Technical Report AI-98-271, Artificial Intelligence Lab,
University of Texas at Austin, May 1998

An Experimental Comparison of Genetic Programming and Inductive Logic Programming on Learning Recursive List Functions

Lappoon R. Tang
Mary Elaine Califf
Raymond J. Mooney
Department of Computer Sciences
University of Texas
Austin, TX 78712-1188
Email : (rupert,mecaliff,mooney)@cs.utexas.edu

March 3, 1998

Abstract

This paper experimentally compares three approaches to program induction: *inductive logic programming* (ILP), *genetic programming* (GP), and *genetic logic programming* (GLP) (a variant of GP for inducing Prolog programs). Each of these methods was used to induce four simple, recursive, list-manipulation functions. The results indicate that ILP is the most likely to induce a correct program from small sets of random examples, while GP is generally less accurate. GLP performs the worst, and is rarely able to induce a correct program. Interpretations of these results in terms of differences in search methods and inductive biases are presented.

Keywords: Genetic Programming, Inductive Logic Programming, Empirical Comparison

This paper will also be submitted to the *8th Int. Workshop on Inductive Logic Programming, 1998.*

1 Introduction

In recent years, two paradigms for inducing programs from examples have become popular. One is *inductive logic programming* (ILP), in which rule-learning methods have been generalized to induce first-order Horn clauses (Prolog programs) from positive and negative examples of tuples satisfying a given target predicate [11, 7, 1]. The other is *genetic programming* (GP) [5, 6] in which genetic (evolutionary) algorithms are applied to tree structures instead of strings and used to induce programs in a functional language such as Lisp from input/output pairs. Although both of these approaches have attracted significant attention and spawned their own conferences, there has been little if any direct comparison of the two approaches despite the fact that they address the same general task of program induction from examples.

In this paper, we present experimental comparisons of several ILP methods and a standard GP method on the induction of four simple, recursive, list-manipulation functions (e.g. append, last). Such problems are typically used as test examples for ILP systems; however, although they also represent simple Lisp programs, they do not seem to be typical problems for testing GP.

GP is generally used to induce Lisp programs; however, the basic method just requires that programs be represented as trees. Consequently, to control for the differences in programming language, we also tested a variant of GP called GLP (Genetic Logic Programming) for inducing Prolog programs.

Since the utility of partially correct programs for such problems is debatable, the experiments we conducted measure the probability of inducing a completely correct program using different methods and different numbers of randomly selected I/O pairs. Overall, the ILP methods were found to produce correct programs more often than GP. GLP performed particularly poorly, almost never producing a correct program. The remainder of the paper describes the systems compared and the experiments conducted and discusses the results obtained.

2 Inductive Logic Programming

GP is used to induce function definitions rather than definitions of potentially non-functional logical relations used in ILP. When ILP is used to induce functions (represented by predicates with arguments for both input and output), most systems must also be given explicit negative examples of I/O pairs, i.e.

tuples of arguments that do *not* satisfy the predicate [10, 15]. Since GP does generally not utilize negative examples, fairly comparing it to such ILP methods is difficult. However, several ILP systems have recently been developed that induce functions from only positive examples of I/O pairs [2, 9, 13]. Consequently, we selected several of these methods, FOIDL, IFOIL, and FFOIL, to allow for a direct comparison to GP. Further comparison of these three ILP system is presented in [4].

2.1 FOIL

Since all of these systems are variations on FOIL [15], we first present a brief overview of this system. FOIL learns a function-free, first-order, Horn-clause definition of a *target* predicate in terms of itself and other *background* predicates. The input consists of extensional definitions of these predicates as a complete set of tuples of constants of specified types that satisfy the predicate. FOIL also requires negative examples of the target concept, which can be supplied directly or computed using a closed-world assumption.

Given this input, FOIL learns a Prolog program one clause at a time using a greedy set-covering algorithm that can be summarized as follows:

Let *positives-to-cover* = positive examples.
While *positives-to-cover* is not empty
    Find a clause, C, that covers a preferably large subset of *positives-to-cover*
        but covers no negative examples.
    Add C to the developing definition.
    Remove examples covered by C from *positives-to-cover*.

Clauses are constructed using a general-to-specific hill-climbing search. FOIL starts with an empty body and adds literals one at a time, at each step picking the one that maximizes an information gain metric, until the clause no longer covers negative examples. The algorithm terminates when the set of positive examples are completely covered by the set of learned clauses.

2.2 FOIDL and IFOIL

FOIDL [9] is based on FOIL but adds three important features:

1. Background knowledge is represented *intensionally* as a logic program.

2. No explicit negative examples need be supplied or constructed. An assumption of *output completeness* can be used instead to implicitly determine if a hypothesized clause is overly-general and, if so, to quantify the degree of over-generality by simply estimating the number of negative examples covered.

3. A learned program can be represented as a *first-order decision list*, an ordered set of clauses each ending with a cut. Only the first matching clause in the ordered list can be used to satisfy a goal. This representation is very useful for problems that are best represented as general rules with specific exceptions.

Functions are learned without explicit negative examples, by assuming that any computed ouput which does not match the single, correct output for the function represents a covered negative example. IFOIL (Intensional FOIL) is just FOIDL without the use of decision lists. The code for these systems is available at `http://www.cs.utexas.edu/users/ml`.

2.3 FFOIL

FFOIL [13] is a descendant of FOIL with modifications similar to FOIDL's, that specialize it for learning functional relations. First, FFOIL assumes that the final argument of the relation is an *output argument* and that the other arguments of the relation uniquely determine the output argument. This assumption is used to provide implicit negative examples: each positive example under consideration whose output variable is not bound by the clause under construction is considered to represent one positive and $r-1$ negatives, where r is the number of constants in the range of the function. Second, FFOIL assumes that each clause will end in a cut, so that previously covered examples can be safely ignored in the construction of subsequent clauses. Thus, FFOIL, like FOIDL, constructs first-order decision lists, though it constructs the clauses in the same order as they appear in the program, while FOIDL constructs its clauses in the reverse order. The code we used for all the experiments was FFOIL version 1.0.

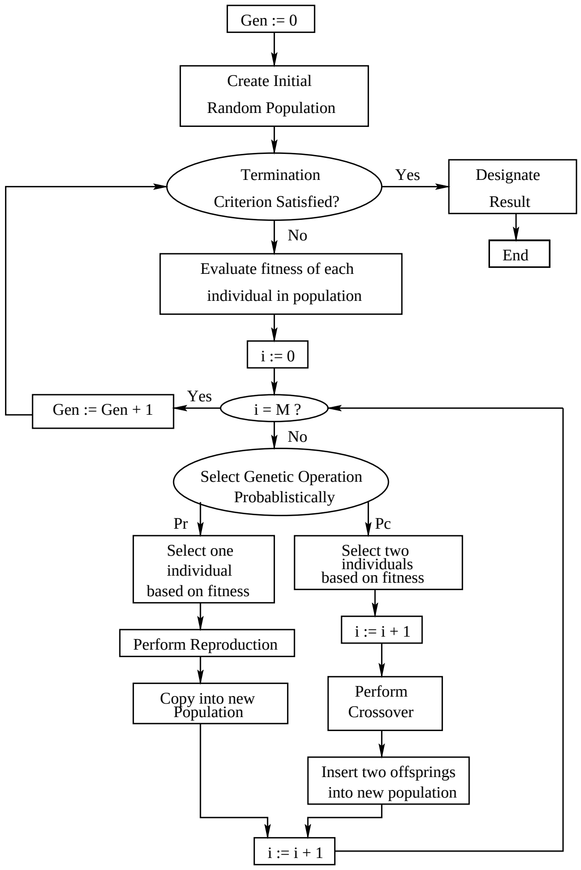


Figure 1: Flowchart for the GP paradigm

3 Genetic Programming

Genetic programming applies a genetic algorithm to Lisp programs instead of bit strings. It starts with an initial population of randomly generated programs composed of functions and terminals from the appropriate problem domain. Using the standard genetic operators of crossover and reproduction, it evolves the population of programs according to a *fitness measure* (e.g. accuracy on a training set of I/O pairs) until an individual program satisfying a *measure criteria* (e.g. 100% accuracy on the training set) is found.

Figure 1 shows an outline of the algorithm. Reproduction is performed by first selecting an individual according to some selection method[1] and then copying the individual without any modification to the new population. Crossover is done by selecting two parents by the same fitness-based method and then choosing an arbitrary crossover point for each parent. The entire subtrees below the crossover points of the two parents are then "swapped" to give two new individuals. Two properties need to be satisfied in GP for the creation of completely correct programs: the *sufficiency* and *closure* properties. The former requires that the set of functions and terminals supplied to the system are sufficient for the expression of a solution to the problem and the latter requires that each function in the function set is able to accept as its arguments any value that may be returned by any of the function in the function set and any value that the terminals may assume. The first property is easy to maintain. Since each function in the function set may return its data type, to maintain the second property we need to specify a list of constraints to the functions (e.g. the set of functions or terminals that are allowed to be the arguments of a certain function) so that GP will always generate syntactically correct programs. Since choosing arbitrary points for crossover may not necessarily result in syntactically legal programs, a test on the validity of offspring is performed after each crossover. The GP code we used for all the experiments is that provided as a supplement to [5].

4 Genetic Logic Programming

Genetic logic programming was apparently first suggested in [16] as an alternative approach to ILP. The idea is to use genetic search on the space of logic programs. We have implemented our own version of GLP for inducing Prolog programs. *Function sufficiency* is maintained straightwardly by using the background predicates provided to ILP and the target predicate itself. However, to observe *terminal sufficiency*, a sufficient set of distinct variables must be provided. Maintaining the *closure property* requires specifying the type of each variable and the types of the arguments of each predicate to the system. When GLP generates programs, it accepts only those that satisfy all the typing constraints.

Logic programs are represented as logical expressions in Lisp syntax. For example, the program for `(member ?x, ?y)` given the background knowledge

[1] We used *fitness-proportionate* selection.

(components (?x . ?y) ?x ?y) can be represented as:

```
(or (components ?y ?x ?z)
    (and (components ?y ?v ?z) (member ?x ?z)))
```

Like FOIDL and IFOIL, GLP is provided with intensional definitions of all the background predicates, and standard Prolog execution is used to determine accuracy on the I/O pairs in the training set. Since the original GP code does not work with a logical representation, we modified the code to work with this new representation and interfaced it with a Prolog interpreter written in Lisp.[2] The same Prolog interpretor is used with FOIDL and IFOIL.

5 Experimental Evaluation

5.1 Experimental Methodology

To compare the performance of ILP, GP and GLP we chose four functional list processing programs: conc (aka append), last, shift, and translate (aka sublis). These are a series of list-processing examples and exercises from Chapter 3 of the Bratko's text on Prolog [3] previously used to evaluate ILP systems [14]; however, such list processing functions are standard examples in both Lisp and Prolog. For each problem, the background knowledge provided consists of the list functions encountered previously in the text, which are guaranteed to be sufficient but may include irrelevant functions. The training data for the experiments are randomly selected from the universe of input lists over three atoms up to a length of three.

The standard experiments with GP and GLP were run with a population size of 200 and a maximum number of generations of 50. The fitness metric was the percentage of training examples for which correct output is generated. Since Prolog programs can generate multiple outputs, in GLP an example is considered correct if and only if the single correct output is generated.

Since partially-correct list programs are of limited utility, we present learning curves that measure the probability that a completely correct program is produced from random training sets of positive examples of various sizes. This probability was measured by running 20 independent trials in which each system is trained on the same set of random positive examples of a given size and

[2]The Prolog interpreter we used was that provided with [12].

determining the percentage of trials in which the system learns a completely correct definition (as determined by a manual inspection of the resulting programs).

5.2 Experimental Results

The results are shown in Figure 2. In all cases, the performance of GLP was the worst; it did not produce a completely correct program in any of the trials. FOIDL and IFOIL's results on learning conc were clearly much better than those of GP and GLP. Both ILP systems experienced a surge in their performance given only a few training examples and gradually improved given more training data, ending with a percentage of correct programs of more than 70%. GP and GLP, however, did not seem to improve their performance with more training data and could not produce more than 20% of correct programs in the experiment. FFOIL did poorly at the beginning but managed to jump up to more than 50% correct programs given all the training data. The performance of FOIDL and FFOIL in learning last was significantly better than that of GP which was slight better than IFOIL. The performance of the ILP systems on learning shift was significantly better than GP. FFOIL could output a correct program for each of the trials given 10 or more training examples while FOIDL could do so with 15 or more training examples. The percentage of correct definitions for GP given all the training examples was 20%. In learning translate, two of the ILP systems, FOIDL and IFOIL, were significantly better than GP and GLP. FFOIL's performance was about the same as the GP's and GLP's. Given 10 training examples, one-third of the training data, the ILP and GP systems performed roughly the same; the percentage of correct programs for all of the systems was less than 20%. However, they began to diverge after 15 training examples or more where both FOIDL and IFOIL had 75% correct programs and 0% for GP and GLP given all the training data.

Overall, the ILP systems exhibited the best performance in terms of the probability of producing a correct program. GP came second and GLP was clearly the worst.

5.3 Discussion of Results

The best explanation for the poor performance of GLP compared to GP is the fact that it was less capable of converging on the training data. In fact, GLP

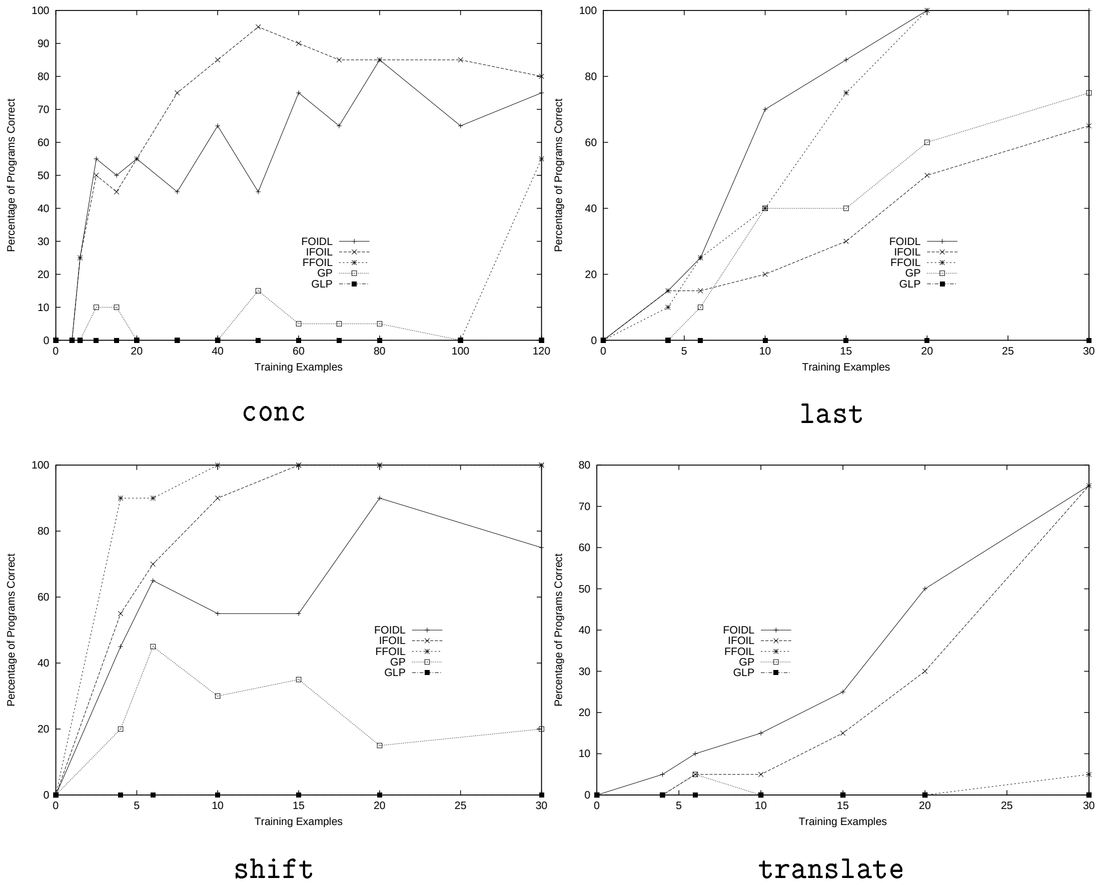
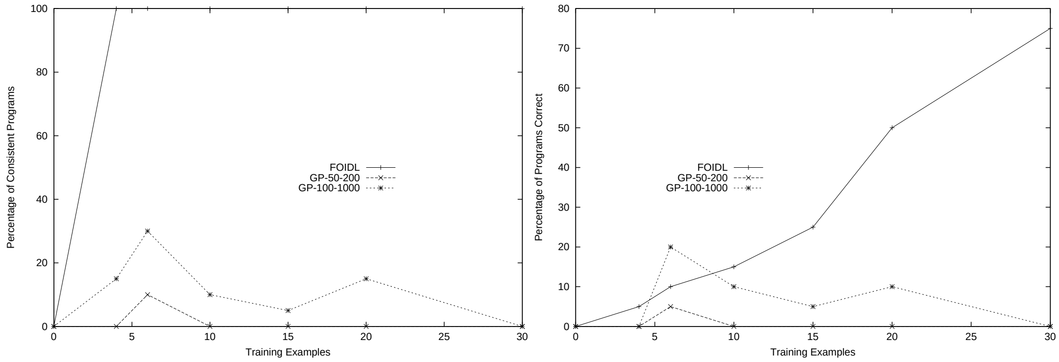


Figure 2: Results for functional list processing programs



Convergence Accuracy

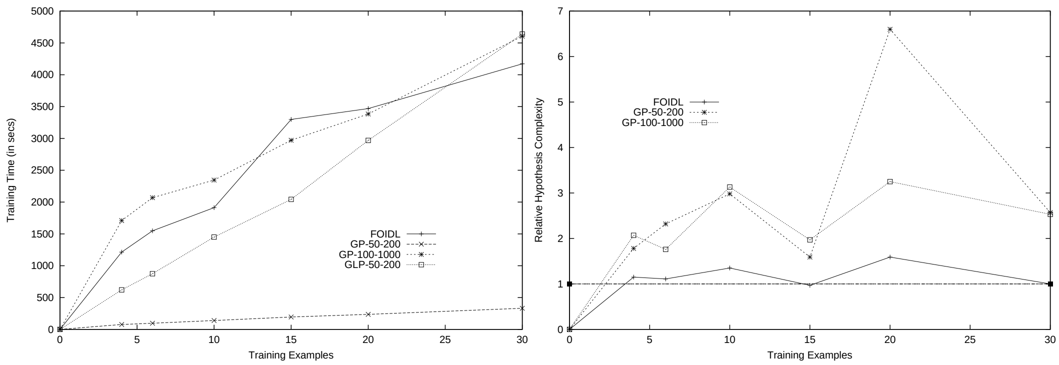
Figure 3: Effect of population size and number of generations on the convergence and accuracy of GP on translate.

was never able to converge on any of the problems. This clearly suggests that the current GP method works better with a functional language than a logical one. Due to the fact that the calling hierarchy is explicit in a functional program's tree structure, the swapping of subtrees is generally a better heuristic for generating promising programs than swapping literals or arguments in the representation of Prolog programs, where the calling hierarchy is "implicit" in the sharing of variables between literals.

While GP works better with a functional representation, most of its results are significantly worse than those of ILP. Only in learning last were the GP results somewhat competitive. Although much better than GLP, GP also had trouble converging on the training data for some of the problems and therefore could not find a correct program while the ILP systems were always able to find at least a consistent program. For example on translate,[3] GP had a difficult time converging, and giving the system twice as many generations and 5 times the population only improved the performance moderately. The effect of both of these factors on the convergence and accuracy of GP is shown in Figure 3. When GP found a program consistent with the training data for translate, it was frequently correct; but it was normally unable to find one.

The effect of the increase in search on GP's training time is shown in

[3]We have chosen translate as an illustration since it is the hardest of the problems; it has the largest number of irrelevant background functions, requires the most number of symbols to represent in both Prolog and Lisp, and it took FOIDL the longest to train.



Training Time **Relative Program Complexity**

Figure 4: Training time on translate and relative program complexity on shift.

Figure 4. All of the systems in the comparison were run on a Sun Sparc 5 under Allegro Common Lisp v4.3.1. FOIDL and IFOIL were more accurate than GP (with a population of 1000 individuals and 100 generations) on translate even though all of the systems used roughly the same amount of training time. Increasing the search in GP much further seems infeasible given that the growth in training time would be quite significant. FOIDL, IFOIL, and GLP all spend a fair fraction of the training time just proving examples using Prolog as evident from the training time of GLP compared to GP. The overhead of using an unoptimized Prolog interpreter in Lisp accounts for the significant difference between GLP-50-200 and GP-50-200, and therefore also largely accounts for the difference between FOIDL and GP-50-200. Therefore, if we just consider the amount of search, the effort expended by FOIDL and IFOIL is actually much less than their run time implies. The search methods employed in the ILP systems are more specific to the particular representation of Prolog programs and perform a more directed, systematic search. The combination of greedy covering and general-to-specific search was apparently more effective at finding consistent programs than the more general evolutionary search. Standard genetic approaches do not take advantage of the general-to-specific ordering on hypotheses [8] and therefore their search is less directed.

Also, the current GP system was less robust at tolerating irrelevant background knowledge. This is clear from GP's significant difference in performance on last versus shift; GP had 75% of correct programs in the former and only 20% in the latter given all the training data. The difficulty of the two programs is fairly similar in terms of the number of symbols required to represent them; last requires 10 symbols and shift requires 12 symbols using background functions provided to them in the experiments. However, in learning last, the systems were given only one irrelevant background function while in learning shift, they were given seven. Even though GP frequently converged when learning shift (50% of the time on average compared to less than 2% on translate), it very often acquired an incorrect program that happened to be consistent with the training data (see the performance of GP on shift in Figure 2). By contrast, the ILP systems seem more robust to irrelevant background, frequently learning shift correctly from even fewer examples than last.

Part of the explanation for this difference may lie in the fact that the ILP systems generally induced simpler programs. Even when they were correct, the programs learned by GP were usually more complex and convoluted. GP tends to search a relatively more complex hypothesis space than ILP. To more carefully examine this issue, we calculated the average relative complexity of learned shift programs for FOIDL and GP. Relative complexity is calculated by counting the number of symbols used in a program compared to the known simplest program. For example, the program (rest (append x (list (first x)))) has a complexity of 6 which is the smallest possible for the Lisp function shift given the background supplied. Dividing this number by the complexity of the simplest known program gives us the relative program complexity (in this case 1.0). We measured relative complexity only for consistent programs since it is easy to a construct trivial inconsistent programs. We used shift instead of translate since GP converged more frequently for the former, and the shift problem has a comparable amount of irrelevant background knowledge. Results on average relative program complexity are shown in Figure 4. Overall, GP learns much more complex programs than FOIDL. This is undoubtably due to the fact that the ILP systems have a stronger "Occam's Razor" bias and specifically attempt to learn short programs.

6 Future Work

The experiments above suggest a number of ways in which the performance of GP might be improved. One is to develop a more refined fitness function that includes a penalty for program size in order to bias the search towards simpler hypotheses. In addition, GLP might be improved by a fitness function that partially rewards correct outputs for an example even when the current Prolog program also produces additional incorrect outputs. Using more specialized genetic operators could also help. For example, one could possibly experiment with crossover operators that are attempt to select "good" crossover points.

As originally suggested by [16], perhaps the right combination of GP and ILP search techniques could prove the most effective. ILP systems generally employ very directed (e.g. greedy) search that frequently works quite well, but can get stuck at local minima. On the other hand, search in GP could clearly benefit from some of the directness of ILP methods.

We have only compared GP and ILP on learning simple recursive list functions. It would of course be informative to compare the approaches on other problems that have been used to test either ILP or GP systems.

7 Conclusions

ILP and GP are both interesting approaches to program induction; however, there has been inadequate experimental comparison to uncover their individual strengths and weaknesses. We hope the comparisons in this paper represent a first step towards more concrete comparisons of the two approaches. On the induction of several simple list manipulation functions, our experiments reveal that ILP is generally more accurate at inducing correct programs given limited data and computing resources. On these problems, the more systematic search employed by ILP, which specifically exploits the semantics and modularity of Prolog clauses, appears to give it an advantage over the more general and search-intensive methods employed by GP. However, these comparisons also indicate ways in which the performance of GP might be improved that could eventually lead to a successful synthesis that exploits the strengths of both approaches.

8 Acknowledgements

We would like to thank John Koza for making the GP code available and Ross Quinlan for providing FFoil. This research was partially supported by the National Science Foundation through grant IRI-9704943 and by a grant from the Daimler-Benz Research and Technology Center.

References

[1] F. Begadano and D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering.* MIT Press, 1995.

[2] F. Bergadano and D. Gunetti. An interactive system to learn functional logic programs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1044–1049, Chambery, France, 1993.

[3] I. Bratko. *Prolog Programming for Artificial Intelligence.* Addison Wesley, Reading:MA, 1990.

[4] M. E. Califf and R. J. Mooney. Advantages of decision lists and implicit negatives in inductive logic programming. *New Generation Computing*, 16(3), to appear 1998.

[5] J. R. Koza. *Genetic Programming: On the Programming of Computers by means of Natural Selection.* MIT Press, MIT, 1992.

[6] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, MIT, 1994.

[7] N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, 1994.

[8] T. Mitchell. *Machine Learning.* McGraw-Hill, New York, NY, 1997.

[9] R. J. Mooney and M. E. Califf. Induction of first-order decision lists: Results on learning the past tense of English verbs. *Journal of Artificial Intelligence Research*, 3:1–24, 1995.

[10] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, Tokyo, Japan, 1990. Ohmsha.

[11] S. H. Muggleton, editor. *Inductive Logic Programming.* Academic Press, New York, NY, 1992.

[12] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann, San Mateo, California, 1992.

[13] J. R. Quinlan. Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, 5:139–161, 1996.

[14] J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In *Proceedings of the European Conference on Machine Learning*, pages 3–20, Vienna, 1993.

[15] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[16] P. A. Whigham and R. I. McKay. Genetic programming and inductive logic. Technical Report CS14/94, University College, University of New South Wales, 1994.