Constraints on Predicate Invention

Ruediger Wirth
wirth@ics.uci.edu
Paul O'Rorke
ororke@ics.uci.edu

Technical Report 91–23

February 28, 1991

Research supported in part by National Science Foundation Grant Number IRI-8813048, Douglas Aircraft Company, and the University of California Microelectronics Innovation and Computer Research Opportunities Program.

We thank Mike Pazzani, Dennis Kibler and the graduate students of the AI & ML community at UCI for discussions of the ideas expressed in the paper and for discussions on related systems such as FOIL and FOCL. Thanks also to Yousri El Fattah, who participated in the initial stages of this research.

Constraints on Predicate Invention[1]

Ruediger Wirth (wirth@ics.uci.edu)
Paul O'Rorke (ororke@ics.uci.edu)

Department of Information and Computer Science
University of California, Irvine, CA 92717
United States of America
Phone: (714) 856-8323
Fax: (714) 856-4056

TOPICS: LEARNING RELATIONS & CONSTRUCTIVE INDUCTION

DRAFT

Febuary 28, 1991

Abstract

The main contribution of this paper is a two step method for inventing new predicates which overcomes some of the shortcomings of previously published methods (implemented in Cigol & LFP2). The method integrates abductive and inductive learning. In the first step, proofs of the training instances are completed by assuming new facts built from a new predicate symbol. In the second step, the general clause derived in order to explain the training instances is used to generate more instances of the newly invented predicate. These instances are then used to induce a general definition of the new predicate.

[1]Research supported in part by National Science Foundation Grant Number IRI-8813048, Douglas Aircraft Company, and the University of California Microelectronics Innovation and Computer Research Opportunities Program.

Contents

List of Figures

1 Introduction

In recent years there has been increasing interest in systems that induce first order logic programs. The approach of inverting resolution (Muggleton & Buntine, 1988; Rouveirol & Puget, 1989; Wirth, 1989) is particularly interesting because it offers a way to extend the vocabulary by inventing new predicates. However, the first implementations were too inefficient to be useful for larger applications.

Quinlan's FOIL (Quinlan, 1990) was an advance towards more efficient induction algorithms for first order languages. Subsequently, Muggleton & Feng (Muggleton & Feng, 1990) presented a new system, called GOLEM, which is based on inverse resolution and which is also able to process large numbers of examples. But, despite their efficiency these two systems are highly dependent on the vocabulary and the form of examples that are given in advance. They cannot extend their vocabulary.

This paper describes an attempt to overcome this limitation. We propose a new way to construct a first-order theory which allows for natural incorporation of background knowledge and the invention of new predicates. The method, implemented in a system called SIERES, is based on a general-to-specific search guided by constraints on the form of clauses.

Unlike FOIL, which searches in a very unconstrained space, SIERES iteratively increases the space by looking at increasingly complex clauses. If it cannot construct a clause that covers the training instances in the current restricted space using known predicates only, SIERES tries to invent a new predicate. There are some strict and heuristic conditions on the new predicate. If the predicate can be constructed, SIERES continues to learn a general definition for it, abductively deriving new instances.

Existing methods for inventing new predicates, for instance in the framework or inverse resolution like CIGOL (Muggleton & Buntine, 1988) or LFP2 (Wirth,1989ab) invent new predicates in order to reformulate a given set of clauses aiming at a more compact representation. Immediately after the new predicate is introduced we cannot prove much more than before. The success set of the program remains the same in the case of Cigol and is only slightly increased in the case of LFP2. It is not until the new predicate is generalized by different operators and reused in different clauses that the success set is actually increased. However, it is often the case that the operators which would generalize the new predicate cannot be applied because they do not lead to a more compact representation. As a result, LFP2 frequently reinvents the same predicate in slightly different settings. It relies on the user to make the connection between the predicates and achieves the generalization of the predicate definition this way.

One of the reasons for this behavior is that the compaction is used for two purposes, compressing the theory and generalizing it. While there is a close connection between data compression and generalization, we claim that in the case of predicate invention it is beneficial to keep them apart.

There are a lot of cases where a new predicate is needed to actually *increase* the success set at the time it is introduced. A new predicate is needed in order to formulate a theory which is more general than the examples. For instance, if we have no background knowledge and we want to learn a general definition for `reverse`, we cannot do this with only the predicate `reverse`. We *need* a new predicate. A general version of DeMorgan's law provides a similar example described below.

The purpose for the introduction of the new predicate in the method described in this paper is fundamentally different from systems like CIGOL and LFP2. New predicates are invented because they make the proof succeed not because they provide a more compact representation. From an abductive point of view, postulating the new relationship expressed by the new predicate helps explain the observations.

2 Integrating Abduction and Induction

This paper describes a novel learning method that integrates abduction and induction. The basic idea is to view proofs as explanations and resolution as an abduction process constructing explanations of observations in terms of a general theory. Abduction is used to complete explanations and infer specific missing facts. Induction is used to invent clauses and predicates in order to extend the general theory and improve its explanatory power.

2.1 Learning Specific Facts by Synthesis

Existing abductive learning methods learn while using general theories to construct explanations of specific observations (O'Rorke, Morris & Schulenburg, 1990). The learning method is a form of abductive inference. Queries that do not ground out in known facts are treated as more or less plausible hypotheses on the grounds that if they were true they would complete explanations of the observations.

A simple abductive learning method called "synthesis" was proposed by Pople (Pople, 1973). Technically, the method works as follows. Given an observation and a theory expressed as Horn clauses, backward chain in search of a proof justifying the literals of the observation. If two queries are generated that are unifiable, unify them and assume that the resulting literal is true. Since it enables one to explain two observations with the same hypothesis, Pople justified this operation in terms of Occam's Razor. Note that Pople's synthesis operation is non-deductive, so this method of abductive learning is a form of knowledge-level learning. In other words, if a literal is added to the theory by synthesis, it enlarges the deductive closure of the theory.

2.2 Learning General Facts by Anti-Synthesis

Least general generalization *LGG* (Lassez, Maher & Marriot, 1988; Plotkin, 1970; Plotkin, 1971) can be used in an abductive framework to learn interesting new literals that are generalizations rather than specializations of literals that appear in existing rules. Assuming that Q_1 and Q_2 are two queries that arise in explanations of the same or different cases, $Q = LGG(Q_1, Q_2)$ is a hypothesis that would explain Q_1 and Q_2.

This is a dual to Pople's synthesis operator; call it anti-synthesis. In synthesis, the queries Q_1 and Q_2 have to be unifiable. This is not necessary in anti-synthesis. The queries Q_1 and Q_2 could be ground literals involving different constants. In synthesis, the queries unify to a new literal (their most general common instance). The queries both subsume this new literal. In anti-synthesis, the new literal Q is the least general common anti-instance (Lassez, et al., 1988) of Q_1 and Q_2. It subsumes Q_1 and Q_2 but different substitutions might be used to get each instance. Like synthesis, anti-synthesis leads to new literals that improve the coherence of explanations.

2.3 Learning New Clauses

The least generalization of abductive hypotheses serves as the initial candidate in our search for a clause that would enable us to complete an explanation. Assuming that the missing clause is applicable to a set of abductive hypotheses, the head of the clause must be unifiable with each hypothesis, so it *must* be a generalization of these hypotheses. Unfortunately, the *LGG* of the abductive hypotheses is often overly general. During the generalization process important connections between input and output arguments are often lost (Kodratoff & Ganascia, 1986). In this case, we specialize the learned clause by adding literals to its body. This enables us to acquire missing clauses other than unit clauses.

There are different ways to specialize a clause (Kietz & Wrobel, 1991; Quinlan, 1990; Shapiro, 1983) and different ways to constrain the search. In the next section we describe a method using a novel combination of constraints.

3 The Method

In the following description of the learning task and our method, we use the terminology of logic programming (Lloyd, 1987).

3.1 The Task: Learning New Clauses

Given:

- background knowledge P and
- a set of initial goals (training instances) $E = \{E_1, \cdots, E_n\}$ that follow from an unknown target program $P_{target} \supset P$, but not from P

the learning goal is to construct a set of clauses C_{target} such that

$$P' = P \cup C_{target} \vdash_{SLD} E.$$

In other words, we want to extend a given theory to cover new examples.

3.2 Strict Constraints on New Clauses

Let us assume we are in a state with goals $\{G_1, \cdots, G_n\}$[2] where none of the clauses of the current program P is applicable to any of the G_i. We have to generate a new clause in order to complete the proof.

Assuming that there is exactly one clause missing, there are strict constraints on this new clause:

- Its head has to be unifiable with the G_i.
- The clause has to produce the proper bindings for the output variables.

SIERES needs to specialize a unit clause if it is too general. Usually, overgeneralizations are discovered using negative examples. But if the input/output behavior of the target predicate is given, for example in the form of *mode declarations* (Shapiro, 1983), there is a syntactic way to identify some important cases of overgeneralization and to provide guidance to the specialization process.

Example: `reverse/2`. Let us assume we have a mode declaration `reverse(+,-)` specifying that the first argument is an input while the second argument is the output. Now, let us look at the following three unit clauses, which could be the least generalizations of sets of instances.

```
reverse([A],[A]).
reverse([A,B],[B,A]).
reverse([A|B],[C|D]).
```

The first two are correct according to the intended meaning of `reverse/2`. For any input list containing one or two elements these two unit clause generate the correct reversed lists. The third one is overly general. In it, there is no connection between the input and output arguments at all. This predicate would be true for any two lists. If we view

[2]These goals could be either abductive hypotheses as described in the previous section or teacher provided training instances as in the usual inductive learning situation.

this unit clause as a procedure with the mode declaration specified above, the output variables would remain unbound because they do not also appear as input variables. These unbound output variables indicate overgeneralization. The need to bind them in the body of a clause provides guidance to the specialization process.

Definition: *Critical terms* of the head of a clause are

- output variables that do not appear in the input arguments
- input variables that do not appear in the output arguments
- terms whose arguments are critical variables

Example: Given mode declaration `reverse(+,-)`, the critical terms of `reverse([A|B], [C|D])` are the members of the set {[A|B], A, B, [C|D], C, D}.

3.3 Heuristic Constraints on New Clauses

In addition to these relatively strict constraints, we employ heuristic constraints on the types of clauses that are to be learned. These additional constraints serve to prune the search space and provide information necessary for the invention of new predicates.

In meaningful clauses, the literals in the body are usually not independent of each other but share at least some variables. This dependency can be used to partially order the literals in a clause.

Definition: A literal L_2 *depends on* a literal L_1 if

- they share a variable V where
- V is an output variable in L_1 and
- V is an input variable of L_2.

Example: In a form of `reverse/2` defined:

```
reverse([A|B], [C|D]) :- reverse(B, E), add_last(E, A, [C|D]).
```

the dependencies are as shown in figure 1. The tail of the input list in the head is passed to the first literal of the body as an input. The output of this recursive call is passed as an input to the final literal of the body. This literal also takes the first element of the initial input to the head and its output is passed back to the head as the output computed by the clause.
Example: In the form of merge sort defined:

```
msort([A|B], C|D]) :- split([A|B], E, F), msort(E, G), msort(F, H), merge(G, H, [C|D]).
```

the dependencies are as shown in figure 2.

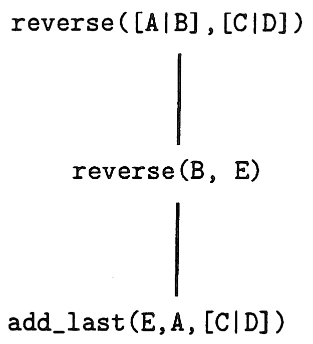

Figure 1: Dependencies in `reverse`

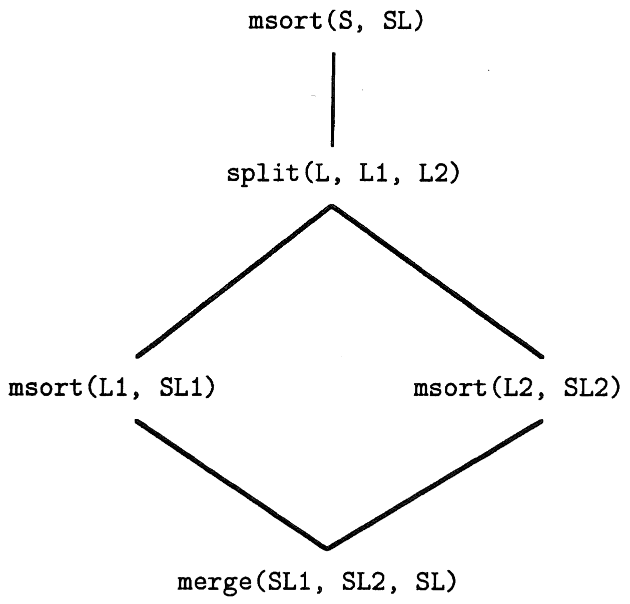

Figure 2: Dependencies in merge sort

3.4 Algorithm

In this section, we describe the method in more detail. As mentioned in the description of the task, the algorithm is given as input background knowledge and training examples. In addition, the algorithm is given mode declarations of all known predicates and a sequence of argument dependency graphs. The output of the algorithm is a set of clauses C such that the examples follow from the new clauses and the background knowledge.

The basic idea of the method (figure 3) is to form the least general generalization of the training instances. If this is too general, a search for more specific clauses is conducted, subject to constraints that prevent the search from getting out of hand (at the cost of missing some concepts). New predicates are introduced as needed.

The current implementation of SIERES constrains search using mode declarations and a limited sequence of argument dependency graphs like the sequence shown in figure 4. The head of the clause can contain only one output argument. Input terms of literals in the body must be subterms of input terms of the head or subterms of output terms of previous body literals. At least one input argument of each body literal must be a subterm of one output argument of the immediately preceding body literal. The output arguments of the last literal must bind all critical output variables in the head.

Critical terms provide a focus of attention while searching for a specialization of an overly general unit clause. The main goal of the search is to find a body that binds the critical output variables.

In searching for the next literal of the body, there are two decisions to be made; what is the predicate symbol and what are its arguments. For the predicate symbol there are two possibilities. We could choose a known one including the one of the head of the clause, or we could introduce a new one if none of the existing predicates fits.

The next problem is to determine the arguments of the predicate chosen. Let us first discuss the case of the known predicate. The number of arguments is given by the arity of the predicate. The task is to select appropriate terms from the terms of the current clause.

If none of the known predicates yields an acceptable extension of the clause, a new predicate can be introduced. The search for the arguments of the new predicate is performed in a similar way. First, all the critical terms are tried. If this does not produce a satisfying result, a new variable is added.

```
Sieres(Predicate, Examples, Theory, Schemata):
  Until Examples provable from Theory,
    Let Bases be potential base cases in Examples.
    LearnBaseCases(Predicate, Bases, Theory).
    Let Examples be Examples - Bases.
    Let GS be an argument dependency graph in Schemata.
    Let G be an instance of GS.
    Set head(G) = LGG(Examples).
    For all E in Examples,
      Let $S_E$ be an instance of GS.
      Set head($S_E$) = E.
    Subject to constraints associated with GS,
      For all E in Examples,
        Instantiate body($S_E$).
      Let G = LGG({$S_E$|E in Examples}).
    If the last literal L in body(G) remains uninstantiated,
      Let Examples' = GenerateExamples(Predicate, Examples, Theory, G).
      Let Predicate' = NewPredicate(L).
      Sieres(Predicate', Examples', Theory, Schemata).
```

Figure 3: Pseudo-code for SIERES

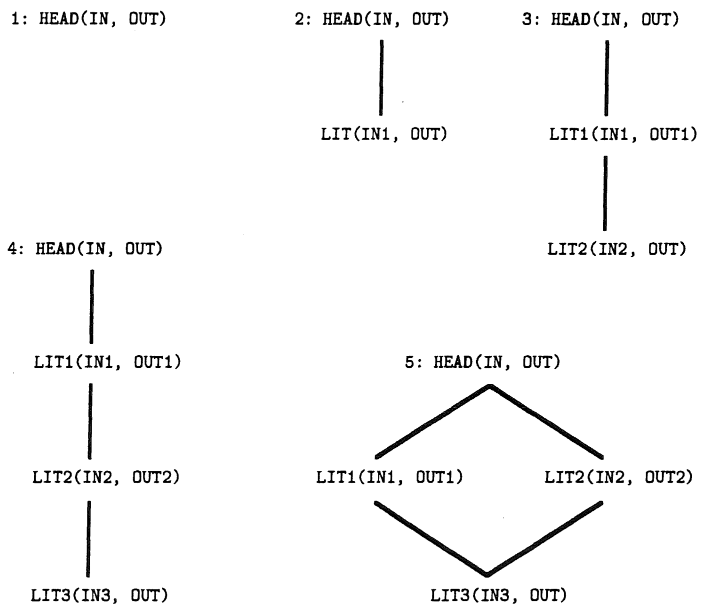

Figure 4: Dependency graphs for clauses of up to four literals

4 Examples

In this section we illustrate the method with examples, learning the definition of append, reverse, and DeMorgan's law.

4.1 Learning append

Let us assume at the beginning we have the training examples

$$E_1 = \texttt{append([s],[t],[s,t])}$$
$$E_2 = \texttt{append([d,e,f],[g,h],[d,e,f,g,h])}$$

and the mode declaration `append(+,+,-)`.[3] We also provide additional instantiations of append. The set of these facts is a subset of an h-easy model (Muggleton, et al., 1990).

SIERES starts out by forming the least generalization of the initial goals:

$$C = LGG(G_1, G_2) = \texttt{append([A|B],[C|D],[A|E])}.$$

but this clause is not acceptable because it does not produce the correct answers when applied to the initial goals. The query `append([s],[t],O)` yields `append([s],[t],[s|E])` and the query `append([d,e,f],[g,h],O)` yields `append([d,e,f],[g,h],[d|E])`. These answers are overly general because they contain unbound output variables. All the variables in the clause C except A are critical. The unbound output variable E is especially important.

SIERES searches for a specialization of the clause, starting with the next simplest rule schema:

$$\texttt{HEAD(IN,OUT)} :- \texttt{LIT(IN,OUT)}.$$

It then initializes a general explanation and two specific explanations for the training instances. SIERES assumes that the output variable has to be E in the general explanation and the proper instantiations in the special explanations:

$$G = \texttt{append([A|B],[C|D],[A|E])} :- \texttt{LIT(IN,E)},$$
$$S_1 = \texttt{append([s],[t],[s,t])} :- \texttt{LIT(IN,[t])},$$
$$S_2 = \texttt{append([d,e,f],[g,h],[d,e,f,g,h])} :- \texttt{LIT(IN,[e,f,g,h])}.$$

Furthermore, the input arguments for the missing literal have to be selected from the subterms of [A|B] and [C|D] and their instantiations. Next, SIERES searches for predicates that could fit into these explanations under these constraints. In the background

[3]If we also want to consider append in different modes, we could simply add the corresponding mode declarations. SIERES would treat the different versions of append as different predicates.

knowledge, it finds the facts $\texttt{append}([\ ], [\texttt{t}], [\texttt{t}])$ and $\texttt{append}([\texttt{e},\texttt{f}], [\texttt{g},\texttt{h}], [\texttt{e},\texttt{f},\texttt{g},\texttt{h}])$, which would complete the specific explanations. By generalizing these explanations SIERES obtains the clause

$$C' = LGG(S_1, S_2) = \texttt{append}([\texttt{A}|\texttt{B}], [\texttt{C}|\texttt{D}], [\texttt{A}|\texttt{E}]) :- \texttt{append}(\texttt{B}, [\texttt{C}|\texttt{D}], \texttt{E}).$$

4.2 Learning DeMorgan's Law

This example employs abductive inference and requires the invention of a new predicate. The goal is to learn the definition of a predicate `equiv` which implements DeMorgan's law for an arbitrary number of terms. The first argument of `equiv` is the input argument and is a negated conjunction. This expression is to be transformed into an equivalent disjunction of negations.

We start out with the following instances.

$$E_1 = \texttt{equiv}(\texttt{not}(\texttt{and}([\texttt{a}])), \texttt{or}([\texttt{not}(\texttt{a})]))$$
$$E_2 = \texttt{equiv}(\texttt{not}(\texttt{and}([\texttt{a},\texttt{b}])), \texttt{or}([\texttt{not}(\texttt{a}), \texttt{not}(\texttt{b})]))$$
$$E_3 = \texttt{equiv}(\texttt{not}(\texttt{and}([\texttt{c},\texttt{d},\texttt{e}])), \texttt{or}([\texttt{not}(\texttt{c}), \texttt{not}(\texttt{d}), \texttt{not}(\texttt{e})])$$

The least generalization is `equiv(not(and([A|B])),or([not(A)|C]))` but it is too general so SIERES seeks to specialize it using the next simplest rule schema. SIERES initializes general and specific explanations:

$$G = \texttt{equiv}(\texttt{not}(\texttt{and}([\texttt{A}|\texttt{B}])), \texttt{or}([\texttt{not}(\texttt{A})|\texttt{C}])) :- \texttt{LIT}(\texttt{IN}, \texttt{C}).$$
$$S_1 = \texttt{equiv}(\texttt{not}(\texttt{and}([\texttt{a}])), \texttt{or}([\texttt{not}(\texttt{a})])) :- \texttt{LIT}(\texttt{IN}, [\ ]).$$
$$S_2 = \texttt{equiv}(\texttt{not}(\texttt{and}([\texttt{a},\texttt{b}])), \texttt{or}([\texttt{not}(\texttt{a}), \texttt{not}(\texttt{b})])) :- \texttt{LIT}(\texttt{IN}, [\texttt{not}(\texttt{b})]).$$
$$S_3 = \texttt{equiv}(\texttt{not}(\texttt{and}([\texttt{c},\texttt{d},\texttt{e}])), \texttt{or}([\texttt{not}(\texttt{c}), \texttt{not}(\texttt{d}), \texttt{not}(\texttt{e})]) :- \texttt{LIT}(\texttt{IN}, [\texttt{not}(\texttt{d}), \texttt{not}(\texttt{e})]).$$

The critical terms are `B` and `C`. SIERES is unable to complete these explanations using existing predicates so it invents a new predicate with the critical terms as the arguments. SIERES then completes the special explanations *by assuming the following*:

$$\texttt{newpred13}([\ ], [\ ])$$
$$\texttt{newpred13}([\texttt{b}], [\texttt{not}(\texttt{b})])$$
$$\texttt{newpred13}([\texttt{d},\texttt{e}], [\texttt{not}(\texttt{d}), \texttt{not}(\texttt{e})])$$

The general clause corresponding to these specific explanations is

$$\texttt{equiv}(\texttt{not}(\texttt{and}([\texttt{A}|\texttt{B}])), \texttt{or}([\texttt{not}(\texttt{A})|\texttt{C}])) :- \texttt{newpred13}(\texttt{B}, \texttt{C}).$$

This clause is acceptable provided that SIERES can construct a general definition for `newpred13` that also helps explain additional instances of `equiv`.

In order to learn a definition for `newpred13`, SIERES first needs to construct more instances. This can be done by applying the definition of `equiv` to different instances, e.g. `equiv(not(and([d,e])),or([not(d),not(e)]))`. This way, SIERES automatically constructs a training set for the new predicate, which can be used to learn its general definition.

Ultimately, SIERES learns the following program.

```
equiv(not(and([A|B])),or([not(A)|C])) :- newpred13(B,C).
newpred13([A|B],[not(A)|C]) :- newpred13(B,C).
newpred13([ ],[ ]).
```

5 Current Status, Limitations, and Future Work

The ideas described in this paper have been implemented in an experimental system called SIERES. This program has been tested on logic programs including `append` and DeMorgan's law as described in the discussion of examples. In addition, SIERES can learn the following definitions of merge sort and `reverse`.

```
merge_sort([A|B],[C|D]) :- split([A|B],E,F),
                           merge_sort(E,G),
                           merge_sort(F,H),
                           merge(G,H,[C|D]).
```

SIERES invents a new predicate that adds an element at the end of a list in learning the following definition of `reverse`.

```
reverse([A|B],[C|D]) :- reverse(B,E),add_last(A,E,[C|D]).
add_last(A,[B|C],[B|D]) :- add_last(A,C,D).
add_last(A,[ ],[A]).
```

In this program, `A` is added to the end of `E` to get the output `[C|D]`. `A` is the last element of `D`, and `C` is the last element of `B` and the first element of `E`.

In the current implementation, training examples and background knowledge must be in the form of ground unit clauses before they can be used in learning. So theories given as PROLOG clauses are first used to generate ground instances. One of the immediate implementation goals is to allow a more natural use of the background theory.

More important research issues include the following. The current method does not allow for noise in the data. It learns one clause at a time and is not yet capable of learning disjunctive definitions. Only predicates that instantiate their output are learnable and predicates can be invented only at the end of a clause.

As a consequence of the current relatively tight argument dependency constraints, quick sort and `reverse` as defined below are not learnable. In both clauses the second argument of `append` is a term constructed from terms stemming from different literals.

```
qsort([H|T],Sorted) :- partition(H,T,L1,L2),
                       qsort(L1,SL1),
                       qsort(L2,SL2),
                       append(SL1,[H|SL2],Sorted).
reverse([A|B],[C|D]) :- reverse(B,E),append(E,[A],[C|D]).
```

This suggests that the current constraints may be too restrictive. So one of our immediate aims is to explore variations on the constraints looking to improve the space of learnable concepts while avoiding combinatorially explosive search.

6 Conclusion

We have described a learning method, implemented in a system called SIERES. The method integrates abduction and induction in a natural way. Constraints provided by syntactic least general generalization, critical terms, and argument dependency graphs focus a general to specific search for new clauses. Predicates are invented *if needed.*

The method invents new predicates in two steps. In the first step, a proof of the training instances is completed by assuming new facts built from a new predicate symbol. In the second step, the general clause derived in order to explain the training instances is used to generate more instances of the newly invented predicate. These instances are used for inducing a general definition of the new predicate.

Acknowledgments

We thank Mike Pazzani, Dennis Kibler and the graduate students of the AI & ML community at UCI for discussions of the ideas expressed in the paper and for discussions on related systems such as FOIL and FOCL. Thanks also to Yousri El Fattah, who participated in the initial stages of this research.

References

Kietz, J.-U., & Wrobel, S. (1991). Controlling the complexity of learning in logic through syntactic and task-oriented models. In S. Muggleton (Ed.), *First International Workshop on Inductive Logic Programming* . Porto, Portugal:

Kodratoff, Y., & Ganascia, J.-G. (1986). Improving the generalization step in learning. In R. Michalski, J. Carbonell, & T. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence approach* (pp. 215-244). Los Altos, CA: Morgan Kaufmann.

Lassez, J.-L., Maher, M. J., & Marriot, K. (1988). Unification revisited. In J. Minker (Eds.), *Foundations of deductive databases and logic programs* (pp. 587-626). Los Altos, CA: Morgan Kaufmann.

Lloyd, J. W. (1987). *Foundations of logic programming* (2nd ed.). Berlin: Springer-Verlag.

Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 256-269). Ann Arbor, MI: Morgan Kaufmann.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. *Proceeedings of the First Conference on Algorithmic Learning Theory* (pp. 1-14). Tokyo, Japan: Ohmsha.

O'Rorke, P., Morris, S., & Schulenburg, D. (1990). Theory formation by abduction: A case study based on the chemical revolution. In J. Shrager, & P. Langley (Eds.), *Computational Models of Scientific Discovery and Theory Formation* (pp. 197-224). San Mateo, CA: Morgan Kaufmann.

Plotkin, G. D. (1970). A note on inductive generalization. In B. Meltzer, & D. Michie (Eds.), *Machine Intelligence* (pp. 153-163). Edinburgh: Edinburgh University Press.

Plotkin, G. D. (1971). A further note on inductive generalization. In B. Meltzer, & D. Michie (Eds.), *Machine Intelligence* (pp. 101-124). Edinburgh: Edinburgh University Press.

Pople, H. E. (1973). On the mechanization of abductive logic. *Proceedings of the International Joint Conference on Artificial Intelligence* (pp. 147-152). Stanford, CA: Morgan Kaufmann.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239-266.

Rouveirol, C., & Puget, J. F. (1989). A Simple Solution for Inverting Resolution. In K. Morik (Ed.), *Proceedings of the Fourth European Working Session on Learning* (pp. 210-210). Montpellier: Pitman.

Shapiro, E. Y. (1983). *Algorithmic Program Debugging.* Cambridge, MA: The MIT Press.

Wirth, R. (1989). Completing logic programs by inverting resolution. In K. Morik (Ed.), *Proceedings of the Fourth European Working Session on Learning* (pp. 239-250). Montpellier: Pitman.