Inverting Implication

Stephen Muggleton
Oxford University Computing Laboratory,
11 Keble Road,
Oxford OX1 3QD,
UK.

Abstract

All generalisations within logic involve inverting implication. Yet, ever since Plotkin's work in the early 1970's methods of generalising first-order clauses have involved inverting the clausal subsumption relationship. However, even Plotkin realised that this approach was incomplete. Since inversion of subsumption is central to many Inductive Logic Programming approaches, this form of incompleteness has been propagated to techniques such as Inverse Resolution and Relative Least General Generalisation. A more complete approach to inverting implication has been attempted with some success recently by Lapointe and Matwin. In the present paper the author derives general solutions to this problem from first principles. It is shown that clausal subsumption is only incomplete for self-recursive clauses. Avoiding this incompleteness involves algorithms which find "nth roots" of clauses. Completeness and correctness results are proved for a non-deterministic algorithms which constructs nth roots of clauses. It is shown how this algorithm can be used to invert implication in the presence of background knowledge. In conclusion the relationship between these results and Hoare's logical definition of programming from specifications is discussed.

1 Introduction

Plotkin [19] was the first to show that θ-subsumption and implication between clauses are not equivalent. The difference between the two is important since almost all inductive algorithms which generalise first-order clauses invert θ-subsumption rather than implication. This inevitably leads to a form of incompleteness in these algorithms. In this paper methods of constructing the inverse implicants of clauses are explored. In section 7 it is shown how methods developed in earlier sections can be extended to the problem of inverting implication in the presence

of background knowledge. First the difference between Plotkin's θ-subsumption and implication between clauses will be reviewed. The reader is referred to Appendix A for the usual definitions in Logic Programming and Inductive Logic Programming (ILP).

Clause C θ-subsumes clause D whenever there exists a substitution θ such that $C\theta \subseteq D$. Clause C implies clause D, or $C \rightarrow D$, whenever every model of C is a model of D. Whenever clause C θ-subsumes clause D it also implies D. However the converse does not hold. For instance Plotkin shows that with clauses

$$\begin{aligned} C &= p(f(X)) \leftarrow p(X) \\ D &= p(f(f(X))) \leftarrow p(X) \end{aligned}$$

C implies D, since D is simply C self-resolved. However C does not θ-subsume D. In discussing this problem Niblett [18] proves various general results. For instance he shows that implication between Horn clauses is decidable and also that there is not always a unique least generalisation under implication of an arbitrary pair of clauses. For instance, the clause D above and the clause $E = p(f(f(f(X)))) \leftarrow p(X)$ have both C and the clause $p(f(X)) \leftarrow p(Y)$ as least generalisations.

Gottlob [8] also proves a number of properties concerning implication between clauses. Notably let C^+, C^- be the positive and negative literals of C and D^+, D^- be the same for D. Now if $C \rightarrow D$ then C^+ θ-subsumes D^+ and C^- θ-subsumes D^-.

2 Sub-unification

The problem of inverting implication is discussed in a recent paper by Lapointe and Matwin [11]. They note that inverse resolution [15, 14, 22, 24] is incapable of reversing SLD derivations in which the hypothesised clause is used more than once. In fact Plotkin [19] showed that the same problem appears in the use of relative least general generalisation of clauses. Lapointe and Matwin go on to describe sub-unification, a process of matching sub-terms. They demonstrate that sub-unification is able to construct recursive clauses from fewer examples than would be required by ILP systems such as Golem [16] and FOIL [20]. For instance, given the atoms $append([], X, X)$ and $append([a, b, Y], [1, 2], [a, b, Y, 1, 2])$ sub-unification can be used to construct the recursive clause

$$append([U|V], W, [X|Y]) \leftarrow append(V, W, Y)$$

Unlike the approach taken originally with inverse resolution [15], Lapointe and Matwin do not derive sub-unification from resolution. Instead sub-unification is based on a definition of most general sub-unifiers. Although the operations

described by Lapointe and Matwin are shown to work on a number of examples it is not clear how general the mechanism is.

In this paper a general approach to inverting implication is developed. The approach taken involves a new form of inverting resolution which is derived from first principles.

3 Implication and resolution

In this section the relationship between resolution and implication between clauses is investigated. Below a definition equivalent to Robinson's [21] resolution closure is given. The function $\mathcal{L}$ below contains only the linear derivations of Robinson's function $\mathcal{R}$ (see Appendix A.3). However, the closure is equivalent up to renaming of variables given that linear derivation (as opposed to input derivation) is known to be complete.

Definition 1 (Resolution closure) *Let T be a set of clauses. The function $\mathcal{L}$ is recursively defined as*

$$\begin{aligned} \mathcal{L}^1(T) &= T \\ \mathcal{L}^n(T) &= \{C : C_1 \in \mathcal{L}^{n-1}(T), C_2 \in T, C \text{ is the resolvent of } C_1 \text{ and } C_2\} \end{aligned}$$

the resolution closure $\mathcal{L}^*(T)$ is $\mathcal{L}^1(T) \cup \mathcal{L}^2(T) \cup \ldots$

Lee [12] first proved the subsumption theorem, a reproof of which can be found in Bain and Muggleton [1]. The theorem can be stated as follows.

Theorem 2 (Subsumption theorem) *Let T be a set of clauses and C be a non-tautological clause. $T \models C$ if and only if there exists D in $\mathcal{L}^*(T)$ and substitution θ such that $D\theta \subseteq C$.*

In order to apply this to the case of implication between clauses Theorem 2 can be applied to the special case in which T is a single clause.

Corollary 3 (Implication between clauses using resolution) *Let C be an arbitrary clause and D be a non-tautological clause. $C \models D$, or $C \rightarrow D$, if and only if there exists a clause E in $\mathcal{L}^*(\{C\})$ and substitution θ such that $E\theta \subseteq D$.*
Proof. *Follows directly as a special case of theorem 2.*

Restating corollary 3, $C \rightarrow D$ whenever one of the following conditions holds

1. D is a tautology.
2. C θ-subsumes D.
3. E θ-subsumes D where E is constructed by repeatedly self-resolving C.

The first two conditions are somewhat trivial. The third condition demonstrates the significance of self-recursive clauses in this problem. It is clearly no coincidence that Plotkin's example (Section 1) and the clauses investigated by Lapointe and Matwin (Section 2) are self-recursive.

4 Nth powers and nth roots of clauses

The set of clauses constructed by self-recursing C, $\mathcal{L}^*(\{C\})$, is partitioned into levels by the function $\mathcal{L}$. By viewing resolution as a product operation Muggleton and Buntine [15] (see '$\cdot$' operator in A.3) stated the problem of finding the inverse resolvent of a pair of clauses as that of finding the set of quotients of two clauses. Following the same analogy the set $C^2 = \mathcal{L}^2(\{C\})$ might be called the squares of the clause C and $C^3 = \mathcal{L}^3(\{C\})$ the cubes of C. The following definition captures this idea.

Definition 4 (nth powers of a clause) *Let C and D be clauses. For $n \geq 1$, D is an nth power of C if and only if D is an alphabetic variant of a clause in $\mathcal{L}^n(\{C\})$.*

Taking the analogy a bit further one might also talk about the nth roots of a clause.

Definition 5 (nth roots of a clause) *Let C and D be clauses. D is an nth root of C if and only if C is an nth power of D.*

Corollary 3 can now be restated in terms of nth roots of a clause.

Corollary 6 (Implication between clauses in terms of nth roots) *Let C be an arbitrary clause and D be a non-tautological clause. $C \rightarrow D$ if and only if for some positive integer n, C is an nth root of a clause E which θ-subsumes D.*

It is fairly straightforward to enumerate the set of clauses which θ-subsume a given clause. Therefore the problem of finding the set of clauses which imply a given clause C reduces to that of enumerating the set of nth roots of clauses which θ-subsume C. The special case of clauses which immediately θ-subsume C occurs with $n = 1$.

5 Constructing the square roots of a clause

Before attempting the harder problem of constructing arbitrary nth roots of clauses let us consider the simpler problem of constructing the square roots of a clause. Figure 1 shows the self resolution of the clause C to give D, where $D \in C^2$. Assume that this resolution involves the complementary pair $\langle l\theta_1, \overline{l'}\theta_2 \rangle$ where l is a positive literal in C and $\overline{l'}$ is a negative literal in C and $\theta_1\theta_2$ is the most general unifier (mgu) of l and l'. From the definition of a complementary pair (Appendix A.1)

$$l\theta_1 = l'\theta_2$$

where the domains of θ_1 and θ_2 are subsets of $vars(l)$ and $vars(l')$ respectively. Clause C can be written as

$$l \leftarrow l' \wedge B \tag{1}$$

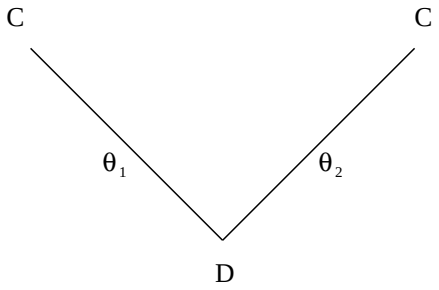

Figure 1: Squaring a clause

where B is a conjunction of literals. If C is a definite clause and l is an atom then B will be a conjunction of atoms. D, any self-resolvent of C, has the general form

$$l\theta_2 \leftarrow l'\theta_1 \wedge B\theta_1 \wedge B\theta_2 \tag{2}$$

The problem now is how to construct C (clause 1) from D (clause 2). To do so l, l' and B need to be reconstructed. A simple non-deterministic and incomplete first-cut approach to this problem would seem to be as follows.

Algorithm 1 (a simple, flawed square root algorithm)

1. *Choose from D a pair of literals $\langle l\theta_2, \overline{l'\theta_1}\rangle$ with the same predicate symbol.*
2. *Partition the clause $D - \{l\theta_2, \overline{l'\theta_1}\}$ into two equal cardinality conjuncts $B\theta_1$ and $B\theta_2$ which are both instances of a conjunct B.*
3. *Construct θ_1 and θ_2 by matching B to $B\theta_1$ and $B\theta_2$.*
4. *Invert the substitution θ_1 on $l'\theta_1$ and θ_2 on $l\theta_2$ to get l' and l respectively.*
5. *Return $l \leftarrow l' \wedge B$*

The following example, which uses Prolog-like notation, demonstrates Algorithm 1 at work.

Example 7 (Trace of Algorithm 1)
Let clause C be

$$lt(F,G) \leftarrow succ(F,H),\ lt(H,G)$$

and D be

$$lt(I,J) \leftarrow succ(I,K),\ succ(K,L),\ lt(L,J)$$

The steps in Algorithm 1 are followed below to reconstruct C from D.

1. *Let $l\theta_2 = lt(I,J)$ and $l'\theta_1 = lt(L,J)$.*

2. *Let* $B\theta_2 = succ(I, K)$ *and* $B\theta_1 = succ(K, L)$*, which are both instances of* $B = succ(M, N)$.

3. $\theta_1 = \{M/K, N/L\}$ *and* $\theta_2 = \{M/I, N/K\}$.

4. $l = lt(M, J)$ *and* $l' = lt(N, J)$.

5. *Return* $lt(M, J) \leftarrow lt(N, J), succ(M, N)$

which is an alphabetic variant of C.

Algorithm 1 has a number of shortcomings. Firstly, it is non-deterministic. This could be overcome by constructing all possible solutions and returning those which self-resolve to give alphabetic variants of C. Secondly, Example 7 demonstrates that the substitutions θ_1 and θ_2 constructed in step 3 of Algorithm 1 can be incomplete. In Example 7 neither θ_1 nor θ_2 contain a substitution for the variable J. Plotkin's clauses in Section 1 provide an extreme example of the incomplete construction of θ_1 and θ_2. In Plotkin's example B is empty and therefore step 3 will fail to extract θ_1 and θ_2. Thirdly, the inversion of the substitutions in step 4 is not straightforward. If the use of inverse substitutions described in [15] (defined also in Appendix A.2) is followed then there can be a multiplicity of possible inverse substitutions of a particular substitution θ applied to a given literal l. Many of these problems can be avoided by first flattening the clause D, constructing its square roots and then unflattening the results.

5.1 Flattening clauses

Rouveirol and Puget [23, 22] describe operations called *flattening* and *unflattening* to simplify inverse resolution. This form of operation is well-known within the literature of integrating logic programming and functional programming (see for instance [7, 4]). Rouveirol and Puget's flattening operation transforms clauses with function symbols into clauses in a function-free form. Unflattening a clause transforms it back to its original form. The approach taken in this paper to flattening clauses differs from Rouveirol and Puget in that only equality literals are introduced rather than introducing new predicates. For the purposes of finding the square roots of a clause, flattened clauses will be used with a particular goal in mind. The goal is to ensure that the *mgu* involved in self-resolving a clause is a special kind of substitution known as a *renaming*. Since renamings are easy to invert they help solving some of the problems with Algorithm 1.

5.2 Renaming

Lloyd [13] defines a renaming substitution, or a renaming for short, as follows (see Appendix A.1 for the definition of vars(F)).

Definition 8 (Renamings) *Let F be a well-formed formula and $\theta = \{u_1/v_1, .., u_n/v_n\}$ be a substitution. θ is a renaming of F if and only if $u_1, .., u_n$ are all distinct variables, $v_1, .., v_n$ are all distinct variables and $(vars(F) - \{u_1, .., u_n\}) \cap \{v_1, .., v_n\} = \emptyset$.*

Renamings are easy to invert because they are one-to-one mappings. The inverse of a renaming is defined as follows.

Definition 9 (Inverse renaming) *Let $\theta = \{u_1/v_1, .., u_n/v_n\}$ be a renaming of the formula F. θ^r, the inverse of θ for F, is the substitution $\{v_1/u_1, .., v_n/u_n\}$.*

The following theorems about renamings can now be shown.

Lemma 10 (Renaming composed with its inverse is identity function) *Let θ be a renaming of the well-formed formula F. $F\theta\theta^r = F$.*
Proof. *Each variable u in the domain of θ is mapped to a distinct variable v by θ. v is then mapped back to u using θ^r.*

Lemma 11 (Composition of renamings is a renaming) *Let θ be a renaming of the formula F and τ be a renaming of $F\theta$. The substitution $\sigma\tau$ is a renaming of F.*
Proof. *Let u be mapped to v in θ. If v is in the domain of τ and v is mapped to w in τ then u is mapped uniquely to w in $\sigma\tau$. Otherwise u is mapped uniquely to v in $\sigma\tau$.*

5.3 Flattening using equalities

As stated earlier the goal is to use flattening to ensure that the *mgu* involved in self-resolving a clause C is a renaming of C. This can be done by flattening the clause so that the only terms in the two recursing literals are sets of distinct variables.

Example 12 (Flattening, squaring and unflattening a clause)
Let clause C be

$$member(G, [H|I]) \leftarrow member(G, I)$$

C is flattened to to

$$member(G, J) \leftarrow member(G, I),\ J = [H|I]$$

in which the only terms in the two recursing literals are sets of distinct variables. Self-resolve C involves resolving it with $C\sigma$ where σ is a renaming of all the variables in C. Thus $C\sigma$ might be

$$member(G', J') \leftarrow member(G', I'),\ J' = [H'|I']$$

The head of C can be resolved away with the member atom in the body of $C\sigma$. The mgu involved is the renaming $\theta = \{G'/G, I'/J\}$ of C and the resolvent is

$$member(G, J') \leftarrow member(G, I),\ J' = [H'|J],\ J = [H|I]$$

Unflattening this clause gives

$$member(G, [H', H|I]) \leftarrow member(G, I)$$

which is the square of the clause C.

Flattening and unflattening of clauses need now to be formally defined. First the function unflat is defined as follows.

Definition 13 (Unflattening) *Let C be the clause $D \vee \overline{E}$ where D contains no equality literals and E is the conjunction $s_1 = t_1 \wedge s_2 = t_2 \wedge .. \wedge s_n = t_n$. $unflat(C) = D\epsilon_1\epsilon_2..\epsilon_n$ where ϵ_i is the mgu of s_i and t_i for $1 \leq i \leq n$.*

Unflattening is equivalent to resolving away all equality literals in the body of a clause using the single axiom equality theory

$$X = X$$

The set of flattened clauses is defined as follows.

Definition 14 (Flattening) *Let C and D be clauses. $D \in flat(C)$ if and only if C is an alphabetic variant of $unflat(D)$.*

5.4 Canonical flattening

Next a canonical flattening will be defined to capture the method of flattening applied in Example 12.

Definition 15 (Canonical flattening) *Let C and $D = F \vee \overline{E}$ be clauses in which F contains no equality literals and E is a conjunction of atoms. D is a canonical flattening of C, or $D \in cf(C)$, if and only if $D \in flat(C)$ and every literal in F has the form $p(v_1, .., v_n)$ or $\overline{p(v_1, .., v_n)}$ in which $v_1, .., v_n$ are distinct variables and every atom in E has the form $x = y$ or $x = f(y_1, .., y_m)$.*

Since it is intended to use canonical flattening to improve Algorithm 1 it is necessary to show the following.

Theorem 16 (Mgu of squaring a canonical flattening is a renaming) *Let C be a clause and D be a canonical flattening of C. If D is self-resolved to give E then the mgu involved in the resolution is a renaming of D.*
Proof. *Let $D = l \leftarrow l' \wedge B$, let $D\sigma$ be D standardised apart using the renaming σ of D and let $\langle l, \overline{l'\sigma} \rangle$ be the pair of literals involved in the resolution of D and $D\sigma$.*

Letting $l = p(u_1, .., u_n)$ and $l'\sigma = p(v_1, .., v_n)$ the mgu involved in the resolution is $\theta = \{v_1/u_1, .., v_n/u_n\}$. The variables $u_1, .., u_n$ and $v_1, .., v_n$ are all distinct since both D and $D\sigma$ are canonical flattenings of C (see Definition 15). The sets $\{u_1, .., u_n\}$ and $\{v_1, .., v_n\}$ are disjoint and none of the variables $\{v_1, .., v_n\}$ appear in D since D and $D\sigma$ have been standardised apart. Therefore, by Definition 8, θ is a renaming of D.

Next it is necessary to be assured that flattening a pair of clauses, resolving them and then unflattening the resolvent gives the same result as resolving the original clauses.

Lemma 17 (Unflattening distributes over resolution) *Let C_1 and C_2 be clauses and D_1 and D_2 be flattenings of C_1 and C_2 respectively. The clause F is the resolvent of D_1 and D_2 only if $unflat(F)$ is the resolvent of C_1 and C_2.*
Proof. *Let $D_1 = l \leftarrow B_1 \wedge E_1$ and $D_2 = l_2 \leftarrow l' \wedge B_2 \wedge E_2$ where E_1 and E_2 are the set of all equality atoms in D_1 and D_2. Let $C_1 = unflat(D_1) = (l \leftarrow B_1)\epsilon_1$ and $C_2 = unflat(D_2) = (l_2 \leftarrow l' \wedge B_2)\epsilon_2$ where ϵ_1 and ϵ_2 are the substitutions produced by resolving away E_1 and E_2 respectively. Let $\epsilon = \epsilon_1\epsilon_2$. Let D_1 and D_2 resolve on the pair of literals $\langle l, \overline{l'}\rangle$ to give F. Then $unflat(F)$ is equivalent to*

$$unflat(l_2 \leftarrow B_1 \wedge B_2 \wedge E_1 \wedge E_2 \wedge (l = l')) \tag{3}$$

since unflat will unify and remove l and l'. In the same way resolving C_1 and C_2 using the pair of literals $\langle l\epsilon, \overline{l'\epsilon}\rangle$ is equivalent to

$$unflat((l_2 \leftarrow B_1 \wedge B_2 \wedge (l = l'))\epsilon) \tag{4}$$

Formula 4 can be derived from formula 3 by resolving away $E_1 \wedge E_2$. This completes the proof.

As a corollary flattening, squaring and unflattening a clause must be equivalent to squaring the original clause.

Corollary 18 (Flattening, squaring and unflattening) *Let C be a clause and D be a flattening of C. The clause F is a square of D only if $unflat(F)$ is a square of C.*
Proof. *Let the renamings σ and τ be used to standardise apart C and D respectively. The corollary is now simply a restatement of Lemma 17 with $C_1 = C$, $C_2 = C\sigma$, $D_1 = D$ and $D_2 = D\tau$.*

This corollary applies to squaring a clause whereas the intention is to canonically flatten a clause, extract its square root and unflatten the result. Figure 2 iilustrates the following theorem.

Theorem 19 (Canonical flattening and squaring) *Let C be a clause and D be a canonical flattening of C. The clause F is a square of D only if F is a*

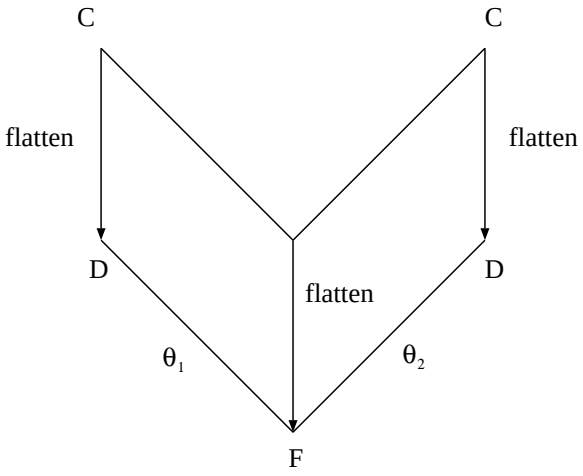

Figure 2: Flattening and squaring is equivalent to squaring and flattening

canonical flattening of a square of C.

Proof. *Given Corollary 18 it is only necessary to show that squaring a canonical flattening produces a canonical flattening. Let $D = l \leftarrow l' \wedge B \wedge E$ where E is conjunction of all equality atoms. Let σ standardises $D\sigma$ apart from D. Let D and $D\sigma$ resolve with complementary pair $\langle l\theta_1, \overline{l'\sigma}\theta_2\rangle$ to give $F =$*

$$l\sigma\theta_2 \leftarrow l'\theta_1 \wedge (B \wedge E)\theta_1 \wedge (B \wedge E)\sigma\theta_2$$

According to Lemma 11, since σ is a renaming of D and θ_2 is a renaming of $D\sigma$ their composition $\sigma\theta_2$ is a renaming of D. Every subcomponent of F is a renaming of a canonical flattening. Therefore F is a canonical flattening.

5.5 An improved square root algorithm

Theorem 19 suggests the following nondeterministic algorithm for extracting the square roots of a clause.

Algorithm 2 (Square root algorithm)

1. *Canonically flatten the input clause G to give clause F.*
2. *Choose from F a pair of literals $\langle l\theta_2, \overline{l'\theta_1}\rangle$ with the same predicate symbol.*
3. *Partition the clause $F - \{l\theta_2, \overline{l'\theta_1}\}$ into two equal cardinality conjuncts $B\theta_1$ and $B\theta_2$ from which B is constructed by taking the least generalisation of corresponding pairs of literals which are alphabetic variants.*
4. *Construct the renamings θ_1 and θ_2 of D by matching B to $B\theta_1$ and $B\theta_2$.*
5. *Apply θ_1^r to $l'\theta_1$ and θ_2^r to $l\theta_2$ to get l' and l respectively (see Lemma 10).*

6. *Return* $C = unflat(l \leftarrow l' \wedge B)$ *if* G *is the square of* C.

This can now be applied to Plotkin's example from Section 1.

Example 20 (Trace of Algorithm 2)
Let clause G *be* $p(f(f(X))) \leftarrow p(X)$.
The steps in Algorithm 2 are followed below.

1. F *is* $p(Y) \leftarrow p(X)$, $Y = f(Z)$, $Z = f(X)$.
2. *Let* $l\theta_2 = p(Y)$ *and* $l'\theta_1 = p(X)$.
3. *Let* $B\theta_2$ *be* $(Y = f(Z))$, $B\theta_1$ *be* $(Z = f(X))$ *and* B *be* $(U = f(V))$.
4. θ_1 *is the renaming* $\{U/Z, V/X\}$ *of* D *and* θ_2 *is the renaming* $\{U/Y, V/Z\}$ *of* D.
5. θ_1^r *is* $\{Z/U, X/V\}$, θ_2^r *is* $\{Y/U, Z/V\}$, $l = l\theta_2\theta_2^r = p(U)$ *and* $l' = l'\theta_1\theta_1^r = p(V)$.
6. *Return* $C = unflat(p(U) \leftarrow p(V), U = f(V))$ *which is* $p(f(V)) \leftarrow p(V)$. G *is the square of* C.

The following theorem shows that Algorithm 2 is complete and correct in a non-deterministic sense.

Theorem 21 (Completeness and correctness of Algorithm 2) *Let* C *be a clause and* G *be a square of* C. *When Algorithm 2 is presented with* G *there is a set of choices made in steps 1, 2 and 3 which will construct an alphabetic variant of* C.

Proof. *Algorithm 2 is correct since step 6 guarantees that any solution returned will be a square root of* C. *Therefore it is necessary to show it is complete in the sense that an alphabetic variant of every square root of* G *can be constructed given appropriate choices for the non-deterministic steps 1, 2 and 3. This is guaranteed by Theorem 19 since there is a canonical flattening* F *of* G *which has the form*

$$l\sigma\theta_2 \leftarrow l'\theta_1 \wedge (B \wedge E)\theta_1 \wedge (B \wedge E)\sigma\theta_2$$

if $D = l \leftarrow l' \wedge B \wedge E$ *and* $C = (l \leftarrow l' \wedge B)\epsilon$ *(see proof of Theorem 19).*

5.6 Problems with Algorithm 2

Despite Theorem 21, applying Algorithm 2 is not without problems. The problems are to do with generating flattened clauses in step 1. Consider again the canonically flattened clause $D =$

$$l \leftarrow l' \wedge B \wedge E$$

Original variables	u_1	v_1	..	u_n	v_n
θ_2	x_1	w_1	..	x_n	w_n
θ_1	w_1	y_1	..	w_n	y_n

Figure 3: Initialisation of substitution table

and its square $F =$

$$l\theta_2 \leftarrow l'\theta_1 \wedge B\theta_1 \wedge B\theta_2 \wedge E\theta_1 \wedge E\theta_2$$

Certain terms and literals will not appear in $unflat(F)$ under the following conditions.

- Literals m and m' appear in $B \wedge E$ for which $m\theta_1 = m'\theta_2$. Only one instance will therefore appear in F.
- The equality $v = t$ appears in $E\theta_1 \wedge E\theta_2$ and v does not appear in any other literal in D. The term t will therefore not appear in $unflat(F)$.
- The equalities $v = s$ and $v = t$ appear in $E\theta_1 \wedge E\theta_2$ and $s \neq t$. Only the term formed by unifying s and t appears in $unflat(F)$.

Restrictions could be devised for the clausal language to which Algorithm 2 can be straightforwardly applied. However this approach will not be followed up in this paper.

5.7 Tabulated substitutions

Steps 3 and 4 of Algorithm 2 cannot be wholly separated. From a programming point of view it makes sense to build up the substitutions θ_1 and θ_2 at the same time as matching literals in step 3. Only literals which lead to θ_1 and θ_2 being renamings should be matched. A simple way to do this is to build up a three row table of variables. The first row represents variables in the flattened version of C. The second and third rows represent the unique mappings of these variables in θ_2 and θ_1. When initialising this table it is possible to take advantage of a constraint related to the unification of the recursing literals l and l'. Suppose that $l = p(u_1, .., u_n)$ and $l' = p(v_1, .., v_n)$. These must unify to give a literal $p(w_1, .., w_n)$ where $l\theta_2 = p(x_1, .., x_n)$ and $l'\theta_1 = p(y_1, .., y_n)$. This constraint can be represented in the initialised substitution table shown in Figure 3.

The following example demonstrates the use of such a substitution table in Algorithm 2 for the predicate *split* which breaks a list into two approximately equal lengthed sublists.

Example 22 (Use of substitution table)
Let G be $split([H,I|J],[H|K],[I|L]) \leftarrow split(J,K,L)$. The steps in Algorithm 2 are followed below.

1. *F is $split(M,N,O) \leftarrow split(J,K,L), M=[H|P], P=[I|J], N=[H|K], O=[I|L]$.*

2. *Let $l\theta_2 = split(M,N,O)$ and $l'\theta_1 = split(J,K,L)$.*

3. *The substitution table is initialised to*

Original variables	Q	R	S	T	U	V
θ_2	M	w_1	N	w_2	O	w_3
θ_1	w_1	J	w_2	K	w_3	L

   Let $B\theta_2$ be $(M=[H|P], N=[H|K])$, $B\theta_1$ be $(P=[I|J], O=[I|L])$ and B be $(Q=[W|R], S=[W|V])$. The final substitution table is

Original variables	Q	R	S	T	V	W
θ_2	M	P	N	O	K	H
θ_1	P	J	O	K	L	I

   (Note that that the constraint described in Figure 3 led to the merging of columns of original variables T and U).

4. *θ_1 is the renaming $\{P/Q, J/R, O/S, K/T, L/V, I/W\}$ and $\theta_2$is the renaming $\{M/Q, P/R, N/S, O/T, K/V, H/W\}$.*

5. *θ_1^r is the renaming $\{Q/P, R/J, S/O, T/K, V/L, W/I\}$, $\theta_2^r$is the renaming $\{Q/M, R/P, S/N, T/O, V/K, W/H\}$, $l = l\theta_2\theta_2^r = split(Q,S,T)$ and $l' = l'\theta_1\theta_1^r = split(R,T,V)$.*

6. *Return $C = unflat(split(Q,S,T) \leftarrow split(R,T,V), Q=[W|R], S=[W|V])$ which is $split([W|R],[W|V],T) \leftarrow split(R,T,V)$. G is the square of C.*

6 Constructing the nth roots of a clause

In this section the construction of nth roots of clauses is investigated. Assume that D is a canonical flattening of the clause C . Figure 4 shows D resolved n times against itself to give the clause F, where $F \in D^n$. Suppose that this self-resolution always involves instances of the literals l and $\overline{l'}$ from D. This is

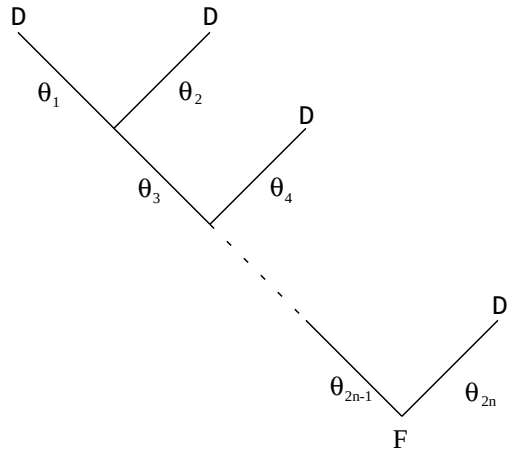

Figure 4: Self-resolving a clause n times

a simplifying assumption since D need not always self-resolve using instances of the same pair of literals. If clause D is

$$l \leftarrow l' \wedge B$$

then clause F is

$$\begin{aligned} l\theta_{2n} \leftarrow l'\theta_1\theta_3..\theta_{2n-1} \wedge\ & B\theta_1\theta_3..\theta_{2n-1} \wedge \\ & B\theta_2\theta_3\theta_5..\theta_{2n-1} \wedge \\ & \ldots \wedge \\ & B\theta_{2n-2}\theta_{2n-1} \wedge \\ & B\theta_{2n} \end{aligned}$$

Note that F contains n instances of B. One of these instances has the same substitution as the instance of l (θ_{2n}). Another instance of B has the same substitution as the instance of l' ($\theta_1\theta_3..\theta_{2n-1}$). In the above it was assumed that self-resolution always involved instances of the literals l and $\overline{l'}$ from D. Somewhat surprisingly, the phenomenon of corresponding substitutions occurs for all clauses in D^n.

Theorem 23 *Let D be a clause, n be an integer greater than 1 and F be a clause in D^n. F has the form $l\gamma_1 \vee \overline{l'\gamma_2} \vee B\gamma_1 \vee B\gamma_2 \vee .. \vee B\gamma_n$ where l and l' are atoms in D with the same predicate symbol and $B = D - \{l, \overline{l'}\}$.*

Proof. *The proof is by mathematical induction on k. The base case, $n = 2$, is true since if D self-recurses with complementary pair $\langle l\theta_1\overline{l'}\theta_2\rangle$ then F is $l\theta_2 \leftarrow l'\theta_1 \vee B\theta_1 \vee B\theta_2$. Assume the theorem is true for all integers from 2 to k and prove it follows for $k+1$. Let F in D^k be $l\gamma_1 \vee \overline{l'\gamma_2} \vee B\gamma_1 \vee B\gamma_2 \vee .. \vee B\gamma_k$ and D be $l \vee \overline{l'} \vee B$. Assume F and D resolve with complementary pair $\langle l\gamma_1\alpha, \overline{l'}\beta\rangle$ and mgu $\alpha\beta$ where the domains of α and β are subsets of $vars(l\gamma_1)$ and $vars(l')$ respectively. The*

resolvent of F and D in D^{k+1} is $l\beta \vee \overline{l'\gamma_2\alpha} \vee B\gamma_1\alpha \vee B\gamma_2\alpha \vee .. \vee B\gamma_k\alpha \vee B\beta$. This clause fulfills the condition and completes the proof.

By extending the arguments of the previous section it can be shown that F must be a canonical flattening of a clause G which is an nth power of C.

Theorem 24 (Canonical flattening and nth powers) *Let C be a clause and D be a canonical flattening of C. The clause F is an nth power of D only if F is a canonical flattening of an nth power of C.*
Proof. *The proof is by mathematical induction on k. The base case, $n = 1$, is true since D is a canonical flattening of C. Assume it is true for all n up to k and prove for $k+1$. F in D^{k+1} is constructed by resolving D with F' in D^k. From the inductive hypothesis and Lemma 17, F is a flattening of a clause in D^{k+1}. Since the mgu θ involved in constructing F is simply a composition of renamings it follows from Lemma 11 that θ is a renaming. The set difference between F and F' is a renaming of a subset of D and thus F is a canonical flattening of a clause in D^{k+1}. This completes the proof.*

This allows a slight variant of Algorithm 2 to be used for constructing the nth roots of a given clause. In the following we assume that n is given and that F contains the literals $l\beta$ and $\overline{l'\alpha}$.

Algorithm 3 (nth root algorithm)

1. *Canonically flatten the input clause G to give clause F.*
2. *Choose from F a pair of literals $\langle l\beta, \overline{l'\alpha}\rangle$ with the same predicate symbol.*
3. *Choose two equal cardinality conjuncts of literals $B\alpha$ and $B\beta$ from clause $H = (F - \{l\beta, \overline{l'\alpha}\})$ each of which have an nth of the cardinality of H from which B is constructed by taking the least generalisation of corresponding pairs of literals which are alphabetic variants.*
4. *Construct renamings α and β by matching B to $B\alpha$ and $B\beta$.*
5. *Apply α^r to $l\alpha$ and β^r to $l'\beta$ to get l and l' respectively.*
6. *Return $C = unflat(l \leftarrow l' \wedge B)$ if G is the square of C.*

The following example demonstrates Algorithm 3 on Plotkin's example from Section 1 with $n = 4$.

Example 25 (Trace of Algorithm 3)
Let clause G be $p(f(f(f(f(X))))) \leftarrow p(X)$.
The steps in Algorithm 3 are followed below.

1. *F is $p(H) \leftarrow p(X)$, $H = f(I)$, $I = f(J)$, $J = f(K)$, $K = f(X)$.*

2. *Let* $l\alpha = p(H)$ *and* $l'\beta = p(X)$.

3. *Let* $B\alpha$ *be* $(H = f(I))$, $B\beta$ *be* $(K = f(X))$ *and* B *be* $(L = f(M))$.

4. α *is the renaming* $\{L/H, M/I\}$ *and* β *is the renaming* $\{L/K, M/X\}$.

5. α^r *is the renaming* $\{H/L, I/M\}$, β^r *is the renaming* $\{K/L, X/M\}$, $l = l\alpha\alpha^r = p(L)$ *and* $l' = l'\beta\beta^r = p(M)$.

6. *Return* $C = unflat(p(L) \leftarrow p(M),\ L = f(M))$ *which is* $p(f(M)) \leftarrow p(M)$. G *is the fourth power of* C.

The following theorem shows the completeness and correctness of Algorithm 3.

Theorem 26 (Completeness and correctness of Algorithm 3) *Let* C *be a clause and* G *be an nth power of* C. *When Algorithm 3 is presented with* G *there is a set of choices made in steps 1, 2 and 3 which will construct an alphabetic variant of* C.

Proof. *Algorithm 2 is correct since step 6 guarantees that any solution returned will be a square root of* C. *Therefore it is necessary to show its completeness. This is guaranteed by Theorems 23 and 24.*

6.1 Problems with Algorithm 3

The primary problem with applying Algorithm 3 is deciding the value of n for any given clause. One clue here comes from the form of the canonically flattened clause shown in Section 6. Since clause $H = (F - \{l\alpha, \overline{l'\beta}\})$ contains n instances of B, the number of occurrences of every predicate and function symbol in H must be a multiple of n. From this fact there should be a small finite number of candidate values for n for any given clause. However, it should be noted that literals and terms can be lost in the three ways listed in Section 5.6.

Note also that Corollary 3 says that in order to construct clause C which implies clause D, it is necessary to first construct a clause E which subsumes D. But how should E be chosen? One way is to drop literals from a canonical flattening of D until appropriate numbers of occurrences of predicate and function symbols remain in E. Then take the nth root to give C.

7 Implication and background knowledge

In the normal setting of Inductive Logic Programming [14] generalisation is carried out in the presence of background knowledge. In this section the solution to inverting implication between clauses is extended to the case in which background knowledge is present.

Assume a background clausal theory T and a clause (or example) C which is not entailed by T. Assume that there is a single clause D such that

$$T \wedge D \models C$$

This problem can be transformed to one involving implication between single clauses as follows.

$$\begin{aligned} T \wedge D &\models C \\ D &\models (T \rightarrow C) \\ &\models D \rightarrow (T \rightarrow C) \\ &\models D \rightarrow \overline{(T \wedge \overline{C})} \\ &\models D \rightarrow \overline{(l_1 \wedge l_2 \wedge \ldots)} \end{aligned}$$

In the last line $(T \wedge \overline{C})$ is replaced with a conjunction of all ground literals which can be derived from $(T \wedge \overline{C})$. This can be viewed as replacing the formula with a model of the formula. Since $(l_1 \wedge l_2 \wedge \ldots)$ is a conjunction of literals, the last line above represents implication between two clauses. The clause $(\overline{l_1} \vee \overline{l_2} \vee \ldots)$ can be constructed to be of finite length if T is generative (see [16]) and elements of the model are only constructed to a finite depth of resolution. This clause can then be used to construct D using the methods described in previous sections.

8 Conclusion

In this paper the general problem of inverting implication is discussed. This problem is at the heart of research into Inductive Logic Programming and Machine Learning in general since all forms of generalisation involve inverting implication. The methods and algorithms described in this paper are derived from a first principles approach to the problem and extend previous approaches such as those using inverse resolution [15, 14, 22, 24] and relative least general generalisation [19, 2, 16].

Although a first attempt has been made at this problem in a previous paper by Lapointe and Matwin [11] the author believes the approach taken in this paper to be more general and comprehensive. Various remaining problems with the square root and nth root algorithms are described in Sections 5.6 and Section 6.1. The problems of time complexity of these algorithms is not discussed here. Also no implementation of the approach described in this paper has been made.

As Lapointe and Matwin noted, the advantages of extending the generalisation techniques beyond those of inverse resolution [14] lie in the fact that fewer examples are required to learn recursive clauses. Recursive clauses have not been vital for the success of several real world applications of Inductive Logic Programming [17, 10, 6, 5]. However, they are of central interest within problems

involving construction of arbitrary programs. Traditionally this area has involved deductive techniques within the area known as formal methods. According to a recent paper by Hoare [9]

> Given specification S, the task is to find a program P which satisfies it, in the sense that every possible observation of every possible behaviour of the program P will be among the behaviours described (and therefore permitted by) the specification of S. In logic, this can be assured with mathematical certainty by a proof of the simple implication
>
> $$\vdash P \rightarrow S.$$

Note that this problem is encompassed by the discussion in this paper. However, the requirements for Inductive Logic Programming are slightly weaker than those described by Hoare, since if the specification S is an incomplete set of examples then not all behaviours of P are defined. However, there is nothing in the present discussion to stop one making use of arbitrary (non-ground) formulae instead of examples. Such formulae could comprise a specification in the sense that "every possible observation of every possible behaviour of the program P will be among the behaviours described". The background knowledge referred to in Inductive Logic Programming maps to the the set of abstract data types and data operations used in formal methods approaches. Note also that using the approach in this paper $P \rightarrow S$ should be ensured by construction and is similar in that way to the approach of transformational programming introduced first by Burstall and Darlington [3].

Acknowledgements.
The author would like especially to thank Ashwin Srinivasan for useful input during this research. Thanks are also due to Wray Buntine, Stuart Russell and members of the Turing Institute Inductive Logic Programming group for helpful and interesting discussions on the topics in this paper. This work was supported by the Esprit Ecoles ILP project 6020.

Appendix

A Definitions from logic

A.1 Formulae in first order predicate calculus

A variable is represented by an upper case letter followed by a string of lower case letters and digits. A function symbol is a lower case letter followed by a string of lower case letters and digits. A predicate symbol is a lower case letter followed by a string of lower case letter and digits. The negation of F is $\overline{F}$. A

variable is a term, and a function symbol immediately followed by a bracketed n-tuple of terms is a term. Thus $f(g(X), h)$ is a term when f, g and h are function symbols and X is a variable. A predicate symbol immediately followed by a bracketted n-tuple of terms is called an atomic formula, or atom. Both l and $\overline{l}$ are literals whenever l is an atomic formula. In this case l is called a positive literal and $\overline{l}$ is called a negative literal. The literals l and $\overline{l}$ are said to be each others complements and form, in either order, a complementary pair. A finite set (possibly empty) of literals is called a clause. The empty clause is represented by $\square$. A clause represents the disjunction of its literals. Thus the clause $\{l_1, l_2, .. \overline{l_i}, \overline{l_{i+1}}, ...\}$ can be equivalently represented as $(l_1 \vee l_2 \vee .. \overline{l_i} \vee \overline{l_{i+1}} \vee ...)$ or $l_1, l_2, .. \leftarrow l_i, l_{i+1}, ...$ A Horn clause is a clause which contains at most one positive literal. A definite clause is a clause which contains exactly one positive literal. The positive literal in a definite clause is called the head of the clause while the negative literals are collectively called the body of the clause. A set of clauses is called a clausal theory. The empty clausal theory is represented by $\blacksquare$. A clausal theory represents the conjunction of its clauses. Thus the clausal theory $\{C_1, C_2, ...\}$ can be equivalently represented as $(C_1 \wedge C_2 \wedge ...)$. A set of Horn clauses is called a logic program. Apart from representing the empty clause and the empty theory, the symbols $\square$ and $\blacksquare$ represent the logical constants *False* and *True* respectively. Any clause, such as $l \leftarrow l$, which is equivalent to $\blacksquare$ is said to be a *tautology*. Literals, clauses and clausal theories are all well-formed-formulae (wff's). Let E be a wff or term. $vars(E)$ denotes the set of variables in E. E is said to be ground if and only if vars$(E) = \emptyset$.

A.2 Models and substitutions

A set of ground literals which does not contain a complementary pair is called an interpretation. Let M be an interpretation, C be a clause and $\mathcal{C}$ be the set of all ground clauses obtained by replacing the variables in C by ground terms. M is a model of C if and only if each clause in $\mathcal{C}$ contains at least one literal found in M. M is a model for clausal theory T if and only if M is a model for each clause in T. Let F_1 and F_2 be two wff's. F_1 semantically entails F_2, or $F_1 \models F_2$ if and only if every model of F_1 is a model of F_2. F_1 is said to syntactically entail F_2 using I, or $F_1 \vdash_I F_2$, if and only if F_2 can be derived from F_1 using the set of deductive inference rules I. The set of inference rules I is said to be deductively sound and complete if and only if $F_1 \vdash_I F_2$ whenever $F_1 \models F_2$. In this case the subscript can be dropped and one can merely write $F_1 \vdash F_2$. Let F_1 and F_2 be two wff's. F_1 is said to be more general than F_2 if and only if $F_1 \vdash F_2$. A wff F is satisfiable if there is a model for F and unsatisfiable otherwise. F is unsatisfiable if and only if $F \models \square$. The deduction theorem states that $F_1 \wedge F_2 \models F_3$ if and only if $F_1 \models F_2 \rightarrow F_3$.

Let $\theta = \{v_1/t_1, .., v_n/t_n\}$. θ is said to be a substitution when each v_i is a variable and each t_i is a term, and for no distinct i and j is v_i the same as v_j.

The set $\{v_1, .., v_n\}$ is called the domain of θ, or dom(θ), and $\{t_1, .., t_n\}$ the range of θ, or rng(θ). Lower case Greek letters are used to denote substitutions. Let E be a well-formed formula or a term and $\theta = \{v_1/t_1, .., v_n/t_n\}$ be a substitution. The instantiation of E by θ, written $E\theta$, is formed by replacing every occurrence of v_i in E by t_i. Let F be a well-formed formula and $\theta = \{u_1/v_1, .., u_n/v_n\}$ be a substitution. θ is a renaming of F if and only if $u_1, .., u_n$ are all distinct variables, $v_1, .., v_n$ are all distinct variables and $(vars(F) - \{u_1, .., u_n\}) \cap \{v_1, .., v_n\} = \emptyset$. Every sub-term within a given term or literal l can be uniquely referenced by its *place* within l. Places within terms or literals are denoted by n-tuples of natural numbers and defined recursively as follows. The term at place $\langle i \rangle$ within $f(t_0, .., t_m)$ is t_i. The term at place $\langle i_0, .., i_n \rangle$ within $f(t_0, .., t_m)$ is the term at place $\langle i_1, .., i_n \rangle$ in t_{i_0}. Let t be a term found at place p in literal l, where l is a literal within clause C. The place of t in C is denoted by the pair $\langle l, p \rangle$. Let E be a clause or a term and $\theta = \{v_1/t_1, .., v_n/t_n\}$ be a substitution. The corresponding inverse substitution θ^{-1} is $\{\langle t_1, \{p_{1,1}, .., p_{1,m_1}\}\rangle/v_1, .., \langle t_n, \{p_{n,1}, .., p_{1,m_n}\}\rangle/v_n\}$. An inverse substitution is applied by replacing all t_i at places $p_{i,1}, .., p_{i,m_i}$ within E by v_i. Clearly $E\theta\theta^{-1} = E$. Note that an inverse substitution is not strictly a substitution but rather a rewrite. Let C and D be clauses. It is said that C θ-subsumes D if and only if there exists a substitution θ such that $C\theta \subseteq D$.

A.3 Resolution

Let F_1 and F_2 be two wff's and θ be a renaming of F_1. $F_1\theta$ and F_2 are said to be standardised apart whenever there is no variable which occurs in both $F_1\theta$ and F_2. If θ is used to standardise apart formula F and $F\theta$ then F and $F\theta$ are said to be alphabetic variants. The substitution θ is said to be the unifier of the atoms l and l' whenever $l\theta = l'\theta$. μ is the most general unifier (mgu) of l and l' if and only if for all unifiers γ of l and l' there exists a substitution δ such that $(l\mu)\delta = l\gamma$. $((C - \{l\}) \cup (D - \{\overline{l'}\}))\theta$ is said to be the resolvent of the clauses C and D whenever C and D are standardised apart, $l \in C$, $\overline{l'} \in D$, θ is the mgu of l and l'. That is to say that $\langle l\theta, \overline{l'}\theta \rangle$ is a complementary pair. The resolvent of clauses C and D is denoted $(C \cdot D)$ when the complementary pair of literals is unspecified. The '$\cdot$' operator is commutative, non-associative and non-distributive.

Let T be a clausal theory. Robinson [21] defined the function $\mathcal{R}^n(T)$ recursively as follows. $\mathcal{R}^0(T) = T$. $\mathcal{R}^n(T)$ is the union of $\mathcal{R}^{n-1}$ and the set of all resolvents constructed from pairs of clauses in $\mathcal{R}^{n-1}(T)$. Robinson showed that T is unsatisfiable if and only if there is some n for which $\mathcal{R}^n(T)$ contains the empty clause ($\Box$).

References

[1] M. Bain and S. Muggleton. Non-monotonic learning. In D. Michie, editor, *Machine Intelligence 12*. Oxford University Press, 1991.

[2] W. Buntine. Generalised subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36(2):149–176, 1988.

[3] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24:44–67, 1977.

[4] B. Carlson, F. Kant, and Wünsche. A scheme for functions in logic programming. UPMAIL 57, Uppsala Programming Methodology and Artificial Intelligence Laboratory, Uppsala, Sweden, 1989.

[5] B. Dolsak and S. Muggleton. The application of Inductive Logic Programming to finite element mesh design. In S.H. Muggleton, editor, *Inductive Logic Programming*. Academic Press, London, 1992.

[6] C. Feng. Inducing temporal fault dignostic rules from a qualitative model. In S.H. Muggleton, editor, *Inductive Logic Programming*. Academic Press, London, 1992.

[7] E. Giovannetti and C. Moiso. Some aspects of the integration between logic programming and functional programming. In P Jorrand and V. Sgurev, editors, *Artificial Intelligence II. Methodology, Systems, Applications*, pages 69–79. North-Holland, 1987.

[8] G. Gottlob. Subsumption and implication. *Information Processing Letters*, 24(2):109–111, 1987.

[9] C.A.R. Hoare. Programs are predicates. In *Proceedings of the Final Fifth Generation Conference*, Tokyo, 1992. Ohmsha.

[10] R. King, S. Muggleton, and M.J.E. Sternberg. Drug design by machine learning. *Proceedings of the National Academy of Sciences*, 1992. To appear.

[11] S. Lapointe and S. Matwin. Sub-unification: a tool for efficient induction of recursive programs. In *Proceedings of the Ninth International Machine Learning Conference*, Los Altos, 1992. Morgan Kaufmann.

[12] C. Lee. *A completeness theorem and a computer program for finding theorems derivable from given axioms*. PhD thesis, University of California, Berkeley, 1967.

[13] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1984.

[14] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[15] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352. Kaufmann, 1988.

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

[17] S. Muggleton, R. King, and M. Sternberg. Predicting protein secondary structure using inductive logic programming, 1991. submitted to Proteins.

[18] T. Niblett. A study of generalisation in logic programs. In *EWSL-88*, London, 1988. Pitman.

[19] G.D. Plotkin. *Automatic Methods of Inductive Inference.* PhD thesis, Edinburgh University, August 1971.

[20] R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[21] J.A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, January 1965.

[22] C. Rouveirol. Extensions of inversion of resolution applied to theory completion. In S.H. Muggleton, editor, *Inductive Logic Programming.* Academic Press, London, 1992.

[23] C. Rouveirol and J-F Puget. A simple and general solution for inverting resolution. In *EWSL-89*, pages 201–210, London, 1989. Pitman.

[24] R. Wirth. Completing logic programs by inverse resolution. In *EWSL-89*, pages 239–250, London, 1989. Pitman.