Appears in the Proceeding of ILP 2000

A New Algorithm for Learning Range Restricted Horn Expressions (Extended Abstract) [⋆]

Marta Arias and Roni Khardon

Division of Informatics, University of Edinburgh
The King's Buildings, Edinburgh EH9 3JZ, Scotland
{marta,roni}@dcs.ed.ac.uk

Abstract. A learning algorithm for the class of *range restricted Horn expressions* is presented and proved correct. The algorithm works within the framework of *learning from entailment*, where the goal is to exactly identify some pre-fixed and unknown expression by making questions to *membership* and *equivalence* oracles. This class has been shown to be learnable in previous work. The main contribution of this paper is in presenting a more direct algorithm for the problem which yields an improvement in terms of the number of queries made to the oracles. The algorithm is also adapted to the class of Horn expressions with inequalities on all syntactically distinct terms where a significant improvement in the number of queries is obtained.

1 Introduction

This paper considers the problem of learning an unknown first order expression[1] T from examples of clauses that T entails or does not entail. This type of learning framework is known as *learning from entailment.* [FP93] formalised learning from entailment using equivalence queries and membership queries and showed the learnability of propositional Horn expressions. Generalising this result to the first order setting is of clear interest. Learning first order Horn expressions has become a fundamental problem in *Inductive Logic Programming* [MR94]. Theoretical results have shown that learning from examples only is feasible for very restricted classes [Coh95a] and that, in fact, learnability becomes intractable when slightly more general classes are considered [Coh95b]. To tackle this problem, learners have been equipped with the ability to ask questions. It is the case that with this ability larger classes can be learned. In this paper, the questions that the learner is allowed to ask are *membership* and *equivalence* queries. While our work is purely theoretical, there are systems that are able to learn using equivalence and membership queries (MIS [Sha83], CLINT [RB92], for example). These ideas have also been used in systems that learn from examples only [Kha00].

A learning algorithm for the class of *range restricted Horn expressions* is presented. The main property of this class is that all the terms in the conclusion

[⋆] This work was partly supported by EPSRC Grant GR/M21409.

[1] The unknown expression to be identified is commonly referred to as *target* expression.

of a clause appear in the antecedent of the clause, possibly as subterms of more complex terms. This work is based on previous results on learnability of *function free Horn expressions* and *range restricted Horn expressions*. The problem of learning *range restricted Horn expressions* was solved in [Kha99b] by reducing it to the problem of learning *function free Horn expressions*, solved in [Kha99a]. The algorithm presented here has been obtained by retracing this reduction and using the resulting algorithm as a starting point. However, it has been significantly modified and improved. The algorithm in [Kha99a,Kha99b] uses two main procedures. The first, given a counterexample clause, minimises the clause while maintaining it as a counterexample. The minimisation procedure used here is stronger, resulting in a clause which includes a syntactic variant of a target clause as a subset. The second procedure combines two examples producing a new clause that may be a better approximation for the target. While the algorithm in [Kha99a,Kha99b] uses direct products of models we use an operation based on the lgg (least general generalisation [Plo70]). The use of lgg seems a more natural and intuitive technique to use for learning from entailment, and it has been used before, both in theoretical and applied work [Ari97,RT98,RS98,MF92].

We extend our results to the class of *fully inequated range restricted Horn expressions*. The main property of this class is that it does not allow unification of its terms. To avoid unification, every clause in this class includes in its antecedent a series of inequalities between all its terms. With a minor modification to the learning algorithm, we are able to show learnability of the class of fully inequated range restricted Horn expressions. The more restricted nature of this class allows for better bounds to be derived.

The rest of the paper is organised as follows. Section 2 gives some preliminary definitions. The learning algorithm is presented in Section 3 and proved correct in Section 4. The results are extended to the class of fully inequated range restricted Horn expressions in Section 5. Finally, Section 6 compares the results obtained in this paper with previous results and includes some concluding remarks.

2 Preliminaries

We consider a subset of the class of universally quantified expressions in first order logic. Definitions of first order languages can be found in standard texts, e.g. [Llo87]. We assume familiarity with notions such as *term*, *atom*, *literal*, *Horn clause* in the part of syntax and *interpretation*, *truth value*, *satisfiability* and *logical implication* in the part of semantics.

A *Range Restricted Horn clause* is a definite Horn clause in which every term appearing in its consequent also appears in its antecedent, possibly as a subterm of another term. A *Range Restricted Horn Expression* is a conjunction of Range Restricted Horn clauses.

A *multi-clause* is a pair of the form $[s, c]$, where both s and c are sets of literals such that $s \cap c = \emptyset$; s is the *antecedent* of the multi-clause and c is the *consequent*. Both are interpreted as the conjunction of the literals they contain. Therefore, the multi-clause $[s, c]$ is interpreted as the logical expression $\bigwedge_{b \in c} s \rightarrow b$. An ordinary clause $C = s_c \rightarrow b_c$ corresponds to the multi-clause $[s_c, \{b_c\}]$.

We say that a logical expression T *implies* a multi-clause $[s, c]$ if it implies all of its single clause components. That is, $T \models [s, c]$ iff $T \models \bigwedge_{b \in c} s \rightarrow b$.

A multi-clause $[s, c]$ is *correct* w.r.t an expression T iff $T \models [s, c]$. A multi-clause $[s, c]$ is *exhaustive* w.r.t T if every literal $b \notin s$ such that $T \models s \rightarrow b$ is included in c. A multi-clause is *full* w.r.t T if it is correct and exhaustive w.r.t. T.

The *size* of a term is the number of occurrences of variables plus twice the number of occurrences of function symbols (including constants). The *size* of an atom is the sum of the sizes of the (top-level) terms it contains plus 1. Finally, the *size* of a multi-clause $[s, c]$ is the sum of sizes of atoms in s.

Let s_1, s_2 be any two sets of literals. We say that s_1 *subsumes* s_2 (denoted $s_1 \preceq s_2$) if and only if there exists a substitution θ such that $s_1 \cdot \theta \subseteq s_2$. We also say that s_1 is a *generalisation* of s_2.

Let s be any set of literals. Then $ineq(s)$ is the set of all inequalities between terms appearing in s. As an example, let s be the set $\{p(x, y), q(f(y))\}$ with terms $\{x, y, f(y)\}$. Then $ineq(s) = \{x \neq y, x \neq f(y), y \neq f(y)\}$ also written as $(x \neq y \neq f(y))$ for short.

Least General Generalisation. The algorithm proposed uses the *least general generalisation* or *lgg* operation [Plo70]. This operation computes a generalisation of two sets of literals. It works as follows.

The *lgg* of two terms $f(s_1, ..., s_n)$ and $g(t_1, ..., t_m)$ is defined as the term $f(lgg(s_1, t_1), ..., lgg(s_n, t_n))$ if $f = g$ and $n = m$. Otherwise, it is a new variable x, where x stands for the *lgg* of that pair of terms throughout the computation of the *lgg* of the set of literals. This information is kept in what we call the *lgg* table. The *lgg* of two *compatible* atoms $p(s_1, ..., s_n)$ and $p(t_1, ..., t_n)$ is $p(lgg(s_1, t_1), ..., lgg(s_n, t_n))$. The *lgg* is only defined for compatible atoms, that is, atoms with the same predicate symbol and arity. The *lgg* of two *compatible* positive literals l_1 and l_2 is the *lgg* of the underlying atoms. The *lgg* of two *compatible* negative literals l_1 and l_2 is the negation of the *lgg* of the underlying atoms. Two literals are compatible if they share predicate symbol, arity and sign. The *lgg* of two sets of literals s_1 and s_2 is the set $\{lgg(l_1, l_2) \mid (l_1, l_2)$ are two compatible literals of s_1 and $s_2\}$.

Example 1. Let $s_1 = \{p(a, f(b)), p(g(a, x), c), q(a)\}$ and $s_2 = \{p(z, f(2)), q(z)\}$ with $lgg(s_1, s_2) = \{p(X, f(Y)), p(Z, V), q(X)\}$. The *lgg* table produced during the computation of $lgg(s_1, s_2)$ is

`[a - z => X]`	(from $p(\underline{a}, f(b))$ with $p(\underline{z}, f(2))$)
`[b - 2 => Y]`	(from $p(a, f(\underline{b}))$ with $p(z, f(\underline{2}))$)
`[f(b) - f(2) => f(Y)]`	(from $p(a, \underline{f(b)})$ with $p(z, \underline{f(2)})$)
`[g(a,x) - z => Z]`	(from $p(\underline{g(a, x)}, c)$ with $p(\underline{z}, f(2))$)
`[c - f(2) => V]`	(from $p(g(a, x), \underline{c})$ with $p(z, \underline{f(2)})$)

2.1 The Learning Model

We consider the model of *exact learning from entailment* [FP93]. In this model examples are clauses. Let T be the target expression, H any hypothesis presented by the learner and C any clause. An example C is positive for a target theory T if $T \models C$, otherwise it is negative. The learning algorithm can make two types of queries. An *Entailment Equivalence Query* (*EntEQ*) returns "Yes" if $H = T$ and otherwise it returns a clause C that is a counter example, i.e., $T \models C$ and $H \not\models C$ or vice versa. For an *Entailment Membership Query* (*EntMQ*), the learner presents a clause C and the oracle returns "Yes" if $T \models C$, and "No" otherwise. The aim of the learning algorithm is to exactly identify the target expression T by making queries to the equivalence and membership oracles.

2.2 Transforming the target expression

In this section we describe the transformation $U(T)$ performed on any target expression T. This transformation is never computed by the learning algorithm; it is only used in the analysis of the proof of correctness. Related work in [SEMF98] also uses inequalities in clauses, although the learning algorithm and approach are completely different.

The idea is to create from every clause C in T the set of clauses $U(C)$. Every clause in $U(C)$ corresponds to the original clause C with its terms unified in a unique way, different from every other clause in $U(C)$. All possible unifications of terms of C are covered by one of the clauses in $U(C)$. The clauses in $U(C)$ will only be satisfied if the terms are unified in exactly that way. To achieve this, a series of appropriate inequalities are prepended to every transformed clause's antecedent. This process is described in Figure 1. It uses the *most general unifier* operation or *mgu*. Details about the *mgu* can be found in [Llo87].

```
1. Set $U(T)$ to be the empty expression ($T$ is the expression to be transformed).
2. For every clause $C = s_c \to b_c$ in $T$ and for every partition $\pi$ of the set of terms
   (and subterms) appearing in $C$ do
     – Let the partition $\pi$ be $\{\pi_1, \pi_2, ..., \pi_l\}$. Set $\sigma_0$ to $\emptyset$.
     – For $i = 1$ to $l$ do
         • If $\pi_i \cdot \sigma_{i-1}$ is unifiable, then $\theta_i = mgu(\pi_i \cdot \sigma_{i-1})$ and $\sigma_i = \sigma_{i-1} \cdot \theta_i$.
         • Otherwise, discard the partition.
     – If there are two classes $\pi_i$ and $\pi_j$ $(i \neq j)$ such that $\pi_i \cdot \sigma_l = \pi_j \cdot \sigma_l$, then discard
       the partition.
     – Otherwise, set $U_\pi(C) = ineq(s_c \cdot \sigma_l), s_c \cdot \sigma_l \to b_c \cdot \sigma_l$ and $U(T) = U(T) \wedge U_\pi(C)$.
3. Return $U(T)$.
```

Fig. 1. The transformation algorithm

We construct $U(T)$ from T by considering every clause separately. For a clause C in T with set of terms $\mathcal{T}$, we generate a set of clauses $U(C)$. To do that, consider all partitions of the terms in $\mathcal{T}$; each such partition, say π, can generate a clause of $U(C)$, denoted $U_\pi(C)$. Therefore, $U(T) = \bigwedge_{C \in T} U(C)$ and $U(C) = \bigwedge_{\pi \in Partitions(\mathcal{T})} U_\pi(C)$. The clause $U_\pi(C)$ is computed as follows. Taking one class at a time, compute its mgu if possible. If there is no mgu, discard that partition. Otherwise, apply the unifying substitution to the rest of elements in classes not handled yet, and continue with the following class. If the representatives[2] of any two distinct classes happen to be equal, then discard that partition as well. This is because the inequality between the representatives of those two classes will never be satisfied (they are equal!), and the resulting clause is superfluous. When all classes have been unified, we proceed to translate the clause C. All (top-level) terms appearing in C are substituted by the mgu found for the class they appear in, and the inequalities are included in the antecedent. This gives the transformed clause $U_\pi(C)$.

Example 2. Let the clause to be transformed be $C = p(f(x), f(y), g(z)) \rightarrow q(x, y, z)$. The terms appearing in C are $\{x, y, z, f(x), f(y), g(z)\}$. We consider some possible partitions:

– When $\pi = \{x, y\}, \{z\}, \{f(x), f(y)\}, \{g(z)\}$.

Stage	mgu	θ	σ	*Partitions Left*
0			$\emptyset$	$\{x,y\}, \{z\}, \{f(x), f(y)\}, \{g(z)\}$
1	$\{x,y\}$	$\{y \mapsto x\}$	$\{y \mapsto x\}$	$\{z\}, \{f(x), f(x)\}, \{g(z)\}$
2	$\{z\}$	$\emptyset$	$\{y \mapsto x\}$	$\{f(x), f(x)\}, \{g(z)\}$
3	$\{f(x), f(x)\}$	$\emptyset$	$\{y \mapsto x\}$	$\{g(z)\}$
4	$\{g(z)\}$	$\emptyset$	$\{y \mapsto x\}$	

$C \cdot \sigma_4 = p(f(x), f(x), g(z)) \rightarrow q(x, x, z)$ and
$U_\pi(C) = (x \neq z \neq f(x) \neq g(z)), p(f(x), f(x), g(z)) \rightarrow q(x, x, z)$.

– When $\pi = \{x, y, z\}, \{f(x), g(z)\}, \{f(y)\}$.

Stage	mgu	θ	σ	*Partitions Left*
0			$\emptyset$	$\{x,y,z\}, \{f(x), g(z)\}, \{f(y)\}$
1	$\{x,y,z\}$	$\{y,z \mapsto x\}$	$\{y,z \mapsto x\}$	$\{f(x), g(x)\}, \{f(x)\}$
2	$\{f(x), g(x)\}$	No mgu		PARTITION DISCARDED

If the target expression T has m clauses, then the number of clauses in the transformation $U(T)$ is bounded by mt^t, with t being the maximum number of distinct terms appearing in one clause of T (the number of partitions of a set with t elements is bounded by t^t). Notice however that there are many partitions that will be discarded by the process, for example all those partitions containing some class with two functional terms with different top-level function symbol, or partitions containing some class with two terms such that one is a subterm of the other. Therefore, the number of clauses in the transformation will be in practice much smaller than mt^t.

The transformation $U(T)$ of a range restricted expression T is also range restricted. It can be also proved that $T \models U(T)$, since every clause in $U(T)$ is

[2] We call the representative of a class any of its elements applied to the mgu.

subsumed by some clause in T. As a consequence, $U(T) \models C$ implies $T \models C$ and hence $U(T) \models [s, c]$ implies $T \models [s, c]$.

3 The Algorithm

The algorithm keeps a sequence S of representative counterexamples. The hypothesis H is generated from this sequence, and the main task of the algorithm is to *refine* the counterexamples in S in order to get a more accurate hypothesis in each iteration of the main loop, line 2, until hypothesis and target expression coincide.

There are two basic operations on counterexamples that need to be explained in detail. These are *minimisation* (line 2b), that takes a counterexample as given by the equivalence oracle and produces a positive, full counterexample; and *pairing* (line 2c), that takes two counterexamples and generates a series of candidate counterexamples. The counterexamples obtained by combination of previous ones (by *pairing* them) are the candidates to refine the sequence S. These operations are carefully explained in the following sections 3.1 and 3.2.

The algorithm uses the procedure rhs. The first version of rhs has 1 input parameter only. Given a set of literals s, $rhs(s)$ computes the set of all literals not in s implied by s w.r.t. the target expression. That is, $rhs(s) = \{b \notin s \mid EntMQ(s \rightarrow b) = Yes\}$.

The second version of rhs has 2 input parameters. Given the sets s and c, $rhs(s, c)$ outputs those literals in c that are implied by s w.r.t. the target expression. That is, $rhs(s, c) = \{b \in c \mid b \notin s \text{ and } EntMQ(s \rightarrow b) = Yes\}$. Notice that in both cases the literals b considered are literals containing terms that appear in s only. Both versions ask membership queries to find out which of the possible consequents are correct. The resulting sets are in both cases finite since the target expression is range restricted.

```
1. Set $S$ to be the empty sequence and $H$ to be the empty hypothesis.
2. Repeat until $EntEQ(H)$ returns "Yes":
   (a) Let $x$ be the (positive) counterexample received ($T \models x$ and $H \not\models x$).
   (b) Minimise counterexample $x$ - use calls to $EntMQ$.
       Let $[s_x, c_x]$ be the minimised counterexample produced.
   (c) Find the first $[s_i, c_i] \in S$ such that there is a basic pairing $[s, c]$ of terms of
       $[s_i, c_i]$ and $[s_x, c_x]$ satisfying:
         i. $size(s) \lneq size(s_i)$
         ii. $rhs(s, c) \neq \emptyset$
   (d) If such an $[s_i, c_i]$ is found then replace it by the multi-clause $[s, rhs(s, c)]$.
   (e) Otherwise, append $[s_x, c_x]$ to $S$.
   (f) Set $H$ to be $\bigwedge_{[s,c] \in S} \{s \rightarrow b \mid b \in c\}$.
3. Return $H$
```

Fig. 2. The learning algorithm

3.1 Minimising the counterexample

The minimisation procedure has to transform a counterexample clause x as generated by the equivalence query oracle into a multi-clause counterexample $[s_x, c_x]$ ready to be handled by the learning algorithm. This is done by removing literals and generalising terms.

```
1. Let $x$ be the counterexample obtained by the EntEQ oracle.
2. Let $s_x$ be the set of literals $\{b \mid H \models antecedent(x) \rightarrow b\}$ and set $c_x$ to $rhs(s_x)$.
3. Repeat until no more changes are made
     – For every functional term $t$ appearing in $s_x$, in decreasing order of size, do
         • Let $[s'_x, c'_x]$ be the multi-clause obtained from $[s_x, c_x]$ after substituting all
           occurrences of the term $f(\bar{t})$ by a new variable $x_{f(\bar{t})}$.
         • If $rhs(s'_x, c'_x) \neq \emptyset$, then set $[s_x, c_x]$ to $[s'_x, rhs(s'_x, c'_x)]$.
4. Repeat until no more changes are made
     – For every term $t$ appearing in $s_x$, in increasing order of size, do
         • Let $[s'_x, c'_x]$ be the multi-clause obtained after removing from $[s_x, c_x]$ all
           those literals containing $t$.
         • If $rhs(s'_x, c'_x) \neq \emptyset$, then set $[s_x, c_x]$ to $[s'_x, rhs(s'_x, c'_x)]$.
5. Return $[s_x, c_x]$.
```

Fig. 3. The minimisation procedure

The minimisation procedure constructs first a full multi-clause that will be refined in the following steps. To do this, all literals implied by $antecedent(x)$ and the clauses in the hypothesis will be included in the first version of the new counterexample's antecedent: s_x (line 2). This can be done by forward chaining using the hypothesis' clauses, starting with the literals in $antecedent(x)$. Finally, the consequent of the first version of the new counterexample (c_x) will be constructed as $rhs(s_x)$.

Next, we enter the loop in which terms are generalised (line 3). We do this by considering every term that is not a variable (constants are also included), one at a time. The way to proceed is to substitute every occurrence of the term by a new variable, and then check whether the multi-clause is still positive. If so, the counterexample is updated to the new multi-clause obtained. The process finishes when there are no terms to be generalised in $[s_x, c_x]$. Note that if some term cannot be generalised, it will stay so during the computation of this loop, so that by keeping track of the failures, unnecessary computation time and queries can be saved.

Finally, we enter the loop in which literals are removed (line 4). We do this by considering one term at a time. We remove every literal containing that term in s_x and c_x and check if the multi-clause is still positive. If so, the counterexample is updated to the new multi-clause obtained. The process finishes when there are no terms to be dropped in $[s_x, c_x]$.

Example 3. Parentheses are omitted and the function f is unary. Let T be the single clause $p(fx) \to q(x)$. We start with counterexample $[p(fa), q(b) \to q(a)]$ as obtained after step 2 of the minimisation procedure.

$[s_x, c_x]$	*Stage*	$[s'_x, c'_x]$	$rhs(s'_x, c'_x)$	*To check*
	GEN			$\{fa, a, b\}$
$[p(fa), q(b) \to q(a)]$	$fa \mapsto X$	$[p(X), q(b) \to q(a)]$	$\emptyset$	$\{a, b\}$
$[p(fa), q(b) \to q(a)]$	$a \mapsto X$	$[p(fX), q(b) \to q(X)]$	$q(X)$	$\{b\}$
$[p(fX), q(b) \to q(X)]$	$b \mapsto Y$	$[p(fX), q(Y) \to q(X)]$	$q(X)$	$\{\}$
	DROP			$\{X, Y, fX\}$
$[p(fX), q(Y) \to q(X)]$	X	$[q(Y) \to \emptyset]$	$\emptyset$	$\{Y, fX\}$
$[p(fX), q(Y) \to q(X)]$	Y	$[p(fX) \to q(X)]$	$q(X)$	$\{fX\}$
$[p(fX) \to q(X)]$	fX	$[\emptyset \to q(X)]$	$\emptyset$	$\{\}$
$[p(fX) \to q(X)]$				

3.2 Pairings

A crucial process in the algorithm is how two counterexamples are combined into a new one, hopefully yielding a better approximation of some target clause. The operation proposed here uses pairings of clauses, based on the *lgg*.

We have two multi-clauses, $[s_x, c_x]$ and $[s_i, c_i]$ that need to be combined. To do so, we generate a series of matchings between the terms of s_x and s_i, and any of these matchings will produce the candidate to refine the sequence S.

Matchings. A matching is a set whose elements are pairs of terms $t_x - t_i$, where $t_x \in s_x$ and $t_i \in s_i$. If s_x contains less terms than s_i, then there should be an entry in the matching for every term in s_x. Otherwise, there should be an entry for every term in s_i. That is, the number of entries in the matching equals the minimum of the number of terms in s_x and s_i. We only use 1-1 matchings, i.e., once a term has been included in the matching it cannot appear in any other entry of the matching. Usually, we denote a matching by the Greek letter σ.

Example 4. Let $[s_x, c_x]$ be $[\{p(a, b)\}, \{q(a)\}]$ with terms $\{a, b\}$. Let $[s_i, c_i]$ be $[\{p(f(1), 2)\}, \{q(f(1))\}]$ with terms $\{1, 2, f(1)\}$. The possible matchings are:

$$\sigma_1 = \{a - 1, b - 2\} \quad \sigma_3 = \{a - 2, b - 1\} \quad \sigma_5 = \{a - f(1), b - 1\}$$
$$\sigma_2 = \{a - 1, b - f(1)\} \quad \sigma_4 = \{a - 2, b - f(1)\} \quad \sigma_6 = \{a - f(1), b - 2\}$$

An *extended matching* is an ordinary matching with an extra column added to every entry of the matching. This extra column contains the *lgg* of every pair in the matching. The *lgg*s are simultaneous, that is, they share the same table.

An extended matching σ is *legal* if every subterm of some term appearing as the *lgg* of some entry, also appears as the *lgg* of some other entry of σ. An ordinary matching is legal if its extension is.

Example 5. Let σ_1 be $\{a - c, f(a) - b, f(f(a)) - f(b), g(f(f(a))) - g(f(f(c)))\}$ and $\sigma_2 = \{a - c, f(a) - b, f(f(a)) - f(b)\}$. The matching σ_1 is not legal, since the term $f(X)$ is not present in its extension column and it is a subterm of $g(f(f(X)))$, which is present. The matching σ_2 is legal.

Extended σ_1	*Extended σ_2*
`[a - c => X]`	`[a - c => X]`
`[f(a) - b => Y]`	`[f(a) - b => Y]`
`[f(f(a)) - f(b) => f(Y)]`	`[f(f(a)) - f(b) => f(Y)]`
`[g(f(f(a))) - g(f(f(c))) => g(f(f(X)))]`	

Our algorithm considers yet a more restricted type of matching. A *basic matching* σ is defined for two multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$ such that the number of terms in s_x is less than or equal to the number of terms in s_i. It is a 1-1, legal matching such that if entry $f(t_1, ..., t_n) - g(r_1, ..., r_m) \in \sigma$, then $f = g$, $n = m$ and $t_i - r_i \in \sigma$ for all $i = 1, ..., n$. Notice this is not a symmetric operation, since $[s_x, c_x]$ is required to have less distinct terms than $[s_i, c_i]$.

To construct basic matchings given $[s_x, c_x]$ and $[s_i, c_i]$, consider all possible matchings between the *variables* in s_x and the *terms* in s_i only. Complete them by adding the functional terms in s_x that are not yet included in the basic matching in an upwards fashion, beginning with the more simple terms. For every term $f(t_1, ..., t_n)$ in s_x such that all $t_i - r_i$ (with $i = 1, ..., n$) appear already in the basic matching, add a new entry $f(t_1, ..., t_n) - f(r_1, ..., r_n)$. Notice this is not possible if $f(r_1, ..., r_n)$ does not appear in s_i or the term $f(r_1, ..., r_n)$ has already been used. In this case, we cannot complete the matching and it is discarded. Otherwise, we continue until all terms in s_x appear in the matching. By construction, constants in s_x must be matched to the same constants in s_i.

Example 6. Let s_x be $\{p(a, fx)\}$ containing the terms $\{a, x, fx\}$. Let s_i be $\{p(a, f1), p(a, 2)\}$ containing terms $\{a, 1, 2, f1\}$. No parentheses for functions are written. The basic matchings to consider are the following.

- Starting with `[x - a]`: cannot add `[a - a]`, therefore discarded.
- Starting with `[x - 1]`: can be completed with `[a - a]` and `[fx - f1]`.
- Starting with `[x - 2]`: cannot add `[fx - f2]`, therefore discarded.
- Starting with `[x - f1]`: cannot add `[fx - ff1]`, therefore discarded.

One of the key points of our algorithm lies in reducing the number of matchings needed to be checked by ruling out some of the candidate matchings that do not satisfy some restrictions imposed. By doing so we avoid testing too many pairings and hence avoid making unnecessary calls to the oracles. One of the restrictions has already been mentioned, it consists in considering basic pairings only as opposed to considering every possible matching. This reduces the t^t possible distinct matchings to only t^k distinct *basic* pairings[3]. The other restriction on the candidate matching consists in the fact that every one of its entries must appear in the original *lgg* table.

Pairings. The input to a pairing consists of 2 multi-clauses together with a matching between the terms appearing in them. We say that the pairing is

[3] There are a maximum of t^k basic matchings between $[s_x, c_x]$ with k variables and $[s_i, c_i]$ with t terms, since we only combine *variables* of s_x with *terms* in s_i.

induced by the matching under consideration. A *basic* pairing is a pairing for which the inducing matching is basic.

The antecedent s of the pairing is computed as the lgg of s_x and s_i restricted to the matching inducing it. An atom is included in the pairing only if all its top-level terms appear as entries in the extended matching. This restriction is quite strong in the sense that, for example, if an atom $p(a)$ appears in both s_x and s_i then their lgg $p(a)$ will not be included unless the entry `[a - a => a]` appears in the matching. Therefore an entry in the matching is relevant only if it appears in the lgg table. We consider matchings with relevant entries only, i.e., matchings that are subsets of the lgg table.

To compute the consequent c of the pairing, the union of the sets $lgg_{|_\sigma}(s_x, c_i)$, $lgg_{|_\sigma}(c_x, s_i)$ and $lgg_{|_\sigma}(c_x, c_i)$ is computed. Note that in the consequent all the possible lggs of pairs among $\{s_x, c_x, s_i, c_i\}$ are included except $lgg_{|_\sigma}(s_x, s_i)$, that constitutes the antecedent. When computing any of the lggs, the same table is used. That is, the same pair of terms will be bound to the same expression in any of the four possible lggs that are computed in a pairing. To summarise:

$$[s, c] = [lgg_{|_\sigma}(s_x, s_i), lgg_{|_\sigma}(s_x, c_i) \cup lgg_{|_\sigma}(c_x, s_i) \cup lgg_{|_\sigma}(c_x, c_i)].$$

Example 7. Both examples have the same terms as in Example 6, so there is only one basic matching. Ex. 7.1 shows how to compute a pairing. Ex. 7.2 shows that a basic matching may be rejected if it does not agree with the lgg table.

	Example 7.1	*Example 7.2*
s_x	$\{p(a, fx)\}$	$\{p(a, fx)\}$
s_i	$\{p(a, f1), p(a, 2)\}$	$\{q(a, f1), p(a, 2)\}$
$lgg(s_x, s_i)$	$\{p(a, fX), p(a, Y)\}$	$\{p(a, Y)\}$
lgg table	`[a - a => a]` `[x - 1 => X]` `[fx - f1 => fX]` `[fx - 2 => Y]`	`[a - a => a]` `[fx - 2 => Y]`
basic σ	`[a-a=>a] [x-1=>X] [fx-f1=>fX]`	`[a-a=>a] [x-1=>X] [fx-f1=>fX]`
$lgg_{|_\sigma}(s_x, s_i)$	$\{p(a, fX)\}$	PAIRING REJECTED

As the examples demonstrate, the requirement that the matchings are both basic and comply with the lgg table is quite strong. The more structure examples have, the greater the reduction in possible pairings and hence queries is, since that structure needs to be matched. While it is not possible to quantify this effect without introducing further parameters, we expect this to be a considerable improvement in practice.

4 Proof of correctness

During the analysis, s will stand for the cardinality of P, the set of predicate symbols in the language; a for the maximal arity of the predicates in P; k for the maximum number of distinct variables in a clause of T; t for the maximum number of distinct terms in a clause of T; e_t for the maximum number of distinct terms in a counterexample; m for the number of clauses of the target expression T; m' for the number of clauses of the transformation of the target expression $U(T)$ as described in Section 2.2. Due to lack of space, some proofs have been reduced to simple sketches or even omitted. For a detailed account of the proof, see [AK00b]. Before starting with the proof, we give some definitions.

A multi-clause $[s, c]$ *covers* a clause $ineq(s_t), s_t \rightarrow b_t$ if there is a mapping θ from variables in s_t into terms in s such that $s_t \cdot \theta \subseteq s$ and $ineq(s_t) \cdot \theta \subseteq ineq(s)$. Equivalently, we say that $ineq(s_t), s_t \rightarrow b_t$ *is covered* by $[s, c]$.

A multi-clause $[s, c]$ *captures* a clause $ineq(s_t), s_t \rightarrow b_t$ if there is a mapping θ from variables in s_t into terms in s such that $ineq(s_t), s_t \rightarrow b_t$ is covered by $[s, c]$ via θ and $b_t \cdot \theta \in c$. Equivalently, we say that $ineq(s_t), s_t \rightarrow b_t$ *is captured by* $[s, c]$.

It is clear that if the algorithm stops, then the returned hypothesis is correct. Therefore the proof focuses on assuring that the algorithm finishes. To do so, a bound is established on the length of the sequence S. That is, only a finite number of counterexamples can be added to S and every refinement of an existing multi-clause reduces its size, and hence termination is guaranteed.

Lemma 1. *If $[s, c]$ is a positive example for a Horn expression T, then there is some clause $ineq(s_t), s_t \rightarrow b_t$ of $U(T)$ such that $s_t \cdot \theta \subseteq s$, $ineq(s_t) \cdot \theta \subseteq ineq(s)$ and $b_t \cdot \theta \notin s$, where θ is some substitution mapping variables of s_t into terms of s. That is, $ineq(s_t), s_t \rightarrow b_t$ is covered by $[s, c]$ via θ and $b_t \cdot \theta \notin s$.*

Proof. Consider the interpretation I whose objects are the different terms appearing in s plus an additional special object $*$. Let D_I be the set of objects in I. Let σ be the mapping from terms in s into objects in I. The function mappings in I are defined following σ, or $*$ when not specified. We want I to falsify the multi-clause $[s, c]$. Therefore, the extension of I, say $ext(I)$, includes exactly those literals in s (with the corresponding new names for the terms), that is, $ext(I) = s \cdot \sigma$, where the top-level terms in s are substituted by the image in D_I given by σ.

It is easy to see that this I falsifies $[s, c]$, because $s \cap c = \emptyset$ by definition of multi-clause. Since $I \not\models [s, c]$ and $T \models [s, c]$, we can conclude that $I \not\models T$. That is, there is a clause $C = s_c \rightarrow b_c$ in T such that $I \not\models C$ and there is a substitution θ' from variables in s_c into domain objects in I such that $s_c \cdot \theta' \subseteq ext(I)$ and $b_c \cdot \theta' \notin ext(I)$.

Complete the substitution θ' by adding all the remaining functional terms of C. The image that they are assigned to is their interpretation using the function mappings in I and the variable assignment θ'. When all terms have been included, consider the partition π induced by the completed θ', that is, two terms are included in the same class of the partition iff they are mapped to the same domain object by the completed θ'. Now, consider the clause $U_\pi(C)$. This clause is included in $U(T)$ because the classes are unifiable (the existence of $[s, c]$ is the proof for it) and therefore it is not rejected by the transformation procedure.

We claim that this clause $U_\pi(C)$ is the clause $ineq(s_t), s_t \rightarrow b_t$ mentioned in the lemma. Let $\hat{\theta}$ be the mgu used to obtain $U_\pi(C)$ from C with the partition π.

That is, $U_\pi(C) = ineq(s_c \cdot \hat{\theta}), s_c \cdot \hat{\theta} \to b_c \cdot \hat{\theta}$. Let θ'' be the substitution such that $\theta' = \hat{\theta} \cdot \theta''$. The substitution θ'' exists since $\hat{\theta}$ is a mgu and by construction θ' is also a unifier for every class in the partition. The clause $U_\pi(C) = ineq(s_t), s_t \to b_t$ is falsified using the substitution θ'': $s_c \cdot \theta' = s_c \cdot \hat{\theta} \cdot \theta'' = s_t \cdot \theta'' \subseteq ext(I)$, and $b_c \cdot \theta' = b_c \cdot \hat{\theta} \cdot \theta'' = b_t \cdot \theta'' \notin ext(I)$.

Now we have to find a θ for which the three conditions stated in the lemma are satisfied. We define θ as $\theta'' \cdot \sigma^{-1}$. Notice σ is invertible since all the elements in its range are different. It can also be composed to θ'' since all elements in the range of θ'' are in D_I, and the domain of σ consists precisely of all objects in D_I. Notice also that $s = ext(I) \cdot \sigma^{-1}$, and this can be done since the object $*$ does not appear in $ext(I)$. It is left to show that:

– $s_t \cdot \theta \subseteq s$: $s_t \cdot \theta'' \subseteq ext(I)$ implies $s_t \cdot \theta = s_t \cdot \theta'' \cdot \sigma^{-1} \subseteq ext(I) \cdot \sigma^{-1} = s$.

– $ineq(s_t) \cdot \theta \subseteq ineq(s)$. Take any two different terms t, t' of s_t. The inequality $t \neq t' \in ineq(s_t)$, since we have assumed they are different. The terms $t \cdot \theta, t' \cdot \theta$ appear in s, since $s_t \cdot \theta \subseteq s$. In order to be included in $ineq(s)$ they need to be different terms. Hence, we only need to show that the terms $t \cdot \theta, t' \cdot \theta$ are different terms. By way of contradiction, suppose they are not, i.e. $t \cdot \theta = t' \cdot \theta$, so that $t \cdot \theta'' \cdot \sigma^{-1} = t' \cdot \theta'' \cdot \sigma^{-1}$. The substitution σ^{-1} maps different objects into different terms, hence t and t' were mapped into the same domain object of I by θ''. Or equivalently, the terms t_c, t'_c of s_c for which $t = t_c \cdot \hat{\theta}$ and $t' = t'_c \cdot \hat{\theta}$ were mapped into the same domain object. But then they fall into the same class of the partition, hence they have the same representative in s_t and $t = t_c \cdot \hat{\theta} = t'_c \cdot \hat{\theta} = t'$, which contradicts our assumption that t and t' are different.

– $b_t \cdot \theta \notin s$: $b_t \cdot \theta'' \notin ext(I)$ implies $b_t \cdot \theta = b_t \cdot \theta'' \cdot \sigma^{-1} \notin ext(I) \cdot \sigma^{-1} = s$. ■

Lemma 1 implies that every full multi-clause w.r.t. T, say $[s, c]$, captures some clause in $U(T)$. This is because $[s, c]$ is full and $b_t \cdot \theta \notin s$, and hence $b_t \cdot \theta \in c$. Also, $rhs(s, c)$ cannot be empty since the correct $b_t \cdot \theta \in c$ must survive.

A multi-clause $[s, c]$ is a positive counterexample for some target expression T and some hypothesis H if $T \models [s, c]$, $c \neq \emptyset$ and for all literals $b \in c$, $H \not\models s \to b$. Notice that the hypothesis H is always entailed by the target T and therefore the equivalence oracle can only give positive counterexamples. This is because all multi-clauses in the sequence S are correct at any time.

Let $[s_x, c_x]$ be any minimised counterexample. Then, the multi-clause $[s_x, c_x]$ is full w.r.t. T and it is a positive counterexample w.r.t. target T and hypothesis H. Both properties can be proved by induction on the number of times $[s_x, c_x]$ is updated during the minimisation process.

Lemma 2. *Let $[s_x, c_x]$ be a multi-clause as generated by the minimisation procedure. If $[s_x, c_x]$ captures some clause $ineq(s_t), s_t \to b_t$ of $U(T)$, then it must be via some substitution θ such that θ is a variable renaming, i.e., θ maps distinct variables of s_t into distinct variables of s_x only.*

Proof. $[s_x, c_x]$ is capturing $ineq(s_t), s_t \to b_t$, hence there must exist a substitution θ from variables in s_t into terms in s_x such that $s_t \cdot \theta \subseteq s_x$, $ineq(s_t) \cdot \theta \subseteq ineq(s_x)$ and $b_t \cdot \theta \in c_x$. We will show that θ must be a variable renaming.

By way of contradiction, suppose that θ maps some variable v of s_t into a functional term t of s_x (i.e. $v \cdot \theta = t$). Consider the generalisation of the term t in step 3 of the minimisation procedure. We will see that the term t should have been generalised and substituted by the new variable x_t, contradicting the fact that the variable v was mapped into a functional term.

Let $\theta_t = \{t \mapsto x_t\}$ and $[s'_x, c'_x] = [s_x \cdot \theta_t, c_x \cdot \theta_t]$. Consider the substitution $\theta \cdot \theta_t$. We will see that $[s'_x, c'_x]$ captures $ineq(s_t), s_t \rightarrow b_t$ via $\theta \cdot \theta_t$ and hence $rhs(s'_x, c'_x) \neq \emptyset$ and therefore t must be generalised to the variable x_t. To see this we need to show:

– $s_t \cdot \theta \cdot \theta_t \subseteq s'_x$. By hypothesis $s_t \cdot \theta \subseteq s_x$ implies $s_t \cdot \theta \cdot \theta_t \subseteq s_x \cdot \theta_t = s'_x$.

– $ineq(s_t) \cdot \theta \cdot \theta_t \subseteq ineq(s'_x)$. Let t_1, t_2 two distinct terms of s_t. We have to show that $t_1 \cdot \theta \cdot \theta_t$ and $t_2 \cdot \theta \cdot \theta_t$ are two different terms of s'_x and therefore their inequality appears in $ineq(s'_x)$. It is easy to see that they are terms of s'_x since $s_t \cdot \theta \cdot \theta_t \subseteq s'_x$. To see that they are also different terms, notice first that $t_1 \cdot \theta$ and $t_2 \cdot \theta$ are different terms of s_x, since the clause $ineq(s_t), s_t \rightarrow b_t$ is captured by $[s_x, c_x]$. It is sufficient to show that if t'_1, t'_2 are any two distinct terms of s_x, then $t'_1 \cdot \theta_t$ and $t'_2 \cdot \theta_t$ are also distinct terms.

Notice the substitution θ_t maps the term t into a new variable x_t that does not appear in s_x. Consider the first position where t'_1 and t'_2 differ. Then, $t'_1 \cdot \theta_t$ and $t'_2 \cdot \theta_t$ will also differ in this same position, since at most one of the terms can contain t in that position. Therefore they also differ after applying θ_t.

– $b_t \cdot \theta \cdot \theta_t \in c'_x$. By hypothesis $b_t \cdot \theta \in c_x$ implies $b_t \cdot \theta \cdot \theta_t \in c_x \cdot \theta_t = c'_x$. ■

Another property satisfied by minimised multi-clauses is that the number of distinct terms appearing in a minimised multi-clause coincides with the number of distinct terms of any clause of $U(T)$ it captures. This is because the minimisation procedure detects and drops the superfluous term in s_x. Let t be the bound for the number of distinct terms in any clause of $U(T)$. Then, t bounds the number of distinct terms of any minimised multi-clause.

Let $[s_i, c_i]$ be any multi-clause covering a clause in $U(T)$. It can be shown that the number of distinct terms in s_i is greater or equal than the number of terms in the clause it covers. This happens because the substitution showing the covering of $[s_i, c_i]$ does not unify terms. We can conclude that if $[s_i, c_i]$ covers a clause captured by a minimised $[s_x, c_x]$, then s_x has no more terms than s_i.

It can be shown that if $[s_x, c_x]$ and $[s_i, c_i]$ are two full multi-clauses w.r.t. the target expression T, σ is a basic matching between the terms in s_x and s_i that is not rejected by the pairing procedure, and $[s, c]$ is the basic pairing of $[s_x, c_x]$ and $[s_i, c_i]$ induced by σ, then the multi-clause $[s, rhs(s, c)]$ is also full w.r.t. T. This, together with the fact that every minimised counterexample $[s_x, c_x]$ is full w.r.t. T implies that every multi-clause in the sequence S is full w.r.t. T, since its elements are constructed by initially being a full multi-clause and subsequently pairing full multi-clauses.

Lemma 3. *Let S be the sequence $[[s_1, c_1], [s_2, c_2], ..., [s_k, c_k]]$. If a minimised counterexample $[s_x, c_x]$ is produced such that it captures some clause in $U(T)$ $ineq(s_t), s_t \rightarrow b_t$ covered by some $[s_i, c_i]$ of S, then some multi-clause $[s_j, c_j]$ will be replaced by a basic pairing of $[s_x, c_x]$ and $[s_j, c_j]$, where $j \leq i$.*

Proof (Sketch). We will show that if no element $[s_j, c_j]$ where $j < i$ is replaced, then the element $[s_i, c_i]$ will be replaced. We have to prove that there is a basic pairing $[s, c]$ of $[s_x, c_x]$ and $[s_i, c_i]$ with the following two properties: $rhs(s, c) \neq \emptyset$ and $size(s) \lneqq size(s_i)$.

We have assumed that there is some clause $ineq(s_t), s_t \rightarrow b_t \in U(T)$ captured by $[s_x, c_x]$ and covered by $[s_i, c_i]$. Let θ'_x be the substitution showing $ineq(s_t), s_t \rightarrow b_t$ being captured by $[s_x, c_x]$ and θ'_i the substitution showing $ineq(s_t), s_t \rightarrow b_t$ being covered by $[s_i, c_i]$. Thus the following holds: $s_t \cdot \theta'_x \subseteq s_x$; $ineq(s_t) \cdot \theta'_x \subseteq ineq(s_x)$; $b_t \cdot \theta'_x \in c_x$ and $s_t \cdot \theta'_i \subseteq s_i$; $ineq(s_t) \cdot \theta'_i \subseteq ineq(s_i)$.

We construct a matching σ that includes all entries `[`$t \cdot \theta'_x$ `-` $t \cdot \theta'_i$ `=>` $lgg(t \cdot \theta'_x, t \cdot \theta'_i)$`]` such that t is a term appearing in s_t (only one entry for every distinct term of s_t).

Example 8. Let s_t be $\{p(g(c), x, f(y), z)\}$ containing 6 terms: c, $g(c)$, x, y, $f(y)$ and z. Let s_x be $\{p(g(c), x', f(y'), z), p(g(c), g(c), f(y'), c)\}$, containing 6 terms: c, $g(c)$, x', y', $f(y')$, z. Let s_i be $\{p(g(c), f(1), f(f(2)), z)\}$, containing 8 terms: c, $g(c)$, 1, $f(1)$, 2, $f(2)$, $f(f(2))$, z. The substitution θ'_x is $\{x \mapsto x', y \mapsto y', z \mapsto z\}$ and it is a variable renaming. The substitution θ'_i is $\{x \mapsto f(1), y \mapsto f(2), z \mapsto z\}$.

The $lgg(s_x, s_i)$ is $\{p(g(c), X, f(Y), z), p(g(c), Z, f(Y), V)\}$ and it produces the following lgg table.

```
[c - c => c]        [g(c) - g(c) => g(c)]     [x' - f(1) => X]
[y' - f(2) => Y]    [f(y') - f(f(2)) => f(Y)] [z - z => z]
[g(c) - f(1) => Z] [c - z => V]
```

The $lgg_{|_\sigma}(s_x, s_i)$ is $\{p(g(c), X, f(Y), z)\}$ and the extended matching σ is

$c \Rightarrow$ `[c - c => c]` $\quad g(c) \Rightarrow$ `[g(c) - g(c) => g(c)])`
$x \Rightarrow$ `[x' - f(1) => X]` $\quad y \Rightarrow$ `[y' - f(2) => Y]`
$f(y) \Rightarrow$ `[f(y') - f(f(2)) => f(Y)]` $\quad z \Rightarrow$ `[z - z => z]`

The matching σ as described above is 1-1 and it is not discarded by the pairing procedure. Moreover, the number of entries in σ equals the minimum of the number of distinct terms in s_x and s_i. Let $[s, c]$ denote the pairing of $[s_x, c_x]$ and $[s_i, c_i]$ induced by σ.

We first claim that the matching σ is legal and basic. This can be shown by induction on the structure of the terms in s_t that induce every entry in σ. The induction hypothesis is the following. If t is a term in s_t, then the term $lgg(t \cdot \theta'_x, t \cdot \theta'_i)$ and all its subterms appear in the extension of some other entries of σ. The induction uses the fact that s_x contains a variant of s_t. Thus σ is considered by the algorithm.

Next, we argue that the conditions $rhs(s, c) \neq \emptyset$ and $size(s) \lneqq size(s_i)$ hold. Let θ be the substitution that maps all variables in s_t to their corresponding expression assigned in the extension of σ. That is, θ maps any variable v of s_t to the term $lgg(v \cdot \theta'_x, v \cdot \theta'_i)$. In our example, $\theta = \{x \mapsto X, y \mapsto Y, z \mapsto z\}$. The proof of $rhs(s, c) \neq \emptyset$ consists in showing that $[s, c]$ captures $ineq(s_t), s_t \rightarrow b_t$ via θ.

The use of 1-1 matchings in pairings guarantees that the number of literals in the antecedent of a pairing never exceeds that of the multi-clauses originating it, since at most one copy of every atom in the original multi-clauses is included in the pairing. Thus, $size(s) \leq size(s_i)$. To see that the relation is strict, consider the literal $b_t \cdot \theta'_i$. The proof is by cases. If $b_t \cdot \theta'_i \in s_i$, then the size of s must be smaller than that of s_i because its counterpart in s ($b_t \cdot \theta$) does not appear in s and the lgg never substitutes a term by one of greater size. If $b_t \cdot \theta'_i \notin s_i$, then either s_i contains an extra literal (thus $size(s) \lneq size(s_i)$), or $[s_x, c_x]$ can not be a counterexample. ■

As a direct consequence, we obtain that whenever a counterexample is appended to the end of the sequence S, it is because there is no other element in S capturing a clause in $U(T)$ that is also captured by $[s_x, c_x]$.

Lemma 4. *Let $[s_1, c_1]$ and $[s_2, c_2]$ be two full multi-clauses. Let $[s, c]$ be any legal pairing between them. If $[s, c]$ captures a clause $ineq(s_t), s_t \to b_t$, then the following holds:*

1. *Both $[s_1, c_1]$ and $[s_2, c_2]$ cover $ineq(s_t), s_t \to b_t$.*
2. *At least one of $[s_1, c_1]$ or $[s_2, c_2]$ captures $ineq(s_t), s_t \to b_t$.*

Proof. By assumption, $ineq(s_t), s_t \to b_t$ is captured by $[s, c]$, i.e., there is a θ such that $s_t \cdot \theta \subseteq s$, $ineq(s_t) \cdot \theta \subseteq ineq(s)$ and $b_t \cdot \theta \in c$. This implies that if t, t' are two distinct terms of s_t, then $t \cdot \theta$ and $t' \cdot \theta$ are distinct terms appearing in s. Let σ be the 1-1 legal matching inducing the pairing. The antecedent s is defined to be $lgg_{|\sigma}(s_1, s_2)$, and therefore there exist substitutions θ_1 and θ_2 such that $s \cdot \theta_1 \subseteq s_1$ and $s \cdot \theta_2 \subseteq s_2$.

Condition 1. We claim that $[s_1, c_1]$ and $[s_2, c_2]$ cover $ineq(s_t), s_t \to b_t$ via $\theta \cdot \theta_1$ and $\theta \cdot \theta_2$, respectively. Notice that $s_t \cdot \theta \subseteq s$, and therefore $s_t \cdot \theta \cdot \theta_1 \subseteq s \cdot \theta_1$. Since $s \cdot \theta_1 \subseteq s_1$, we obtain $s_t \cdot \theta \cdot \theta_1 \subseteq s_1$. The same holds for s_2. It remains to show that $ineq(s_t) \cdot \theta \cdot \theta_1 \subseteq ineq(s_1)$ and $ineq(s_t) \cdot \theta \cdot \theta_2 \subseteq ineq(s_2)$. Observe that all top-level terms appearing in s also appear as one entry of the matching σ, because otherwise they could not have survived. Further, since σ is legal, all subterms of terms of s also appear as an entry in σ. Let t, t' be any distinct terms appearing in s_t. Since $s_t \cdot \theta \subseteq s$ and σ includes all terms appearing in s, the distinct terms $t \cdot \theta$ and $t' \cdot \theta$ appear as the lgg of distinct entries in σ. These entries have the form `[`$t \cdot \theta \cdot \theta_1$ `-` $t \cdot \theta \cdot \theta_2$ `=>` $t \cdot \theta$`]`, since $lgg(t \cdot \theta \cdot \theta_1, t \cdot \theta \cdot \theta_2) = t \cdot \theta$. Since σ is 1-1, we know that $t \cdot \theta \cdot \theta_1 \neq t' \cdot \theta \cdot \theta_1$ and $t \cdot \theta \cdot \theta_2 \neq t' \cdot \theta \cdot \theta_2$.

Condition 2. By hypothesis, $b_t \cdot \theta \in c$ and c is defined to be $lgg_{|\sigma}(s_1, c_2) \cup lgg_{|\sigma}(c_1, s_2) \cup lgg_{|\sigma}(c_1, c_2)$. Observe that all these lggs share the same table, so the same pairs of terms will be mapped into the same expressions. Observe also that the substitutions θ_1 and θ_2 are defined according to this table, so that if any literal $l \in lgg_{|\sigma}(c_1, \cdot)$, then $l \cdot \theta_1 \in c_1$. Equivalently, if $l \in lgg_{|\sigma}(\cdot, c_2)$, then $l \cdot \theta_2 \in c_2$. Therefore we get that if $b_t \cdot \theta \in lgg_{|\sigma}(c_1, \cdot)$, then $b_t \cdot \theta \cdot \theta_1 \in c_1$ and if $b_t \cdot \theta \in lgg_{|\sigma}(\cdot, c_2)$, then $b_t \cdot \theta \cdot \theta_2 \in c_2$. Now, observe that in any of the three possibilities for c, one of c_1 or c_2 is included in the $lgg_{|\sigma}$. Thus it is the case that either $b_t \cdot \theta \cdot \theta_1 \in c_1$ or $b_t \cdot \theta \cdot \theta_2 \in c_2$. Since both $[s_1, c_1]$ and $[s_2, c_2]$ cover $ineq(s_t), s_t \to b_t$, one of $[s_1, c_1]$ or $[s_2, c_2]$ captures $ineq(s_t), s_t \to b_t$. ■

It is crucial for Lemma 4 that the pairing involved is *legal*. It is indeed possible for a *non-legal* pairing to capture some clause that is not even covered by some of its originating multi-clauses.

Lemma 5. *Every time the algorithm is about to make an equivalence query, it is the case that every multi-clause in S captures at least one of the clauses of $U(T)$ and every clause of $U(T)$ is captured by at most one multi-clause in S.*

Proof (Sketch). All counterexamples included in S are full positive multi-clauses. Therefore, every $[s, c]$ in S captures some clause of $U(T)$. An induction on the number of iterations of the main loop in line 2 of the learning algorithm shows that no two different multi-clauses in S capture the same clause of $U(T)$. ■

Recall that m' stands for the number of clauses in the transformation $U(T)$. Lemma 5 provides us with the bound m' on $|S|$ required to guarantee termination of the algorithm. Counting carefully the number of queries made in every procedure we arrive to our main result.

Theorem 1. *The algorithm exactly identifies range restricted Horn expressions making $O(m'st^a)$ equivalence queries and $O(m's^2t^ae_t^{a+1} + m'^2s^2t^{2a+k})$ membership queries. The running time is polynomial in the number of membership queries.*

5 Fully Inequated Range Restricted Horn Expressions

Clauses of this class can contain a new type of literal, that we call *inequation* or *inequality* and has the form $t \neq t'$, where both t and t' are terms. Inequated clauses may contain any number of inequalities in its antecedent. Let s be any conjunction of atoms and inequations. Then, s^p denotes the conjunction of atoms in s and $s^{\neq}$ the conjunction of inequalities in s. That is $s = s^p \wedge s^{\neq}$. We say s is *completely inequated* if $s^{\neq}$ contains all possible inequations between distinct terms in s^p, i.e., if $s^{\neq} = ineq(s^p)$. A clause $s \to b$ is completely inequated iff s is. A multi-clause $[s, c]$ is completely inequated iff s is. A *fully inequated range restricted Horn expression* is a conjunction of fully inequated range restricted clauses.

Looking at the way the transformation $U(T)$ described in Section 2.2 is used in the proof of correctness, the natural question of what happens when the target expression is already fully inequated (and $T = U(T)$) arises. We will see that the algorithm presented in Figure 2 has to be slightly modified in order to achieve learnability of this class.

The first modification is in the minimisation procedure. It can be the case that after generalising or dropping some terms (as it is done in the two stages of the minimisation procedure), the result of the operation is not fully inequated. More precisely, there may be superfluous inequalities that involve terms not appearing in the atoms of the counterexample's antecedent. These should be eliminated from the counterexample, yielding a fully inequated minimised counterexample.

Given a matching σ and two multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$, its pairing $[s, c]$ is computed in the new algorithm as: $s = ineq(lgg_{|\sigma}(s_x^p, s_i^p)) \cup lgg_{|\sigma}(s_x^p, s_i^p)$ and $c = lgg_{|\sigma}(s_x^p, c_i) \cup lgg_{|\sigma}(c_x, s_i^p) \cup lgg_{|\sigma}(c_x, c_i)$. Notice that inequations in the antecedents are ignored. The pairing is computed only for the atomic information, and finally the fully inequated pairing is constructed by adding all the inequations needed. This can be done safely because the algorithm only deals with fully inequated clauses. The proof of correctness is very similar to the one presented here. Complete details and proof can be found in [AK00a].

Theorem 2. *The modified algorithm exactly identifies fully inequated range restricted Horn expressions making $O(mst^a)$ calls to the equivalence oracle and $O(ms^2t^ae_t^{a+1} + m^2s^2t^{2a+k})$ to the membership oracle. The running time is polynomial in the number of membership queries.*

6 Conclusions

The paper introduced a new algorithm for learning range restricted Horn expressions ($RRHE$) and established the learnability of fully inequated range restricted Horn expressions ($FIRRHE$). The structure of the algorithm is similar to previous ones, but it uses carefully chosen operations that take advantage of the structure of functional terms in examples. This in turn leads to an improvement of worst case bounds on the number of queries required, which is one of the main contributions of the paper. The following table contains the results obtained in [Kha99b] and in this paper.

	Class	$EntEQ$	$EntMQ$
Result in [Kha99b]	$RRHE$	$O(mst^{t+a})$	$O(ms^2t^{t+a}e_t^{a+1} + m^2s^2t^{3t+2a})$
Our result	$RRHE$	$O(mst^{t+a})$	$O(ms^2t^{t+a}e_t^{a+1} + m^2s^2t^{2t+k+2a})$
Our result	$FIRRHE$	$O(mst^a)$	$O(ms^2t^ae_t^{a+1} + m^2s^2t^{2a+k})$

Note that for $RRHE$ the exponential dependence on the number of terms is reduced from t^{3t} to t^{2t+k}. A more dramatic improvement is achieved for $FIRRHE$ where the comparable factor is t^k so that the algorithm is polynomial in the number of terms t though still exponential in the number of variables k. This may be significant as in many cases, while inequalities are not explicitly written, the intention is that different terms denote different objects. The reduction in the number of queries goes beyond worst case bounds. The restriction that pairings are both basic *and* agree with the lgg table is quite strong and reduces the number of pairings and hence queries. This is not reflected in our analysis but we believe it will make a difference in practice. Similarly, the bound $m' \leq mt^t$ on $|U(T)|$ is quite loose, as a large proportion of partitions will be discarded if T includes functional structure.

Finally, the use of lgg results in a more direct and natural algorithm. Moreover, it may help understand the relations between previous algorithms based on lgg [Ari97,RT98,RS98] and algorithms based on direct products [Kha99a]. We hope that this can lead to further understanding and better algorithms for the problem.

References

[AK00a] M. Arias and R. Khardon. Learning Inequated Range Restricted Horn Expressions. Technical Report EDI-INF-RR-0011, Division of Informatics, University of Edinburgh, March 2000.

[AK00b] M. Arias and R. Khardon. A New Algorithm for Learning Range Restricted Horn Expressions. Technical Report EDI-INF-RR-0010, Division of Informatics, University of Edinburgh, March 2000.

[Ari97] Hiroki Arimura. Learning acyclic first-order Horn sentences from entailment. In *Proceedings of the International Conference on ALT*, Sendai, Japan, 1997. Springer-Verlag. LNAI 1316.

[Coh95a] W. Cohen. PAC-learning recursive logic programs: Efficient algorithms. *Journal of Artificial Intelligence Research*, 2:501–539, 1995.

[Coh95b] W. Cohen. PAC-learning recursive logic programs: Negative results. *Journal of Artificial Intelligence Research*, 2:541–573, 1995.

[FP93] M. Frazier and L. Pitt. Learning from entailment: An application to propositional Horn sentences. In *Proceedings of the International Conference on Machine Learning*, pages 120–127, Amherst, MA, 1993. Morgan Kaufmann.

[Kha99a] R. Khardon. Learning function free Horn expressions. *Machine Learning*, 37:241–275, 1999.

[Kha99b] R. Khardon. Learning range restricted Horn expressions. In *Proceedings of the Fourth European Conference on Computational Learning Theory*, pages 111–125, Nordkirchen, Germany, 1999. Springer-verlag. LNAI 1572.

[Kha00] Roni Khardon. Learning horn expressions with LOGAN-H. To appear in ICML, 2000.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1987.

[MF92] S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.

[MR94] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19 & 20:629–680, May 1994.

[Plo70] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[RB92] L. De Raedt and M. Bruynooghe. An overview of the interactive concept-learner and theory revisor CLINT. In S. Muggleton, editor, *Inductive Logic Programming*, pages 163–192. Academic Press, 1992.

[RS98] K. Rao and A. Sattar. Learning from entailment of logic programs with local variables. In *Proceedings of the International Conference on Algorithmic Learning Theory*, Otzenhausen, Germany, 1998. Springer-verlag. LNAI 1501.

[RT98] C. Reddy and P. Tadepalli. Learning first order acyclic Horn programs from entailment. In *International Conference on Inductive Logic Programming*, pages 23–37, Madison, WI, 1998. Springer. LNAI 1446.

[SEMF98] G. Semeraro, F. Esposito, D. Malerba, and N. Fanizzi. A logic framework for the incremental inductive synthesis of datalog theories. In *Proceedings of the International Conference on Logic Program Synthesis and Transformation (LOPSTR'97)*. Springer-Verlag, 1998. LNAI 1463.

[Sha83] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.