting arguments rules* which can generate number-generalized structures. The difference between such empirical approaches and BAGGER2's explanation-based approach is that the newly formed similarity-based concepts typically require verification from corroborating examples, whereas the explanation-based concepts are immediately supported by the domain theory.

## 6. Conclusion

Explanation-based learning systems must generalize explanation structures if they are to be able to fully extract general concepts inherent in the solutions to specific examples. A general approach for doing so has been presented. The BAGGER2 algorithm is capable of learning complicated recursive concepts, can integrate results from multiple examples, and has been shown to perform better than a standard EBL algorithm (EGGS). Experimental results indicate that generalizing explanation structures helps avoid the recently reported negative effects of learning [Minton88]. On problems where learning a recursive rule is not appropriate, the system produces the same result as the EGGS algorithm. Applying the recursive rules learned only requires a minor extension to a Prolog-like system, namely, the ability to explicitly call a specific rule. This research brings EBL closer to its goal of being able to acquire the full concept inherent in the solution to a specific problem.

## Appendix - Initial Rules for the Circuit Problem

```
implement-by(not(not(?x))),?y)  :-
   implcment-by(?x,?y).

implement-by(not(and(?x,?y)),nand(?a,?b))  :-
   have-nands, implcmcnt-by(?x,?a), implement-by(?y,?b).

implement-by(not(?x),nand(?y, 1))  :-
   havc-nands, implement-by(?x,?y).

implement-by(and(?x,?y),nand(nand(?a,?b), 1))  :-
   havc-nands, implement-by(?x,?a), implement-by(?y,?b).

implement-by(not(?x),not(?y))  :-
   have-nots, implement-by(?x,?y).

implement-by(or(?x,?y ),or(?a,?b)) : -
   havc-ors, implement-by(?x,?a), implcment-by(?y,?b).

implement-by(or(?x,?y),nand(?a,?b))  :-
   have-nots, havc-nands,
   implement-by(not(?x),?a), implement-by(not(?y),?b).

implcment-by(not(or(?x,?y)),and(?a,?b))  :-
   have-ands,
   implemcnt-by(not(?x),?a), implcmcnt-by(not(?y),?b).

implement-by(?wire,?wirc)  :-  wire(?wire).
```

## References


lCheng86] P. Cheng and J. G. Carbonell, "The FERMI System: Inducing Iterative Macro-operators from Experience," *Proceedings of the National Conference on Artificial Intelligence,* Philadelphia, PA, August 1986, pp. 490-495.

[Cohen88] W. W. Cohen, "Generalizing Number and Learning from Multiple Examples in Explanation-Based Learning," *Proceedings of the Fifth International Conference on Machine Learning,* Ann Arbor, MI, June 1988, pp. 256-269.

[Jacobson51] N. Jacobson, *Lectures in Abstract Algebra, Vol. 1,* Von Nostrand, Princeton, NJ, 1951.

[Kcller88] R. M. Keller, "Defining Operationality for Explanation-Based Learning," *Artificial Intelligence 35, 2* (1988), pp. 227-241.

[Michalski83] R. S. Michalski, "A Theory and Methodology of Inductive Learning," *Artificial Intelligence 20, 2* (1983), pp. 111-161.

[Minton88] S. Minton, "Quantitative Results Concerning the Utility of Explanation-Based Learning," *Proceedings of the National Conference on Artificial Intelligence,* St. Paul, MN, August 1988, pp. 564-569.

[Mooney86] R. J. Mooney and S. W. Bennett, "A Domain Independent Explanation-Based Generalizer," *Proceedings of the National Conference on Artificial Intelligence,* Philadelphia, PA, August 1986, pp. 551-555.

[Prieditis86] A. E. Prieditis, "Discovery of Algorithms from Weak Methods," *Proceedings of the International Meeting on Advances in Learning,* Les Arcs, Switzerland, 1986, pp. 37-52.

lSammut86] C. Sammut and R. B. Banerji, "Learning Concepts by Asking Questions.," in *Machine Learning: An Artificial Intelligence Approach, Vol. II,* R. S. Michalski, J. G. Carbonell and T. M. Mitchell (ed.), Morgan-Kaufmann, Los Altos, CA, 1986, pp. 167-192.

[Shavlik851 J. W. Shavlik and G. F. DeJong, "Building a Computer Model of Learning Classical Mechanics," *Proceedings of the Seventh Annual Conference of the Cognitive Science Society,* Irvine, CA, August 1985, pp. 351-355.

[Shavlik87] J. W. Shavlik and G. F. DeJong, "BAGGER: An EBL System that Extends and Generalizes Explanations," *Proceedings of the National Conference on Artificial Intelligence,* Seattle, WA, July 1987, pp. 516-520.

[Shavlik88] J. W. Shavlik, "Generalizing the Structure of Explanations in Explanation-Based Learning," Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, IL, January 1988. (Also appears as UILU-ENG-87-2276, AI Research Group, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.)

[Shavlik89] J. W. Shavlik, "Acquiring Recursive and Iterative Concepts with Explanation-Based Learning," Technical Report, Department of Computer Science, University of Wisconsin, Madison, WI, 1989.

[Summers77] P. D. Summers, "A Methodology for LISP Program Construction from Examples," *Journal of the Association for Computing Machinery 24,* (1977), pp. 161-175.