Avoiding Pitfalls When Learning Recursive Theories*

R. M. Cameron-Jones and **J. R. Quinlan**
Basser Department of Computer Science
University of Sydney
Sydney Australia 2006
email: mcj@cs.su.oz.au quinlan@cs.su.oz.au

Abstract

Learning systems that express theories in first-order logic must ensure that the theories are executable and, in particular, that they do not lead to infinite recursion. This paper presents a heuristic method for preventing infinite recursion in the (multi-clause) definition of a recursive relation. The method has been implemented in the latest version of FOIL, but could also be used with any learning method that grows clauses from ground facts by repeated specialization. Results on several examples, including Ackermann's function, are presented.

1 Introduction

A growing cohort of systems now learn first-order theories to explain observations. Early examples, MIS [Shapiro, 1983] and MARVIN [Sammut and Banerji, 1986], have been joined by CIGOL [Muggleton and Buntine, 1988], FOIL [Quinlan, 1990, 1991], GOLEM [Muggleton and Feng, 1990], FOCL [Pazzani, Brunk and Silverstein, 1991; Pazzani and Kibler, 1992], ILE [Rouveirol, 1991] and others. The reason for looking beyond familiar propositional formalisms is their limited expressiveness – observations are not always capable of being represented as attribute values and theories sometimes cannot be captured as decision trees or DNF expressions.

Most first-order learning systems construct Horn clause theories. This subset of first-order logic is rich enough to include variablization, limited quantification, and recursion, yet restricted enough to be computable (hence its use as the basis for the well-known programming language Prolog). Learned theories expressed as collections of Horn clauses are therefore executable programs, explaining why this sub-field of machine learning is also known as inductive logic programming.

Any theory that contains recursive clauses runs the risk of non-termination due to infinite recursion. There is no general method for detecting such flawed recursive theories, since deciding whether a Horn clause program will terminate is an analog of the halting problem and thus undecidable. However, this paper describes a heuristic method that can vet recursive theories while they are being developed, thereby avoiding some problematic recursion. The method guarantees that, for a ground query, the clauses defining a single relation will never invoke themselves so as to cause infinite recursion.

The idea starts with the discovery of orderings on sets of constants that are then extended to orderings of the recursive literals in clauses. The approach, which has been implemented in the new version of FOIL, should also be applicable to any system that learns from ground examples by a process of specialization.

In the following sections we introduce the learning task and give a brief description of FOIL. We then show how sets of constants can be ordered and how this ordering can be used to discover orderings of pairs of variables in a clause, leading to a method for ordering recursive literals in a set of clauses. The paper concludes with empirical results that support the utility of the approach.

2 The Learning Task

Input consists of a collection of *relations*. Each relation is specified by two sets of tuples of constants: those that belong in the relation ($\oplus$ tuples) and those that do not ($\ominus$), where the latter are often taken to be the complement of the former by the closed-world assumption. The constants may be grouped into (possibly overlapping) types or may belong to a single universal type. A relation R of arity k, represented by k-tuples of constants, may also be regarded as a literal $R(V_1, V_2, ..., V_k)$ with k arguments.

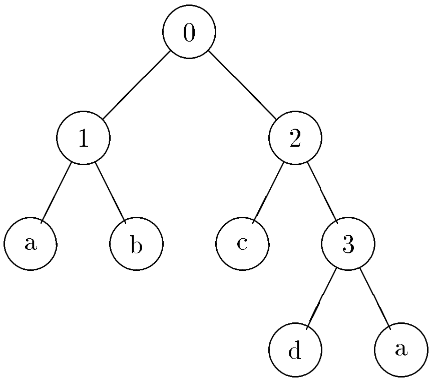
The learning task is to use the extensional definition of the relations provided by the tuples to generate an intensional definition of a designated *target* relation in terms of itself and other relations.

We will use the small example of Figure 1 as a continuing illustration. There are two relations:

$subtree(A, B, C)$	tree A has subtrees B and C
$leaf\text{-}of(A, B)$	A is a leaf of non-leaf tree B

The tuples shown for these relations are the $\oplus$ tuples from the tree in the Figure, the $\ominus$ tuples being the complementary sets. For instance, $\langle 0, 1, 2 \rangle$ is in the relation *subtree* because the tree at node 0 has the subtrees at nodes 1 and 2; $\langle a, 2 \rangle$ is in *leaf-of* because

*This research was made possible by a grant from the Australian Research Council (principal investigator J.R. Quinlan) and assisted by a research agreement with Digital Equipment Corporation.



subtree:	*leaf-of:*
$\langle 0,1,2 \rangle$	$\langle a,0 \rangle$
$\langle 1,a,b \rangle$	$\langle a,1 \rangle$
$\langle 2,c,3 \rangle$	$\langle a,2 \rangle$
$\langle 3,d,a \rangle$	$\langle a,3 \rangle$
	$\langle b,0 \rangle$
	$\langle b,1 \rangle$
	$\langle c,0 \rangle$
	$\langle c,2 \rangle$
	$\langle d,0 \rangle$
	$\langle d,2 \rangle$
	$\langle d,3 \rangle$

Figure 1: Relations *subtree* and *leaf-of*

$\langle a,0 \rangle \oplus$	$\langle 0,0 \rangle \ominus$	$\langle 1,3 \rangle \ominus$	$\langle 2,c \rangle \ominus$	$\langle a,b \rangle \ominus$
$\langle c,a \rangle \ominus$	$\langle a,1 \rangle \oplus$	$\langle 0,1 \rangle \ominus$	$\langle 1,a \rangle \ominus$	$\langle 2,d \rangle \ominus$
$\langle a,c \rangle \ominus$	$\langle c,b \rangle \ominus$	$\langle a,2 \rangle \oplus$	$\langle 0,2 \rangle \ominus$	$\langle 1,b \rangle \ominus$
$\langle 3,0 \rangle \ominus$	$\langle a,d \rangle \ominus$	$\langle c,c \rangle \ominus$	$\langle a,3 \rangle \oplus$	$\langle 0,3 \rangle \ominus$
$\langle 1,c \rangle \ominus$	$\langle 3,1 \rangle \ominus$	$\langle b,2 \rangle \ominus$	$\langle c,d \rangle \ominus$	$\langle b,0 \rangle \oplus$
$\langle 0,a \rangle \ominus$	$\langle 1,d \rangle \ominus$	$\langle 3,2 \rangle \ominus$	$\langle b,3 \rangle \ominus$	$\langle d,1 \rangle \ominus$
$\langle b,1 \rangle \oplus$	$\langle 0,b \rangle \ominus$	$\langle 2,0 \rangle \ominus$	$\langle 3,3 \rangle \ominus$	$\langle b,a \rangle \ominus$
$\langle d,a \rangle \ominus$	$\langle c,0 \rangle \oplus$	$\langle 0,c \rangle \ominus$	$\langle 2,1 \rangle \ominus$	$\langle 3,a \rangle \ominus$
$\langle b,b \rangle \ominus$	$\langle d,b \rangle \ominus$	$\langle c,2 \rangle \oplus$	$\langle 0,d \rangle \ominus$	$\langle 2,2 \rangle \ominus$
$\langle 3,b \rangle \ominus$	$\langle b,c \rangle \ominus$	$\langle d,c \rangle \ominus$	$\langle d,0 \rangle \oplus$	$\langle 1,0 \rangle \ominus$
$\langle 2,3 \rangle \ominus$	$\langle 3,c \rangle \ominus$	$\langle b,d \rangle \ominus$	$\langle d,d \rangle \ominus$	$\langle d,2 \rangle \oplus$
$\langle 1,1 \rangle \ominus$	$\langle 2,a \rangle \ominus$	$\langle 3,d \rangle \ominus$	$\langle c,1 \rangle \ominus$	$\langle d,3 \rangle \oplus$
$\langle 1,2 \rangle \ominus$	$\langle 2,b \rangle \ominus$	$\langle a,a \rangle \ominus$	$\langle c,3 \rangle \ominus$	

$\langle a,0,1,2 \rangle \oplus$	$\langle d,0,1,2 \rangle \oplus$	$\langle 1,1,a,b \rangle \ominus$
$\langle 3,1,a,b \rangle \ominus$	$\langle a,1,a,b \rangle \oplus$	$\langle d,2,c,3 \rangle \oplus$
$\langle 1,2,c,3 \rangle \ominus$	$\langle 3,2,c,3 \rangle \ominus$	$\langle a,2,c,3 \rangle \oplus$
$\langle d,3,d,a \rangle \oplus$	$\langle 1,3,d,a \rangle \ominus$	$\langle 3,3,d,a \rangle \ominus$
$\langle a,3,d,a \rangle \oplus$	$\langle 0,0,1,2 \rangle \ominus$	$\langle 2,0,1,2 \rangle \ominus$
$\langle b,2,c,3 \rangle \ominus$	$\langle b,0,1,2 \rangle \oplus$	$\langle 0,1,a,b \rangle \ominus$
$\langle 2,1,a,b \rangle \ominus$	$\langle b,3,d,a \rangle \ominus$	$\langle b,1,a,b \rangle \oplus$
$\langle 0,2,c,3 \rangle \ominus$	$\langle 2,2,c,3 \rangle \ominus$	$\langle c,1,a,b \rangle \ominus$
$\langle c,0,1,2 \rangle \oplus$	$\langle 0,3,d,a \rangle \ominus$	$\langle 2,3,d,a \rangle \ominus$
$\langle c,3,d,a \rangle \ominus$	$\langle c,2,c,3 \rangle \oplus$	$\langle 1,0,1,2 \rangle \ominus$
$\langle 3,0,1,2 \rangle \ominus$	$\langle d,1,a,b \rangle \ominus$	

Figure 2: Training set (a) initially, and (b) after adding *subtree*(B,C,D)

a is a leaf of the tree at node 2. If the target relation is *leaf-of*, the learning task would be to find a definition[1] such as

$$\begin{array}{lcl} \textit{leaf-of}(A,B) & \leftarrow & \textit{subtree}(B,A,C),\ \neg\textit{subtree}(A,D,E) \\ \textit{leaf-of}(A,B) & \leftarrow & \textit{subtree}(B,C,A),\ \neg\textit{subtree}(A,D,E) \\ \textit{leaf-of}(A,B) & \leftarrow & \textit{subtree}(B,C,D),\ \textit{leaf-of}(A,C) \\ \textit{leaf-of}(A,B) & \leftarrow & \textit{subtree}(B,C,D),\ \textit{leaf-of}(A,D) \end{array}$$

3 FOIL

FOIL learns function-free definitions of this kind. A complete description of the system is given in [Quinlan, 1990, 1991], but the following simplified version is sufficient for our purposes.

FOIL grows a definition one clause at a time until all $\oplus$ tuples in the target relation have been covered by at least one clause. Each clause commences with the same left-hand side and the right-hand side is successively specialized by the addition of literals until the clause covers no $\ominus$ tuples. Clause development is guided by a *training set* of tuples of constants, each tuple representing a possible binding of the variables in the incomplete clause that satisfies all literals on the right-hand side. The initial training set for each clause consists of the uncovered $\oplus$ tuples of the target relation together with all $\ominus$ tuples. Adding a literal to the right-hand side can cause some tuples to be excluded and/or all remaining tuples to be extended by the introduction of new variables.

[1] Some literals in this definition are negated; many systems, including FOIL, permit this small extension to the Horn clause formalism.

Using the above illustration, the training set for the first clause is as shown in Figure 2(a). If *subtree*(B,C,D) is added as the first literal on the right-hand side of the clause, the training set changes as shown in Figure 2(b). The number of tuples is reduced and each remaining tuple is expanded to take account of new variables C and D. The element $\langle a,0,1,2 \rangle$ shows a possible binding $A{=}a$, $B{=}0$, $C{=}1$, $D{=}2$ that satisfies the right-hand side; it is marked $\oplus$ since $\langle a,0 \rangle$ belongs in the target relation *leaf-of*.

When recursive literals are candidates for addition to the right-hand side of a clause, we need to ensure that they cannot cause infinite recursion on ground queries. For instance, if $R(A,B)$ is a symmetric relation, pseudo-definitions such as

$$R(A,B) \ \leftarrow\ R(B,A)$$

must be avoided. The original version of FOIL discovered possible orderings of the constants, as described below, that in turn allowed the establishment of a partial ordering of the variables in a clause. Literal $R(W_1,W_2,...,W_k)$ was considered for addition to a clause with left-hand side $R(V_1,V_2,...,V_k)$ only if $W_i < V_i$ for at least one argument V_i. This is sufficient to prevent problems when there is a single recursive literal in a single clause, but not when there are two or more of either. For example, a recurrence relation of the form

$$R(A,B) = R(A-1,B+1) + R(A+1,B-1)$$

leads to infinite recursion, yet each recursive literal satisfies the ordering requirement above.

We now amend the sufficiency requirement to allow for an arbitrary collection of clauses for the target literal, each of which may contain any number of recursive literals. We want to ensure that a ground query of the target literal R cannot cause the clauses defining R to invoke themselves without termination. This still does not cover infinite recursion arising from the situation in which clauses for a relation R invoke clauses for a relation S which in turn invoke clauses for R.

4 Ordering a Set of Constants

Constants belonging to some common types such as the integers have a well-known natural order. Since the method described here depends on an ordering of constants in all types, including symbolic types, we require a general algorithm for discovering a plausible ordering of the constants belonging to each type. To simplify the presentation we will assume that all constants belong to a single universal type; if there are several types, the procedure is repeated for each in turn.

The assumption underlying the search for an ordering is that some pairs of arguments of some relations must be ordered – this would seem to be a prerequisite for the development of well-behaved recursive theories. For each relation $R(V_1, V_2, ..., V_k)$, every pair of arguments V_i, V_j $(i \neq j)$ is examined to see whether it is possible that $V_i < V_j$. If the relation is specified by $n \oplus k$-tuples of constants, there are pairs of constants $a_{xi}, a_{xj},\ x = 1..n$ corresponding to the pairs of arguments V_i, V_j. If the relation establishes that $V_i < V_j$, then it must follow that $a_{xi} < a_{xj}$ for each $x = 1..n$. This is feasible if and only if the transitive closure of these n inequalities does not contain any cycles, i.e. there is no way to establish that any constant is less than itself. If the closure is cycle-free, we postulate that either $V_i < V_j$ or $V_i > V_j$, since both are equally plausible and we cannot distinguish between them.

Returning to Figure 1, we see that a potential inequality exists between all pairs of arguments of each relation. For example, the first pair of arguments of *subtree* would give the inequalities

$$0 < 1,\ 1 < a,\ 2 < c,\ 3 < d$$

whose transitive closure adds just $0 < a$. Since this is cycle-free, it is plausible that $subtree(A, B, C)$ establishes $A < B$.

Two partial orderings established by different argument pairs from the same or different relations are consistent if the transitive closure of all associated inequalities contains no cycles. From the example above, the first and third arguments of *subtree* give the inequalities

$$0 < 2,\ 1 < b,\ 2 < 3,\ 3 < a$$

and the transitive closure of these and the previous inequalities is still cycle-free. Thus there is an ordering of the constants that would allow $subtree(A, B, C)$ to establish $A < B$ and $A < C$.

When all orderings of argument pairs have been investigated, the next step is to find an ordering of the constants that is consistent with as many of them as possible. FOIL searches for a maximal consistent subset of the argument pairs and extracts from it a total ordering on the constants. The corresponding transitive closure of the inequalities between pairs of constants allows us to determine, for any constant a, the number of constants b such that $a < b$; the constants are then ranked on this number, with ties resolved arbitrarily. In our example, a maximal consistent set is

$$\begin{array}{ll} subtree(A, B, C): & A < B,\ A < C,\ B < C \\ leaf\text{-}of(A, B): & B < A. \end{array}$$

There is only one ordering of the constants consistent with the transitive closure of these inequalities, namely

$$0 < 1 < 2 < c < 3 < d < a < b.$$

Note also that the polarity of this ordering is arbitrary – we might just as well write $b < a < ...$ since we cannot distinguish between ascending and descending order.

5 Ordering Recursive Literals

The ordering of constants in each type now allows us to establish inequalities on the variables in a partly-developed clause and eventually an ordering on the recursive literals themselves.

Consider an incomplete clause whose left-hand side is $R(V_1, V_2, ..., V_k)$ and let the variables in the clause be $V_1, V_2, ..., V_v$ $(v \geq k)$. The corresponding training set consists of m v-tuples of constants $\langle a_{x1}, a_{x2}, ..., a_{xv} \rangle$, $x = 1..m$. For every pair of variables V_i, V_j of the same type, the ordering of constants can be used to establish whether $V_i < V_j$, i.e. whether $a_{xi} < a_{xj}$ for all x. The training set for the incomplete clause

$$leaf\text{-}of(A, B) \leftarrow subtree(B, C, D)$$

appears in Figure 2(b) above. With respect to the chosen order of the constants, $B < C$ (since $0 < 1$, $1 < a$, $2 < c$, and so on) and a similar check of the tuples reveals that $B < D$.

Suppose that the recursive literal $R(W_1, W_2, ..., W_k)$ is a candidate for addition to the right-hand side of the clause. If it is added, we must ensure that this invocation of R can never lead to R being called with the same arguments, causing an infinite recursive loop. Such problems can never arise if there is an ordering of the literals $R(...)$ such that recursive literals on the right-hand sides of clauses are always less than the literal on the left-hand side.

To this end, we define a total ordering on literals involving the target relation R as follows. We select a permutation $p1, p2, ..., pk$ of the variable positions $1, 2, .., k$ in such literals. For each pi we choose one of the inequality relations $<$ or $>$ and denote this choice $<>_{pi}$. Then,

$R(W_1, W_2, ..., W_k) < R(V_1, V_2, ..., V_k)$ if and only if
$W_{p1} <>_{p1} V_{p1}$, or
$W_{p1} = V_{p1} \wedge W_{p2} <>_{p2} V_{p2}$, or
$W_{p1} = V_{p1} \wedge W_{p2} = V_{p2} \wedge W_{p3} <>_{p3} V_{p3}$, or
...

It may seem strange that the inequality $>$ is permitted as a possible $<>_{pi}$, but this must be allowed because the order of the constants could be either ascending or descending.

The literal $R(W_1, W_2, ..., W_k)$ can be added to the right-hand side of the incomplete clause only if we can find an ordering of the literals, as above, in which this and any previous recursive literals are all less than the (unchanging) left-hand side of the clause. "Previous" here means occurring either earlier in the right-hand side of this clause, or in an earlier clause of this definition. Whether or not a suitable literal ordering exists can be established by a simple elimination process that is linear in the number of recursive literals in the definition so far.

In the example, the literal *leaf-of*(A, D) is an acceptable addition to the clause because we can choose the permutation 2, 1 of the relation arguments with $<>_2 = <>_1 = >$. This defines an ordering of the literals that can be paraphrased as

leaf-of(X, Y) is less than *leaf-of*(A, B) iff
  $Y > B$, or
  $Y = B$ and $X > A$

Since we have $D > B$ from above, *leaf-of*(A, D) is less than *leaf-of*(A, B) under this ordering of the literals.

6 Examples

FOIL generated the definition of *leaf-of* shown in Section 2 in less than 0.1 seconds. (All times quoted in this section are for a DECstation 5000.) This small example is not sufficient of itself to convince sceptical readers that the approach works, so we present results on a variety of more challenging tasks.

6.1 Ackermann's function

This is a more demanding test of handling recursive definitions due to the complexity of the recursion involved. Ackermann's function of two non-negative arguments is

$$F(m, n) = \begin{cases} n+1 & \text{if } m = 0 \\ F(m-1, 1) & \text{if } n = 0 \\ F(m-1, F(m, n-1)) & \text{otherwise} \end{cases}$$

Following the usual procedure of converting functions to function-free predicates, we substituted a three-argument predicate $Ack(A, B, C)$ for the two-argument function $F(m, n)$; $Ack(A, B, C)$ is true if $C = F(A, B)$. All values of the function up to 20 were given to FOIL, with $\ominus$ tuples coming from the closed-world assumption – this gave 51 $\oplus$ and 9,210 $\ominus$ tuples. The relation $succ(A, B)$, meaning $B = A + 1$, was also defined for values of B up to 20. The clauses found by FOIL in 95 seconds mirror the definition above:

$$\begin{aligned} Ack(A,B,C) &\leftarrow A{=}0,\ succ(B,C) \\ Ack(A,B,C) &\leftarrow B{=}0,\ succ(D,A), \\ &\qquad Ack(D,E,C),\ succ(B,E) \\ Ack(A,B,C) &\leftarrow succ(D,A),\ succ(E,B), \\ &\qquad Ack(A,E,F),\ Ack(D,F,C) \end{aligned}$$

The literal ordering found for this example was

$Ack(X, Y, Z)$ is less than $Ack(A, B, C)$ iff
  $X < A$, or
  $X = A$ and $Y < B$

	Tuples $\oplus$	Tuples $\ominus$	Clauses	Time (secs)
quicksort	16	240	2	2.8
append	261	8730	2	16.7
reverse	65	4160	2	10.3

Table 1: Results on list-processing tasks

(the third disjunct never being required). The first recursive literal in the last clause satisfies the latter disjunct while the second recursive literal satisfies the former disjunct. This example shows that control of recursion can require a sophisticated ordering on the recursive literals.

6.2 List processing examples

Several studies involving small list-processing tasks have been reported in the inductive programming literature. Three of the more difficult are:

- *Discovering the quicksort procedure*
  There are six relations:

sort(U, S)	sorting list U gives list S
partition(V, U, L, H)	partitioning list U on value V gives sublist L (values $\leq V$), sublist H (values $> V$)
components(L, H, T)	list L has head H and tail T
append(A, B, C)	appending list A to list B gives list C
null(L)	L is the empty list
elt(V)	V is a number

  Following Muggleton, the examples provided for the *sort* relation cover all lists of length up to three containing non-repeated numbers in $\{0, 1, 2\}$. There are 16 such lists, giving 16 $\oplus$ and 240 $\ominus$ tuples. The other relations are defined over this same vocabulary.

- *Finding how to append one list to another*
  Examples covered all lists of length up to three with non-repeating elements drawn from $\{1, 2, 3, 4\}$; since the closed-world assumption would give about 270,000 $\ominus$ tuples, a sample of about 9K of them was provided explicitly. Other relations were *components, null, list* and *reverse* – note that the last two of these were not required for the definition of *append*.

- *Finding how to reverse a list*
  Examples were the same as for the *append* relation above.

Results with these tasks, summarised in Table 1, show that FOIL handles them relatively well.

A more detailed examination of the *quicksort* task serves to illustrate the approach. Constants belonged to two types, numbers and lists. No orderings of the numbers were found, since none of the relations has two arguments of type number – from FOIL's perspective, then, any ordering is satisfactory. The ordering

	Tuples $\oplus$	Tuples $\ominus$	Clauses	Time (secs)
choose	21	189	2	4.5
P	27	1304	3	9.7

Table 2: Results on arithmetic recurrence relations

discovered for the list type started with the empty list, followed by all lists of length one, two, and then three, in agreement with intuition. The definition found,

$$\begin{array}{lcl} sort(A,B) & \leftarrow & A{=}B,\ null(A) \\ sort(A,B) & \leftarrow & components(A,C,D), \\ & & partition(C,D,E,F), \\ & & sort(E,G),\ sort(F,H), \\ & & append(G,I,B),\ components(I,C,H) \end{array}$$

has a clause with two recursive literals; the induced literal ordering, requiring the first argument of any recursive literals to be less than that of the left-hand side, is clearly satisfied in both cases.

6.3 Examples from arithmetic

The final examples concern arithmetic recurrence relations:

- *Finding a recurrence relation for* $\binom{n}{m}$
  The relations used in this task were

  $choose(M,N,X)$ (which means $X = \binom{N}{M}$),
  $multiply(A,B,C)$ ($A \times B = C$), and
  $dec(A,B)$ ($B = A - 1$).

  Tuples for values of N up to 5 and for the corresponding values for *multiply* and *dec* were provided.

- *An artificial relation*
  This recurrence relation was specifically designed to test the approach. The underlying function is

  $$G(m,n) = \begin{cases} \text{n} & \text{if } m = 0 \\ \text{m} & \text{if } n = 0 \\ \text{G(m,n-1) + G(m-1,n)} & \text{otherwise} \end{cases}$$

  The relations for this task were $P(A,B,C)$ (meaning $G(A,B) = C$), $plus(A,B,C)$ and $dec(A,B)$ as above. All tuples involving integer values up to 10 were provided.

The results on these tasks are summarised in Table 2.

In the second task, the constants are just the integers 1,2,...,10 and FOIL discovered their natural order. The definition found was not exactly the expected one:

$$\begin{array}{lcl} P(A,B,C) & \leftarrow & A{=}0,\ B{=}C \\ P(A,B,C) & \leftarrow & dec(A,D),\ plus(B,B,E), \\ & & plus(B,C,F),\ dec(F,G), \\ & & plus(B,E,H),\ P(D,E,G), \\ & & plus(F,H,I) \\ P(A,B,C) & \leftarrow & dec(A,D),\ dec(B,E),\ P(A,E,F), \\ & & P(D,B,G),\ plus(F,G,C) \end{array}$$

The first and third clauses are fine. The second is a convoluted generalisation of the $B{=}0$ case that is valid only for the restricted set of tuples provided – a reminder of the ever-present risk when learning from small numbers of examples. The literal ordering consistent with all three recursive literals is

$P(X,Y,Z)$ is less than $P(A,B,C)$ iff
$X < A$, or
$X = A$ and $Y < B$

and again there is no need for a third disjunct. The relation in this example is symmetric, showing that the approach prevents simple pseudo-definitions such as

$$P(A,B,C) \ \leftarrow\ P(B,A,C)$$

which, of course, would lead to infinite recursion.

7 A Note on Complexity

As the above examples illustrate, the computational cost of the approach is usually moderate. This brief section looks at worst-case rather than average complexity to see where problems might arise.

In the first phase (Section 4), the fundamental operation is that of checking the consistency of a potential constant ordering. The implementation uses an incremental transitive closure method whose worst-case cost is $O(c^4)$ for c constants. (Although other approaches have lower cost, the incremental method used by FOIL allows consistency to be checked as each constraint is added and has advantages when searching for the maximal consistent set of argument pair orderings.) For r relations of arity a, the number of possible ordered relation argument pairs is $O(ra^2)$. If p of these are ordered, the number of possible consistent sets of orders to be checked is $O(3^p)$. (Note that the reverse of any order found must also be considered, but that an order and its reverse are never mutually consistent).

The second phase (Section 5) occurs during clause development. If there are t tuples and v variables, the maximum cost of checking orderings of pairs of variables is $O(tv^2)$. For a target relation of arity a, there are $O((v + a - 1)^a)$ recursive literals that are potential candidates for addition to the right-hand side. If the definition to this point contains $l - 1$ recursive literals, the cost per literal checked is $O(la^2)$.

There are thus two exponential aspects of complexity: one with the number of pairs of argument orders that yield an ordering on the constants, and one with the arity of the target relation. The latter crops up in many places in FOIL, such as when evaluating the gain of potential literals. The cost of checking the ordering requirement for a recursive literal is usually small by comparison with the gain computation, so is not significant. The potential problem with the approach lies in putative situations in which there are many possible orders between pairs of arguments, of which about half are consistent. (The depth first search for a maximal consistent set halts once it becomes apparent that no further set can be larger than the best set found so far; if all pairs of arguments are consistent, little computation is required.) This circumstance has not yet been enountered in trials with a wide selection of problems from the literature. As an alternative to the present scheme, FOIL could use some heuristic method to find a near-maximal set of consistent orderings.

The approach described here decomposes the basic problem of finding an ordering of tuples of the target relation into finding orderings on pairs of their arguments. This allows us to work with smaller sets (c constants rather than $O(c^a)$ tuples) and to use information from other relations in addition to the target relation. Although it may be possible to construct problems for which the tuple ordering approach is the only viable one, the decomposition is computationally attractive and works well in practice.

8 Conclusion

This paper has presented an effective method for sidestepping many problematic recursive clauses while learning a first-order theory. The approach uses an ordering induced on constants of each type to develop an ordering of recursive literals so as to guarantee that a recursive literal on the right-hand side of a clause is always less than the literal on the left-hand side that caused it to be invoked.

Although the method has been implemented and tested as part of the new version of FOIL, it should be able to be incorporated into any learning system that develops clauses by repeated specialisation from a collection of ground facts. FOIL uses only constant atoms but a system that uses general ground terms could order them in a similar manner, either

(a) by treating terms as atoms, or

(b) by mapping all ground facts into function-free form, inducing an order on the atoms, then constructing from this an order relation on ground terms.

Again, if the order of constants or terms is known, or can be determined by some other means, the preparatory heuristic ordering of the constants can be omitted and the method for ordering recursive literals employed directly.

The process of finding an ordering on the constants consumes little time (a maximum of 0.02 seconds for the examples described in the previous section). Once these orders are established, checking whether one variable is either greater than or less than another variable over a training set requires only the comparison of the ranks of pairs of constants and so is also computationally undemanding.

Although the approach is sufficient to prevent infinite recursion arising from ground queries on a single relation, it does not address non-terminating recursion involving more than one relation, e.g. when relation R invokes S and S invokes R. Since FOIL builds definitions for a single relation at a time, preventing infinite recursion of this kind would require a much more complex scheme.

The new version of FOIL incorporating the approach described in this paper is available by anonymous ftp from beta.cs.su.oz.au under the file name /pub/foil4.sh
.

References

[Muggleton and Buntine, 1988] Stephen Muggleton and Wray Buntine. Machine invention of first-order predicates by inverting resolution. *Proceedings Fifth International Conference Machine Learning*, Ann Arbor, Michigan, 339-352. San Mateo, CA: Morgan Kaufmann.

[Muggleton and Feng, 1990] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. *Proceedings First Conference on Algorithmic Learning Theory*. Tokyo: Ohmsha.

[Pazzani *et al*, 1991] Michael J. Pazzani, Clifford A. Brunk, and Glenn Silverstein. A Knowledge-intensive approach to learning relational concepts. *Proceedings Eighth International Workshop on Machine Learning*, Evanston, Illinois, 432-436. San Mateo, CA: Morgan Kaufmann.

[Pazzani and Kibler, 1992] Michael J. Pazzani and Dennis Kibler. The utility of knowledge in inductive learning. *Machine Learning 9*, 1, 57-94.

[Quinlan, 1990] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning 5*, 3, 239-266.

[Quinlan, 1991] J. Ross Quinlan. Determinate literals in inductive logic programming. *Proceedings Twelfth International Joint Conference on Artificial Intelligence*, Sydney, 746-750. San Mateo, CA: Morgan Kaufmann.

[Rouveirol, 1991] Céline Rouveirol. Completeness for induction procedures. *Proceedings Eighth International Workshop on Machine Learning*, Evanston, Illinois, 452-456. San Mateo, CA: Morgan Kaufmann.

[Sammut and Banerji, 1986] Claude A. Sammut and Ranan B. Banerji. 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). San Mateo, CA: Morgan Kaufmann.

[Shapiro, 1983] Ehud Y. Shapiro. *Algorithmic Program Debugging*. Cambridge, MA: MIT Press.