FOIL: A Midterm Report

J. R. Quinlan and R. M. Cameron-Jones

Basser Department of Computer Science
University of Sydney
Sydney Australia 2006
quinlan@cs.su.oz.au, mcj@cs.su.oz.au

Abstract: FOIL is a learning system that constructs Horn clause programs from examples. This paper summarises the development of FOIL from 1989 up to early 1993 and evaluates its effectiveness on a non-trivial sequence of learning tasks taken from a Prolog programming text. Although many of these tasks are handled reasonably well, the experiment highlights some weaknesses of the current implementation. Areas for further research are identified.

1. Introduction

The principal differences between zeroth-order and first-order supervised learning systems are the form of the training data and the way that a learned theory is expressed. Data for zeroth-order learning programs such as ASSISTANT [Cestnik, Kononenko and Bratko, 1986], CART [Breiman, Friedman, Olshen and Stone, 1984], CN2 [Clark and Niblett, 1987] and C4.5 [Quinlan, 1992] comprise preclassified cases, each described by its values for a fixed collection of attributes. These systems develop theories, in the form of decision trees or production rules, that relate a case's class to its attribute values. In contrast, the input to first-order learners (usually) contains ground assertions about a number of multi-argument predicates or relations and the learned theory consists of a logic program, restricted to Horn clauses or something similar, that predicts when a vector of arguments will satisfy a designated predicate.

Early first-order learning systems such as MIS [Shapiro, 1983] and MARVIN [Sammut and Banerji, 1986] were based on the notion of first-order proof. A partial theory was modified when it was insufficient to prove a known fact or able to (mis)prove a known fiction. The dependence on finding proofs meant that systems like these were relatively slow, most of the time being consumed in theorem-proving mode, so that they were able to analyse only small training sets. Later systems such as FOIL [Quinlan, 1990, 1991] and GOLEM [Muggleton and Feng, 1990] abandoned proof-based algorithms for more efficient methods; GOLEM uses Plotkin's *relative least general generalisation* to form clauses while FOIL uses a divide-and-cover strategy adapted from zeroth-order learning. These approaches have proved to be more efficient and robust, enabling larger training sets to be analysed to learn more complex programs. Later systems such as CHAM [Kijsirikul, Numao and Shimura, 1991], FOCL [Pazzani, Brunk and Silverstein, 1991; Pazzani and Kibler, 1992] ILE [Rouveirol, 1991] and FORTE [Richards and Mooney, 1991] often contain elements of both proof-based and empirical approaches.

This paper examines FOIL, summarising its development over the last four years. After outlining its key features, we describe an experiment designed to evaluate its program-writing ability, using problems that human Prolog students are expected to be able to master. Not surprisingly, FOIL has difficulty with some of the problems. We discuss FOIL's shortcomings and what they tell us about the research that will be needed to extend it into a useful logic programming tool.

2. FOIL

In a nutshell, FOIL is a system for learning function-free Horn clause definitions of a relation in terms of itself and other relations. The program is actually slightly more flexible since it can learn several relations in sequence, allows negated literals in the definitions (using standard Prolog semantics), and can employ certain constants in the definitions it produces.

FOIL's input consists of information about the relations, one of which (the *target relation*) is to be defined by a Horn clause program. For each relation it is given a set of tuples of constants that belong to the relation. For the target relation it might also be given tuples that are known not to belong to the relation; alternatively, the closed world assumption may be invoked to state that no tuples, other than those specified, belong to the target relation. Tuples known to be in the target relation will be referred to as $\oplus$ tuples and those not in the relation as $\ominus$ tuples. The learning task is then to find a set of clauses for the target relation that accounts for all the $\oplus$ tuples while not covering any of the $\ominus$ tuples.

The basic approach used by FOIL is an AQ-like covering algorithm [Michalski, Mozetič, Hong and Lavrač, 1986]. It starts with a *training set* containing all $\oplus$ and $\ominus$ tuples, constructs a function-free Horn clause to 'explain' some of the $\oplus$ tuples, removes the covered $\oplus$ tuples from the training set, and continues with the search for the next clause. When clauses covering all the $\oplus$ tuples have been found, they are reviewed to eliminate any redundant clauses and reordered so that any recursive clauses come after the non-recursive base cases.

Perfect definitions that exactly match the data are not always possible, particularly in real-world situations where incorrect values and missing tuples are to be expected. To get around this problem, FOIL uses encoding-length heuristics to limit the complexity of clauses and programs. The final clauses may cover most (rather than all) of the $\oplus$ tuples while covering few (rather than none) of the $\ominus$ tuples. See [Quinlan, 1990] for details.

2.1 Finding a Clause

FOIL starts with the left-hand side of the clause and specialises it by adding literals to the right-hand side, stopping when no $\ominus$ tuples are covered by the clause or when encoding-length heuristics indicate that the clause is too complex. As new variables are introduced by the added literals, the size of the tuples in the training set increases so that each tuple represents a possible binding for all variables that appear in the partially-developed clause.

If the target relation R has k arguments, the process of finding one clause for the definition of R can be summarised as follows:

- Initialise the clause to

  $$R(V_1, V_2, ..., V_k) \leftarrow$$

  and a local training set T to the $\oplus$ tuples not covered by any previous clause and all the $\ominus$ tuples.
- While T contains $\ominus$ tuples and is not too complex:
  - Find a literal L to add to the right-hand side of the clause.
  - Form a new training set T':
    - for each tuple t in T, and
    - for each binding b of any new variables introduced by literal L,
      - if the tuple $t.b$ (obtained by concatenating t and b) satisfies L, then add $t.b$ to T' with the same label ($\oplus$ or $\ominus$) as t.
  - Replace T by T'.
- Prune the clause by removing any unnecessary literals.

Although FOIL incorporates a simple backup mechanism, the clause-building process is essentially a greedy search; once a literal is added to a clause, alternative literals are usually not investigated.

The key question is how to determine appropriate literals to append to the developing clause. FOIL uses two criteria: a literal must either help to exclude unwanted $\ominus$ tuples from the training set, or must introduce new variables that may be needed for future literals. Literals of the first kind are called *gainful* while *determinate* literals are included primarily because they introduce new variables.

2.2 Choosing Gainful Literals

Consider the partially developed clause

$$R(V_1, V_2, ..., V_k) \leftarrow L_1, L_2, ..., L_{m-1}$$

containing variables $V_1, V_2, ..., V_x$. Each tuple in the training set T looks like $\langle c_1, c_2, ..., c_x \rangle$ for some constants $\{c_j\}$, and represents a ground instance of the variables in the clause. Now, consider what happens when a literal L_m of the form

$$P(V_{i_1}, V_{i_2}, ..., V_{i_p})$$

is added to the right-hand side. If the literal contains one or more new variables, the arity of the new training set will increase; let x' denote the number of variables in the new clause. Then, each tuple in the new training set T' will be of the form $\langle d_1, d_2, ..., d_{x'} \rangle$ for constants $\{d_j\}$, and will have the following properties:

- $\langle d_1, d_2, ..., d_x \rangle$ is a tuple in T, and
- $\langle d_{i_1}, d_{i_2}, ..., d_{i_p} \rangle$ is in the relation P.

That is, each tuple in T' is an extension of one of the tuples in T, and the ground instance that it represents satisfies the literal. Every tuple in T thus gives rise to zero or more tuples in T' with the $\oplus$ or $\ominus$ label of a tuple in T' being copied from its ancestor tuple in T.

Let T_+ denote the number of $\oplus$ tuples in T and T'_+ the number in T'. The effect of adding a literal L_m can be assessed from an information perspective as follows. The information conveyed by the knowledge that a tuple in T has label $\oplus$ is given by

$$I(T) = -log_2(T_+ \ / \ |T|)$$

and similarly for $I(T')$. If $I(T')$ is less than $I(T)$ we have 'gained' information by adding the literal L_m to the clause; if s of the tuples in T have extensions in T', the total information gained about the $\oplus$ tuples in T is

$$gain(L_m) = s \times (I(T) - I(T')).$$

FOIL explores the space of possible literals that might be added to a clause at each step, looking for the one with greatest positive gain.

The form of the gain metric allows significant pruning of the literal space, so that FOIL can usually rule out large subspaces without having to examine any literals in them. If a potential literal contains new variables, it is possible to compute the maximum gain that could be obtained by replacing some or all of them with existing variables. When the maximum gain is below that of some literal already considered, the literals resulting from such replacements do not need to be investigated.

Another form of pruning involves literals that use the target relation itself. Since we do not want FOIL to produce non-executable programs that fail due to infinite recursive looping, recursive definitions must be screened carefully. Recursive literals that could lead to problems are barred from consideration, as described below.

2.3 Determinate Literals

Some clauses in reasonable definitions will inevitably contain literals with zero gain. Suppose, for instance, that all objects have a value for some property D, and the literal $D(X, Y)$ defines the value Y for object X. Since this literal represents a one-to-one mapping from X to Y, each tuple in T will give rise to exactly one tuple in T' and so the gain of the literal will always be zero. We could also imagine a literal $P(X, Y)$ that, for any value of X, supplied several possible values for Y. Such a literal might even have negative gain.

If X is a previously defined variable and Y a new variable, there is an important difference between adding literals $D(X, Y)$ and $P(X, Y)$ to a clause; the first will produce a new training set of exactly the same size, while the second may exclude some $\oplus$ tuples or may cause the number of tuples in the training set to grow. This is the key insight underlying *determinate* literals, an idea inspired by GOLEM's *determinate terms* [Muggleton and Feng, 1990]: the value of each new variable is forced or determined by the values of existing variables.

More precisely, suppose that we have an incomplete clause

$$R(V_1, V_2, ..., V_k) \leftarrow L_1, L_2, ..., L_{m-1}$$

with an associated training set T as before. A literal L_m is determinate with respect to this partial clause if L_m contains one or more new variables and there is exactly one extension of each $\oplus$ tuple in T, and no more than one extension of each $\ominus$ tuple, that satisfies L_m. The idea is that, if L_m is added to the clause, no $\oplus$ tuple will be eliminated and the new training set T' will be no larger than T.

FOIL notes determinate literals found while searching for gainful literals as above. The maximum possible gain is given by a literal that excludes all $\ominus$ tuples and no $\oplus$ tuples; in the notation used before, this gain is $T_+ \times I(T)$. Unless a literal is found whose gain is close to ($\geq$ 80% of) the maximum possible gain, FOIL adds *all* determinate literals to the clause and tries again. This may seem rather extravagant, since it is unlikely that all these literals will be useful. However, FOIL incorporates clause-refining mechanisms that remove unnecessary literals as each clause is completed, so there is no ultimate penalty for this all-in approach. Since no $\oplus$ tuples are eliminated and the training set does not grow, the only computational cost is associated with the introduction of new variables and the corresponding increase in the space of subsequent possible literals. It is precisely the enlargement of this space that the addition of determinate literals is intended to achieve.

There is a potential runaway situation in which determinate literals found at one cycle give rise to further determinate literals at the next *ad infinitum*. To circumvent this problem, FOIL borrows another idea from GOLEM. The *depth* of a variable is determined by its first occurrence in the clause. All variables in the left-hand side of the clause have depth 0; a variable that first occurs in some literal has depth one greater than the greatest depth of any previously-occurring variable in that literal. By placing an upper limit on the depth of any variable introduced by a determinate literal, we rule out indefinite runaway. This limit does reduce the class of learnable programs. However, the stringent requirement that a determinate literal must be uniquely satisfied by *all* $\oplus$ tuples means that this runaway situation is unlikely and FOIL's default depth limit of 5 is rarely reached.

2.4 Further Literal Forms

We are now moving into areas covered by recent extensions to FOIL. The first of these concerns the kinds of literals that can appear in the right-hand side of a clause.

Early versions of FOIL considered literals of the forms

- $P(W_1, W_2, ..., W_p)$, $\neg P(W_1, W_2, ..., W_p)$
  where P is a relation and the W_i's are variables, at least one of which must have occurred already in the clause; and
- $V_i = V_j$, $V_i \neq V_j$
  that compare the values of existing variables.

Two further forms have now been added.

In the first of these, certain constants can be identified as *theory* constants that can appear explicitly in a definition. Examples might include a constant [] representing the null list in list-processing tasks, or the integers 0 and 1 in tasks that involve the natural numbers. For such a theory constant c, FOIL will also consider literals of the forms

$$V_i = c, \quad V_i \neq c$$

where V_i is a variable of the appropriate type that appears earlier in the clause. This minor addition is equivalent to declaring a special relation is-c for each such constant c; in fact, the extension is implemented in this way.

The second extension is more substantial. Relations encountered in the real world are not limited to discrete information but commonly include numeric fields as well. We could imagine simple relations such as

atomic-weight(E,W)

that provides the (numeric) atomic weight W of each element E, or

quote(C,B,S)

detailing the buy and sell prices for a commodity C. As a first step towards being able to exploit numeric information like this, FOIL now includes literal types

$$V_i > k, \quad V_i \leq k, \quad V_i > V_j, \quad V_i \leq V_j$$

that allow an existing variable V_i with numeric values to be compared against a threshold k found by FOIL or against another variable V_j of the same type. Such an extension falls a long way short of Prolog facilities that allow a continuous value for V_i to be computed in the clause; however, it does permit bound numeric values to be used in conditions on the right-hand side of a clause.

2.5 Managing Recursion

Recursive theories are expressive and hence powerful, so that the ability to learn recursive programs is one of the principal advantages of first-order systems like GOLEM and FOIL. The increase in expressiveness, however, is counterbalanced by the care that must be taken to avoid nonsensical recursion.

As an illustration, consider the task of learning a program for multiplication of non-negative integers in terms of addition and decrement. We might have three relations:

mult(A,B,C)	meaning	C = A × B
plus(A,B,C)		C = A + B
dec(A,B)		B = A − 1.

A suitable definition for multiply is

$$\mathsf{mult(A,B,C)} \leftarrow \mathsf{A{=}0,\ C{=}0}$$
$$\mathsf{mult(A,B,C)} \leftarrow \mathsf{dec(A,D),\ plus(B,E,C),\ mult(D,B,E)}$$

where the last clause captures the identity

$$A \times B = B + (A - 1) \times B.$$

This definition seems intuitively to be well-behaved in the sense that it will always terminate. On the other hand, a simpler definition

$$\mathsf{mult(A,B,C)} \leftarrow \mathsf{mult(B,A,C)}$$

will clearly lead to an infinite recursive loop. How does FOIL, which is biased towards finding simpler definitions, eschew the latter in favour of the former? The short answer is that, as a clause is being developed, recursive literals must satisfy certain criteria for inclusion in the right-hand side. In particular, a recursive literal on the right-hand side must be judged to be less than the head of the clause in some ordering of literals.

The earliest version of FOIL used a method based on discovering an ordering of the constants appearing in tuples. This method guaranteed that a single clause could not lead to a recursive loop by calling itself directly. The order discovery was removed in following releases, which relied on the user specifying the constants of each type in an appropriate order. Order discovery mechanisms have been reinstated in the most recent versions and the method of ordering recursive literals has been generalised so that the guarantee now applies to sets of clauses for a single relation, not just to a single clause. The following is meant to give an informal sketch of the idea, with a complete discussion available in [Cameron-Jones and Quinlan, 1993].

Returning to the multiply example above, we see that the clause for the general case

$$\mathsf{mult(A,B,C)} \leftarrow \mathsf{dec(A,D),\ plus(B,E,C),\ mult(D,B,E)}$$

cannot lead to infinite recursion since the literal dec(A,D) guarantees that D is always less than A; mult(D,B,E) is thus less than mult(A,B,C) in an intuitive ordering of mult literals. FOIL assumes that some relations provided for a task will behave like dec in establishing an ordering of their arguments and attempts to identify them. For every relation R and every pair of arguments A, B of R that are of the same type Q, FOIL asks:

> Are there orderings of the constants of type Q that are consistent with the hypothesis that $A < B$?

When answers to all these questions have been determined, FOIL establishes a single definitive ordering of the constants of type Q so that the number of such inequalities is maximised.

The now-fixed ordering of constants of each type allows us to determine rankings among pairs of variables in an incomplete clause. If such a clause contains variables $V_1, V_2, ..., V_x$ and the training set consists of tuples of constants $\langle d_{a1}, d_{a2}, ..., d_{ax} \rangle$, $a = 1, 2, ..., |T|$, then $V_i < V_j$ if they belong to the same type and d_{ai} always comes before d_{aj} in the constant ordering for that type.

The inequalities among pairs of variables can be extended to an ordering of literals involving a predicate R and variables. In broad terms, if $W_1, W_2, ...$ denote variables in $V_1, V_2, ..., V_x$, then

$$
\begin{aligned}
&R(W_1, W_2, ..., W_k) < R(V_1, V_2, ..., V_k) \text{ if} \\
&\qquad W_\alpha < V_\alpha, \text{ or} \\
&\qquad W_\alpha = V_\alpha \text{ and } W_\beta < V_\beta, \text{ or} \\
&\qquad W_\alpha = V_\alpha \text{ and } W_\beta = V_\beta \text{ and } W_\gamma < V_\gamma, \text{ or ...}
\end{aligned}
$$

Here α, β, γ etc. denote argument positions that, together with the ordering of variables in the clause, specify a particular ordering of the literals involving R.

Suppose now that we have an incomplete definition for relation R that consists of zero or more completed clauses and a partial clause. A recursive literal $R(W_1, W_2, ..., W_k)$ can be added to the right-hand side of the developing clause only when there are values of α, β etc. as above so that

- this literal is less than the left-hand side of the clause, and
- the same is true for all recursive literals in the completed clauses.

This may sound complex but its implementation is simple and efficient. The restriction on recursive literals in the right-hand side of clauses prevents infinite recursive loops due to a definition of R calling itself directly, yet does not exclude even complex recursive definitions such as that for Ackermann's function:

```
Ack(A,B,C) ← A=0, dec(C,B)
Ack(A,B,C) ← B=0, dec(A,D), Ack(D,E,C), dec(E,B)
Ack(A,B,C) ← dec(A,D), dec(B,E), Ack(A,E,F), Ack(D,F,C)
```

In this case, the ordering of literals found by FOIL is

$$
\begin{aligned}
&\mathsf{Ack}(W_1, W_2, W_3) < \mathsf{Ack}(V_1, V_2, V_3) \text{ if} \\
&\qquad W_1 < V_1, \text{ or} \\
&\qquad W_1 = V_1 \text{ and } W_2 < V_2.
\end{aligned}
$$

In the definition above, dec(A,D) gives D<A in the second and third clauses, and dec(B,E) in the third clause gives E<B, so all recursive literals in these clauses are less than the heads of the clauses. Consequently, this definition can be guaranteed to terminate when invoked with ground instances of A and B.

2.6 Improved Definitions

Programs like FOIL that depend on greedy search will occasionally follow unprofitable paths leading to poor definitions or no definitions at all. FOIL's backup mechanism is designed to ameliorate the latter condition by restarting search at saved backup points. The problem of poor definitions is much more difficult to circumvent.

From its earliest version, FOIL has incorporated post-processing of definitions in which unnecessary literals are excised from finished clauses and redundant clauses are removed from complete definitions. When there are numerous superfluous literals, clause pruning can consume a noticeable amount of time; a recent extension is a fast heuristic pruning method that reverts to the slow-but-sure algorithm in the event of failure.

The most recent versions have two additional mechanisms for producing better clauses. It sometimes happens that, when the possible literals to be added to a clause are being considered, one literal L would complete the clause but another literal of higher gain is selected instead. The search can meander along in this way, leading eventually to a clause that is inferior to the one that would have been produced if L had been chosen. FOIL now remembers the best complete clause that could have been obtained by a different choice of literal at any point. When the clause is complete, the system checks to see whether the remembered clause is at least as good as the final clause and, if so, uses the remembered clause instead. This extension, which requires hardly any additional computation, is responsible for much improved definitions in some tasks.

We have also observed cases in which a non-recursive literal L, chosen to complete a clause, involves only variables that appear in the left-hand side of the clause. Such a literal could clearly have appeared at the beginning of the right-hand side. If the right-hand side contains literals other than L, they may have had the effect of making the clause too specific. To circumvent this possibility, the clause is regrown starting with the single literal L on the right-hand side.

The final polishing involves reordering the clauses. After all clauses making up a definition have been sifted as above to remove redundancies, all non-recursive "base case" clauses are moved to the front so that they appear before any recursive clauses.

3. An Experiment

Many evaluations of learning systems involve a limited amount of background information – just that required for the task at hand – and sometimes carefully chosen training examples as well. Such experiments can demonstrate the feasibility of certain types of learning, but do not address the usefulness of the learning system in practical applications, where there is usually a large amount of irrelevant information and where training examples come from a neutral, unbiased source.

As a step towards a more pragmatic evaluation, we started with Ivan Bratko's well-known text *Prolog Programming for Artificial Intelligence* [Bratko, 1986]. Chapter 3 of this book introduces several programs for manipulating lists and includes a set of student exercises. We conducted trials to see whether FOIL could learn the expository programs and exercises in the same order as they appear in the book, omitting only the last two exercises that were quite different from the others. (One of them, canget, deals with lists specific to the monkey and bananas problem; the other, flatten, uses structured lists.) A brief summary of the problems attempted is:

member(E,L)	E is an element of list L
conc(L1,L2,L3)	appending L1 to L2 gives list L3
member1(E,L)	as for member with conc available
last(E,L)	E is the last element of L
last1(E,L)	ditto, but without using conc
del(E,L1,L2)	deleting an occurrence of E from L1 gives L2
member2(E,L)	as for member with del available
insert(E,L1,L2)	inserting E somewhere in L1 gives L2
sublist(L1,L2)	L1 is a sublist of L2
permutation(L1,L2)	L2 is a permutation of list L1
even/oddlength(L)	L has an even/odd number of elements (both relations to be defined)
reverse(L1,L2)	L2 is the reverse of list L1
palindrome(L)	list L is a palindrome
palindrome1(L)	as above, but not using reverse
shift(L1,L2)	rotating elements of L1 to the left gives L2
translate(L1,L2)	L2 is the results of translating L1 using an element-to-element mapping
subset(S1,S2)	S2 is a subset of set S1
dividelist(L1,L2,L3)	L2 contains the odd-numbered elements of L1, L3 contains the even-numbered elements of L1

We included the additional relation components(L,H,T), meaning list L has head H and tail T, that corresponds to Prolog's built-in [H|T] notation for lists. For each program, all relations encountered previously were available as background knowledge so that there were many irrelevant relations to confuse FOIL's search.

We also attempted to assemble training examples in an unbiased manner. The trials were repeated for two universes, defined as

- U3, the 40 lists containing up to three elements (where each element is in the set {1,2,3}); and
- U4, the 341 similar lists containing up to four elements from {1,2,3,4}.

In a trial, FOIL was given all $\oplus$ tuples over the relevant universe for each relation. In U3, for example, the 142 $\oplus$ tuples for conc include $\langle [], [13], [13] \rangle$ and $\langle [32], [2], [322] \rangle$ but not $\langle [322], [13], [32213] \rangle$ since, in the last case, one of the lists contains more than three elements. Two relations in the book are defined over restricted subclasses of lists, sets in the case of subset and lists without repetitions in the case of permutation. All other relations are defined over all lists. The $\ominus$ tuples for the relation being learned are generally the complement of the $\oplus$ tuples. However, for the second universe U4, some relations would then have an enormous number of such tuples – about $341^3 \approx 40$ million for conc – so we used the FOIL option that selects a random sample of $\ominus$ tuples to keep them down to about 90,000. The relations affected were conc and dividelist (where we used 0.2% of $\ominus$ tuples), del and insert (20%), translate (40%), and sublist, permutation, reverse and shift (80%).

FOIL was allowed 1500 seconds on a DECstation 5000/240 for each problem. As the book had not introduced negation at this stage, negated literals were barred from definitions. All FOIL's other options had their default values, including the default memory limit of 100,000 tuples on any training set.

The outcomes of this experiment are summarised in Table 3.1. In the *result* column, a $\surd$ means that a correct definition was obtained (often, but not always, the same as the program in the book). The notation *restricted* indicates that the definition was correct for the universe over which the examples were defined, but would give incorrect results for lists of arbitrary length. A common problem with the restricted definitions is an incorrect base case that relies on fortuitous properties of the limited domain. For instance, the definition of reverse found in universe U3 was

reverse(A,B) ← A=B, conc(A,C,D), sublist(A,C)
reverse(A,B) ← components(A,C,D), reverse(D,E), conc(F,D,A), conc(E,F,B)

The second (recursive) clause is correct. However, the odd-looking base case exploits the fact that all lists in U3 have length at most 3; if A is a sublist of C and the result of conc'ing A to C has length at most 3, this ensures that A has length 0 or 1. Of course, the first clause is correct for such short lists A.

Task		Tuples ⊕	Tuples ⊖	Result	Time (secs)
member	U3	75	45	√	0.1
	U4	880	484	√	0.9
conc	U3	142	63,858	√	28
	U4	1593	79,300	√	34
member1	U3	75	45	√	1.7
	U4	880	484	√	1.7
last	U3	39	81	restricted	0.2
	U4	340	1024	√	2.7
last1	U3	39	81	√	0.1
	U4	340	1024	√	1.9
del	U3	81	4719	√	422
	U4	1024	92,640	time limit	> 1500
insert	U3	81	4719	√	2.1
	U4	1024	92,640	√	56
member2	U3	75	45	√	0.1
	U4	880	484	√	0.9
sublist	U3	202	1398	√	1.8
	U4	2913	90,697	√	94
permutation	U3	52	204	√	1.6
	U4	749	3476	√	337
even/oddlength	U3	10/30	30/10	unsound mutual recursion	0.1
	U4	273/68	68/273	unsound mutual recursion	63
reverse	U3	40	1560	restricted	9.3
	U4	341	92,796	restricted	220
palindrome	U3	16	24	√	0.1
	U4	41	300	√	0.9
palindrome1	U3	16	24	restricted	928
	U4	41	300	restricted	212
shift	U3	39	1561	√	4.2
	U4	340	92,787	√	253
translate	U3	40	3120	time limit	> 1500
	U4	341	92,573	time limit	> 1500
subset	U3	27	37	restricted	0.2
	U4	81	175	restricted	19
dividelist	U3	40	63,960	restricted	182
	U4	341	79,302	erroneous	901

Table 3.1: results on learning programs

One definition produced by FOIL, dividelist in universe U4, was actually in error, even when only lists in the restricted universe are considered. FOIL relies on $\ominus$ tuples to show up over-generalisations. For this task, the training set included only 0.2% of the $\ominus$ tuples, none of which happened to reveal that the clause was defective. This underlines the heuristic nature of any learning from incomplete information.

Apart from running out of time, the other problem occurred in the task that required definitions of both evenlength and oddlength. The definitions found for U3 were

> evenlength(A) ← del(B,C,A), oddlength(C)
> oddlength(A) ← components(A,B,C), evenlength(C).

Each definition is correct in itself but, together, they lead to recursive looping since C is longer than A in the definition of evenlength but shorter in oddlength. This highlights the fine print in FOIL's guarantee of recursive soundness; an individual definition will not lead to problems, but two definitions invoking each other might.

4. Discussion

The results of this experiment can only be described as mixed. It is encouraging to see that FOIL can find correct definitions for many of the small programs, but less encouraging when we remember that students are expected to be able to produce all of them as a matter of course.

In particular, the fact that later definitions tend to be restricted (if they are found at all) highlights FOIL's sensitivity to irrelevant information. For example, when all the superfluous relations were removed, a correct definition of subset

> subset(A,B) ← B=[]
> subset(A,B) ← components(A,C,D), components(B,C,E), subset(D,E)
> subset(A,B) ← components(A,C,D), subset(D,B)

was found from U4 in only 0.5 seconds.

Another cause for concern is that recursive definitions require near-complete sets of $\oplus$ tuples. If we consider the simplest task, member in universe U3, it is interesting to observe the effect of deleting a single $\oplus$ tuple without changing the $\ominus$ tuples (corresponding to an item of missing information, but no mis-information). If the tuple is of the form $\langle X, Y \rangle$ where X is an element and Y is a list, then:

- There is no effect if Y is of length 3.
- If Y is of length 1 or 2, at least one recursive continuation is affected. FOIL still finds a correct definition but adds an extra clause to cover the apparent "special case".

When 25% of the $\oplus$ tuples were deleted at random, the resulting definition was still "correct" but contained three superfluous clauses.

The tasks in this experiment have the property that each can be defined by a Horn clause program without the use of negated literals. Even when negated literals are allowed, the definition language used by FOIL is too weak to capture some ideas. As an illustration, the first-order expression

$$(\forall x \ \mathrm{likes}(x, y)) \ \supset \ \mathrm{happy}(y)$$

cannot be written as a Prolog definition without the use of a cut or the establishment of an ancillary concept. Similarly, a program to recognise sentences of the language $a^* b^* c^*$ requires an extra concept such as sequence-of(Seq,Elt); a Prolog programmer would see this immediately and define the subsidiary predicate. FOIL cannot invent new relations of this kind, and can only apply negation to individual literals. Consequently, there are some quite simple concepts for which FOIL cannot find general definitions, no matter how many examples it is given.

5. Conclusion

As the title of this paper suggests, FOIL is still under development. In its current form it is an experimental vehicle for exploring ideas in learning, not a practical tool for constructing substantial logic programs. In the same way, ID3 circa 1978 was an experimental program that required a lot more work before a practical tool, C4.5, was obtained.

Several shortcomings of the system were mentioned in the previous section. Generalising slightly, we can identify the following features that will be required by any robust system for learning recursive logic programs:

- *Construction of new predicates:* Logic programmers make frequent use of predicates that do not appear in the problem statement. This is sometimes required to express the program in Horn clause form, but more frequently because ancillary predicates make the program simpler and more efficient. FOIL has no facilities for inventing new predicates, but the promising research of Muggleton and Buntine [1988], Kietz and Morik [1993] and others suggests that such facilities may be able to be grafted on.

- *Strategy for constructing programs:* Human logic programmers are taught to get the simplest base case first, then to develop the general recursive case. This kind of strategic approach is missing from FOIL, which just attempts to bite off as many $\oplus$ tuples as possible in each clause. This super-greedy strategy can lead to problems of the kind illustrated by the reverse example. Instead of the simple base case

  reverse(A,B) ← A=[], B=[]

  FOIL greedily tries to extend this to include single-element lists, leading to the restricted definition of section 3.

- *Selective use of relations:* At the moment, any learning task can be made harder for FOIL simply by including more and more irrelevant relations, thereby increasing the number of literals that must be examined at each step. We hypothesise that any practical system for learning logic programs must employ a characterisation of each remembered relation, so that a relation is only considered when there is a prior reason to believe that it may be of use.

- *Incomplete training sets:* It seems unlikely that near-complete sets of $\oplus$ tuples will be available when constructing recursive definitions for relations in the context of real-world problems. Practical training sets will be small and, in problems involving synthesis of a novel theory, the given tuples will not be helpfully selected with the form of the final definition in mind. While FOIL can currently learn non-recursive definitions from sparse training cases, it has difficulty with recursive theories under these conditions.

- *Extended treatment of numeric fields:* Not many first-order systems seem to have addressed the issue of using continuous-valued information. FOIL's use of numeric fields is limited to thresholding and comparisons of known values rather than computing new values. Since many practical Prolog programs involve computation, learning systems that are intended to generate these programs must somehow come to grips with computational clauses.

With the inclusion of theory constants and tests on numeric values, FOIL can now express any theory derivable by zeroth-order learning systems such as C4.5. We have carried out some initial tests running FOIL on zeroth-order attribute-value data in which there is a single relation with one argument for each attribute. Since FOIL explores a strictly larger hypothesis space than these systems, it is not surprising that FOIL is slower. It will be interesting to see whether the increased search results in more accurate theories than those learned by zeroth-order systems.

The current version of FOIL is always available by anonymous ftp from 129.78.8.1, file name pub/foilN.sh for some integer N.

Acknowledgements

This research was supported by a grant from the Australian Research Council and by a research agreement with Digital Equipment Corporation.

References

1. Bratko, I. (1986). *Prolog Programming for Artificial Intelligence.* Wokingham, UK: Addison-Wesley.

2. Breiman, L., Friedman, J.H., Olshen, R.A. and Stone, C.J. (1984). *Classification and Regression Trees.* Belmont: Wadsworth International.

3. Cameron-Jones, R.M., and Quinlan, J.R. (1993). Avoiding pitfalls when learning recursive theories (draft). Available by anonymous ftp from 129.78.8.1, file pub/recurse.tex.

4. Cestnik, B., Kononenko, I. and Bratko, I. (1987). ASSISTANT 86: a knowledge elicitation tool for sophisticated users. In Bratko and Lavrač (Eds.) *Progress in Machine Learning.* Wilmslow: Sigma Press.

5. Clark, P and Niblett, T. (1987). Induction in noisy domains. In Bratko and Lavrač (Eds.) *Progress in Machine Learning.* Wilmslow: Sigma Press.

6. Kietz, J. and Morik, K. (1993). A polynomial approach to the constructive induction of structural knowledge. *Machine Learning*, to appear.

7. Kijsirikul, B., Numao, M. and Shimura, M. (1991). Efficient learning of logic programs with non-determinate, non-discriminating literals. *Proceedings Eighth International Workshop on Machine Learning*, Evanston, Illinois, 417-421.

8. Michalski, R.S., Mozetič, I., Hong, J. and Lavrač, N. (1986). The multipurpose incremental learning system AQ15 and its testing application to three medical domains. *Proceedings Fifth National Conference on Artificial Intelligence*, Philadelphia, 1041-1045.

9. Muggleton, S., and Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. *Proceedings Fifth International Conference Machine Learning*, Ann Arbor, 339-352.

10. Muggleton, S., and Feng, C. (1990). Efficient induction of logic programs. *Proceedings First Conference on Algorithmic Learning Theory*, Tokyo.

11. Pazzani, M.J., Brunk, C.A. and Silverstein, G. (1991). A knowledge-intensive approach to learning relational concepts. *Proceedings Eighth*

International Workshop on Machine Learning, Evanston, Illinois, 432-436.

12. Pazzani, M.J. and Kibler, D. (1992). The utility of knowledge in inductive learning. *Machine Learning 9*, 1, 57-94.

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

14. Quinlan, J.R. (1991). Determinate literals in inductive logic programming. *Proceedings Twelfth International Joint Conference on Artificial Intelligence*, Sydney, 746-750.

15. Quinlan, J.R. (1992). *C4.5: Programs for Machine Learning.* San Mateo: Morgan Kaufmann.

16. Richards, B.L. and Mooney, R.J. (1991). First-order theory revision. *Proceedings Eighth International Workshop on Machine Learning*, Evanston, Illinois, 447-451.

17. Rouveirol, C. (1991). Completeness for induction procedures. *Proceedings Eighth International Workshop on Machine Learning*, Evanston, Illinois, 452-456.

18. Sammut, C.A., and Banerji, R.B. (1986). Learning concepts by asking questions. In R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.) *Machine Learning: An Artificial Intelligence Approach* (Vol 2). Los Altos: Morgan Kaufmann.

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