Division of Informatics, University of Edinburgh

Laboratory for Foundations of Computer Science

A New Algorithm for Learning Range Restricted Horn Expressions

by

Marta Arias, Roni Khardon

Informatics Research Report EDI-INF-RR-0010

Division of Informatics **March 2000**

http://www.informatics.ed.ac.uk/

A New Algorithm for Learning Range Restricted Horn Expressions

Marta Arias, Roni Khardon

Informatics Research Report EDI-INF-RR-0010

DIVISION *of* INFORMATICS
Laboratory for Foundations of Computer Science
March 2000

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 further improvement in the number of queries is obtained.

Keywords : computational learning theory, Horn expressions, least general generalisation, learning from entailment

A New Algorithm for Learning Range Restricted Horn Expressions*

Marta Arias
Division of Informatics
University of Edinburgh
marta@dcs.ed.ac.uk

Roni Khardon
Division of Informatics
University of Edinburgh
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 further 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*. A great deal of work has been done in this learning setting. For example, [FP93] formalised learning from entailment using equivalence queries and membership queries in the study of 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 the field of *Inductive Logic Programming* (see [MR94] for a survey). This field has produced several systems that are able to learn in the first order setting using equivalence and membership entailment queries. Among these are, for example, MIS [Sha83, Sha91] and CLINT [DRB92].

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 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 learnability of the class of range restricted Horn expressions was solved in [Kha99b] by reducing it to the case of function free Horn expressions, already 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 to be a more natural and intuitive technique to use for learning from entailment, and it has been used

*This work was partly supported by EPSRC Grant GR/M21409.

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

before, both in theoretical and applied work [Ari97, RT98, RS98, MF92]. Thus the contributions of this paper are to give a more direct algorithm for the class and establish better bounds in terms of running time and number of queries to the oracles.

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 then 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.

2 Preliminaries

2.1 Range Restricted Horn Expressions

We consider a subset of the class of universally quantified expressions in first order logic. The learning problem assumes a pre-fixed known and finite signature of the language. This signature $\mathcal{S}$ consists of a finite set of predicates P and a finite set of functions F, both predicates and functions with their associated arity. Constants are functions with arity 0. A set of variables $x_1, x_2, x_3, ...$ is used to construct expressions.

Definitions of first order languages can be found in standard texts ([Llo87]). Here we briefly introduce the necessary constructs. A variable is a *term* of depth 0. If $t_1, ..., t_n$ are terms, each of depth at most i and one with depth precisely i and $f \in F$ is a function symbol of arity n, then $f(t_1, ..., t_n)$ is a term of depth $i + 1$.

An *atom* is an expression $p(t_1, ..., t_n)$ where $p \in P$ is a predicate symbol of arity n and $t_1, ..., t_n$ are terms. An atom is called a *positive literal.* A *negative literal* is an expression $\neg l$ where l is a positive literal.

A *clause* is a disjunction of literals where all variables are taken to be universally quantified. A *Horn clause* has at most one positive literal and an arbitrary number of negative literals. A Horn clause $\neg p_1 \vee ... \vee \neg p_n \vee p_{n+1}$ is equivalent to its implicational form $p_1 \wedge ... \wedge p_n \rightarrow p_{n+1}$. We call $p_1 \wedge ... \wedge p_n$ the *antecedent* and p_{n+1} the *consequent* of the clause.

A Horn clause is said to be *definite* if it has exactly one positive literal. 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.

The truth value of first order expressions is defined relative to an interpretation I of the predicates and function symbols in the signature $\mathcal{S}$. An *interpretation*[2] I includes a domain D which is a finite set of elements. For each function $f \in F$ of arity n, I associates a mapping from D^n to D. For each predicate symbol $p \in P$ of arity n, I specifies the truth value of p on n-tuples over D. The *extension* of a predicate in I is the set of positive instantiations of it that are true in I.

Let p be an atom, I an interpretation and θ a mapping of the variables in p to objects in I. The ground positive literal $p \cdot \theta$ is true in I iff it appears in the extension of I. A ground negative literal is true in I iff its negation is not.

[2] Also called *structure* or *model.*

A Horn clause $C = p_1 \wedge ... \wedge p_n \rightarrow p_{n+1}$ is true in a given interpretation I, denoted $I \models C$ iff for any variable assignment θ (a total function from the variables in C into the domain elements of I), if all the literals in the antecedent $p_1\theta, ..., p_n\theta$ are true in I, then the consequent $p_{n+1}\theta$ is also true in I. A Horn Expression T is true in I, denoted $I \models T$, if all of its clauses are true in I. The expressions T is true in I, I satisfies T, I is a model of T, and $I \models T$ are equivalent.

Let T_1, T_2 be two Horn expressions. We say that T_1 implies T_2, denoted $T_1 \models T_2$, if every model of T_1 is also a model of T_2.

2.2 The Learning Model

In this paper we consider the model of *exact learning from entailment*, that was formalised by [FP93] in the propositional setting. 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.

2.3 Some definitions

Definition 1 (Multi-clause) 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 $[s_c, \{b_c\}]$.

Definition 2 (Implication relation) 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 $T \models \bigwedge_{b \in c} s \rightarrow b$.

Definition 3 (Correct multi-clause) A multi-clause $[s, c]$ is said to be *correct* w.r.t an expression T if for every literal $b \in c$, $T \models s \rightarrow b$. That is, $T \models [s, c]$.

Definition 4 (Exhaustive multi-clause) A multi-clause $[s, c]$ is said to be *exhaustive* w.r.t an expression T if every literal b such that $T \models s \rightarrow b$ is included in c.

Definition 5 (Full multi-clause) A multi-clause is said to be *full* w.r.t an expression T if it is correct and exhaustive w.r.t. T.

Definition 6 (Size of a multi-clause) The *size* of a multi-clause is defined as:

$$size([s, c]) = |s| + variables(s) + 2 \cdot functions(s),$$

where $|\cdot|$ refers to the number of literals, $variables(\cdot)$ to the number of occurrences of variables and $functions(\cdot)$ to the number of occurrences of functions symbols.

2.4 Most General Unifier

Definition 7 (Unifier) Let Σ be a finite set of expressions. A substitution θ is called a *unifier* for Σ if $\Sigma \cdot \theta$ is a singleton. If there exists a unifier for Σ, we say that Σ is *unifiable*. The only expression in $\Sigma \cdot \theta$ will also be called a *unifier*.

Definition 8 (Most General Unifier) The substitution θ is a *most general unifier* (abbreviated to *mgu*) for Σ if θ is a unifier for Σ and if for any other unifier σ there is a substitution γ such that $\sigma = \theta\gamma$. Also, the only element in $\Sigma \cdot \theta$ will be called a *mgu* of Σ if θ is a *mgu*.

Definition 9 (Disagreement Set) Let Σ be a finite set of expressions. The *disagreement set* of Σ is defined as follows. Locate the leftmost symbol position at which not all members of Σ have the same symbol, and extract from each expression in Σ the subexpression beginning at that symbol position. The set of all these expressions is the disagreement set.

Example 1 $\Sigma = \{p(x, \underline{y}, v), p(x, \underline{f(g(a))}, x), p(x, \underline{f(z)}, f(a))\}$. The disagreement set of Σ appears underlined.

```
1. Let $\Sigma$ be the set of expressions to be unified.
2. Set $k$ to 0 and $\sigma_0$ to $\emptyset$, the empty substitution.
3. Repeat until $\Sigma \cdot \sigma_k$ is a singleton
    (a) Let $D_k$ be the disagreement set for $\Sigma \cdot \sigma_k$.
    (b) If there exists $x$ and $t$ in $D_k$ such that $x$ is a variable not occurring in $t$,
        then set $\sigma_{k+1} = \sigma_k \cdot \{x \mapsto t\}$.
    (c) Otherwise, report that $\Sigma$ is not unifiable and stop.
4. Return $\sigma_k$.
```

Figure 1: The unification algorithm

Theorem 1 (Unification Theorem) *Let Σ be a finite set of expressions. If Σ is unifiable, then the Unification Algorithm terminates and gives a mgu for Σ. If Σ is not unifiable, then the Unification Algorithm terminates and reports the fact that Σ is not unifiable.*

Proof. The unification algorithm is described in Figure 1. See [Llo87] for the proof. ■

2.5 Least General Generalisation

Definition 10 (Subsumption) 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.

Definition 11 (Selection) A *selection* of two sets of literals s_1 and s_2 is a pair of literals (l_1, l_2) such that $l_1 \in s_1$, $l_2 \in s_2$, and both l_1 and l_2 have the same predicate symbol, arity and sign.

Definition 12 (Least General Generalisation) Let s, s', s_1, s_2 be clauses. We say that s is the *least general generalisation* (*lgg*) of s_1 and s_2 if and only if s subsumes both s_1 and s_2, and if there is any other clause s' subsuming both s_1 and s_2, then s' also subsumes s.

Plotkin proved in [Plo70] that the *lgg* of any two sets of literals exists if and only if they have a selection. Moreover, he gave an algorithm to find it and proved its correctness. The algorithm appears in Figure 2.

The computation of the *lgg* generates a table that given two terms, each appearing in one of the input sets of literals, determines the term to which that pair of terms will be generalised.

Example 2 As an example, consider the following two sets. The symbols $a, b, c, 1, 2$ stand for constants, f is a unary function, g is a binary function, x, z are variables and p, q are predicate symbols of arity 2 and 1, respectively.

- *If s_1 and s_2 are sets of literals,*
$$lgg(s_1, s_2) = \{lgg(l_1, l_2) \mid (l_1, l_2) \textit{ is a selection of } s_1 \textit{ and } s_2\}$$
- *If p is a predicate of arity n,*
$$lgg(p(s_1, ..., s_n), p(t_1, ..., t_n)) = p(lgg(s_1, t_1), ..., lgg(s_n, t_n))$$
- *If $f(s_1, ..., s_n)$ and $g(t_1, ..., t_m)$ are two terms,*
$$lgg(f(s_1, ..., s_n), g(t_1, ..., t_m)) = f(lgg(s_1, t_1), ..., lgg(s_n, t_n))$$
if $f = g$ and $n = m$. Else, 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.

Figure 2: The *lgg* algorithm

- $s_1 = \{p(a, f(b)), p(g(a, x), c), q(a)\}$
- $s_2 = \{p(z, f(2)), q(z)\}$
- We compute $lgg(s_1, s_2)$:
  - Selection: $p(a, f(b))$ with $p(z, f(2))$.
    - The terms $a - z$ generate entry `[a - z => X]`.
    - The terms $f(b) - f(2)$ generate entries `[b - 2 => Y]`, `[f(b) - f(2) => f(Y)]`.
  - Selection: $p(g(a, x), c)$ with $p(z, f(2))$.
    - The terms $g(a, x) - z$ generate entry `[g(a,x) - z => Z]`.
    - The terms $c - f(2)$ generate entry `[c - f(2) => V]`.
  - Selection: $q(a)$ with $q(z)$.
    - The terms $a - z$ appear already as an entry of the table, therefore no new entry is generated.
- $lgg(s_1, s_2) = \{p(X, f(Y)), p(Z, V), q(X)\}$
- The *lgg* table for it is
```
[a - z => X]
[b - 2 => Y]
[f(b) - f(2) => f(Y)]
[g(a,x) - z => Z]
[c - f(2) => V]
```

2.6 Transforming the target expression

In this section we describe the transformation $U(T)$ performed on any target expression T. It is very similar to the transformation described in [Kha99a] and it serves similar purposes. This transformation is never computed by the learning algorithm, it is only used in the analysis of the proof of correctness.

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 the difference that in every clause in $U(C)$ some terms of C have been unified in a certain way, different for every clause 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. The set $U(C)$ covers all possible ways of unifying terms of the original clause C.

Definition 13 (Function $ineq(\cdot)$) Let s be any set of literals. Then $ineq(s)$ is the set of all inequalities between terms appearing in s.

Example 3 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.

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 $U(C) = \bigwedge_{\pi \in Partitions(\mathcal{T})} U_\pi(C)$.

To compute the clause $U_\pi(C)$, take the partition and order its classes in any way. 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[3] 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 $U_\pi(C)$. This process is described in Figure 3.

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

Figure 3: The transformation algorithm

Example 4 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	$\{x\}$	$\emptyset$	$\emptyset$	$\{y\}, \{z\}, \{f(x)\}, \{f(y)\}, \{g(z)\}$
2	$\{y\}$	$\emptyset$	$\emptyset$	$\{z\}, \{f(x)\}, \{f(y)\}, \{g(z)\}$
3	$\{z\}$	$\emptyset$	$\emptyset$	$\{f(x)\}, \{f(y)\}, \{g(z)\}$
4	$\{f(x)\}$	$\emptyset$	$\emptyset$	$\{f(y)\}, \{g(z)\}$
5	$\{f(y)\}$	$\emptyset$	$\emptyset$	$\{g(z)\}$
6	$\{g(z)\}$	$\emptyset$	$\emptyset$	

[3]We call the representative of a class any of its elements applied to the unifying substitution (*mgu*).

$C \cdot \sigma_6 = p(f(x), f(y), g(z)) \rightarrow q(x, y, z)$.

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

- 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	$\{g(z)\}$	$\emptyset$	$\{y \mapsto x\}$	

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

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

- 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*		

Note that this partition is not a good one because it is not possible to unify $f(\cdot)$ with $g(\cdot)$, which reflects the fact that the expressions $f(\cdot)$ and $g(\cdot)$ could not possibly be syntactically equivalent (which is the effect of including two different terms into a same class in the partition).

- 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	$\{x, y, z\}$	$\{y, z \mapsto x\}$	$\{y, z \mapsto x\}$	$\{f(x)\}, \{f(x)\}, \{g(x)\}$

The reason why we discard partitions in which there is a term (obtained after applying the various unifying substitutions) appearing in at least two different classes, is because the idea behind the partitions is that the elements belonging to a same equivalence class, will be unified and will be distinct to every element in any other class. When one literal happens to be in two distinct classes, the inequality between the two will never be satisfied, and the resulting clause's antecedent will be never satisfied. Such clauses do not provide any information. And hence, can be ignored.

- 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	$\{x, y, z\}$	$\{y, z \mapsto x\}$	$\{y, z \mapsto x\}$	$\{f(x), f(x)\}, \{g(x)\}$
2	$\{f(x), f(x)\}$	$\emptyset$	$\{y, z \mapsto x\}$	$\{g(x)\}$
3	$\{g(z)\}$	$\emptyset$	$\{y, z \mapsto x\}$	

$C \cdot \sigma_3 = p(f(x), f(x), g(x)) \rightarrow q(x, x, x)$.

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

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

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

Looking at the previous example we can draw the following conclusions. First, no clause will be generated for those partitions containing some class with two functional terms with different top-level function symbol (since no pair of such terms is unifiable). And also, no clause will be generated for those partitions containing some class with two terms such that one is a subterm of the other (since no pair of such terms is unifiable).

This results in an important restriction on the total number of clauses of the transformation, since many partitions are discarded. However, we will use the number of all possible partitions as an upper bound of the number of clauses of the transformation of the target expression. Namely, 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. This is because the number of partitions of a set with t elements is bounded by t^t. And any of the m clauses in T can produce at most t^t clauses for $U(T)$.

Lemma 2 *Let C be any range restricted Horn clause and π any partition of its terms that has not been discarded according to the unifying method applied to the classes of π. Then, the clause $U_\pi(C)$ is also range restricted.*

Proof. Let $|\pi|$ be l. Consider the clause $C \cdot \sigma_l$ as computed in the procedure described in Figure 3. We claim is that this clause $C \cdot \sigma_l$ is range restricted. All the terms appearing in $C \cdot \sigma_l$'s consequent have the form $t \cdot \sigma_l$, where t is some term in C's antecedent. Since C is range restricted, t also appears in C's antecedent, and hence, $t \cdot \sigma_l$ must also appear in $C \cdot \sigma_l$'s antecedent. Therefore, $C \cdot \sigma_l$ is range restricted. And $U_\pi(C)$ is also range restricted, since they only differ in that $U_\pi(C)$ has some more inequality literals in the antecedent than $C \cdot \sigma_l$. ■

Lemma 3 $T \models U(T)$.

Proof. To see this, it suffices to notice that every clause in $U(T)$ is subsumed by the clause in T that originated it. Therefore, the implication applies. ■

Corollary 4 *If $U(T) \models C$, then $T \models C$. Also, if $U(T) \models [s, c]$, then $T \models [s, c]$.*

Proof. Suppose $U(T) \models C$. By Lemma 3, $T \models U(T)$. It follows that $T \models U(T) \models C$ and therefore $T \models C$ as required. Since any multi-clause $[s, c]$ can be split into its individual clauses, i.e. $\{s \to b \mid b \in c\}$, the second result follows. ■

However, the inverse implication $U(T) \models T$ of Lemma 3 does not hold. To see this, consider the following example.

Example 5 We present an expression T, its transformation $U(T)$ and an interpretation I such that $I \models U(T)$ but $I \not\models T$.

- $T = \{p(a, f(a)) \to q(a)\}$.
- $U(T) = \{(a \neq f(a)), p(a, f(a)) \to q(a)\}$.
- Interpretation I: $D_I = \{1\}$; constant $a = 1$; function $f(1) = 1$; $ext(I) = \{p(1, 1)\}$.
- $I \not\models T$ because $p(a, f(a))^{under\ I} = p(1, 1) \in ext(I)$ but $q(a)^{under\ I} = q(1) \notin ext(I)$.
- $I \models U(T)$ because inequality $(a \neq f(a))^{under\ I} = (1 \neq 1)$ is $false$ and therefore the antecedent of the clause is falsified. Hence, the clause is satisfied.

3 The Algorithm

```
1. Set $S$ to be the empty sequence.
2. Set $H$ to be the empty hypothesis.
3. 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 $\bigwedge_{[s,c] \in S} \{s \rightarrow b \mid b \in c\}$.
4. Return $H$
```

Figure 4: The learning 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 order to get a more accurate hypothesis in each iteration of the main loop, line 3, until hypothesis and target expressions coincide.

There are two basic operations on counterexamples that need to be explained in detail. These are *minimisation* (line 3b), that takes a counterexample as given by the equivalence oracle and produces a positive, full counterexample. And *pairing* (line 3c), that takes two counterexamples and generates a series of candidate counterexamples. The counterexamples obtained by combination of previous ones are the candidates to refine the sequence S. These operations are carefully explained in the following sections 3.1 and 3.2.

The basic structure handled by the algorithm is the full multi-clause w.r.t. the target expression T. All counterexamples take the form of a full multi-clause. Although the equivalence oracle does not produce a counterexample in this form, it is converted by calling the procedure *rhs*. This happens during the minimisation procedure.

Given a set s of ground literals, its corresponding set c of consequents can be easily found using the $EntMQ$ oracle. For every literal not in s built up using terms in s we make an entailment membership query and include it in c only if the answer to the query is "Yes"[4]. This is done by the procedure *rhs*. There are two versions for this procedure, one taking one parameter and another taking two. If there is only one input parameter, then the set of consequents is computed trying all possibilities. If a second input parameter is specified, only those literals appearing in this second set are checked and included in the result if necessary. This second version prevents from making unnecessary calls to the membership oracle in case we know beforehand that some literals will not be implied. To avoid unnecessary calls to the oracle, literals in c with terms not appearing in s will be automatically ruled out. To summarise:

- $rhs(s) = \{b \mid b \notin s \text{ and } EndMQ(s \rightarrow b) = Yes\}$
- $rhs(s, c) = \{b \mid b \in c \text{ and } EndMQ(s \rightarrow b) = Yes\}$

[4]It is sufficient to consider only literals built up from terms appearing in s, since the target expression is range restricted.

3.1 Minimising the counterexample

```
1. Let $x$ be the counterexample obtained by the EntEQ oracle.
2. Let $s_x$ be the set of literals $\{b \mid H \models antecedent(x) \rightarrow b\}$.
3. Set $c_x$ to $rhs(s_x)$.
4. Repeat until no more changes are made
    • For every functional term $t$ appearing in $s_x$, in decreasing order of weight, 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 $[s'_x, rhs(s'_x, c'_x)]$.
5. Repeat until no more changes are made
    • For every term $t$ appearing in $s_x$, in increasing order of weight, 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 $[s'_x, rhs(s'_x, c'_x)]$.
6. Return $[s_x, c_x]$.
```

Figure 5: The minimisation procedure

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. The way in which this procedure tries to minimise the counterexample is by removing literals and generalising terms.

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 off 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 4). We do this by considering every term that is not a variable (i.e. 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. And we continue trying to generalise some other functional terms not yet considered. The process finishes when there are no terms that can 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. Note, too, that terms containing some new created variable need not be checked, because the order in which terms are checked is from more complex to more simple ones, and if we have some term containing a new created variable, then this term will have been checked already, when the internal term had not yet been generalised.

Finally, we enter the loop in which literals are removed (line 5). 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 it is still positive. If so, then the counterexample is updated to the new multi-clause obtained. And we continue trying to remove more literals that have not been considered so far. The process finishes when there are no terms that can be dropped in $[s_x, c_x]$. Note also that there is a better way to compute step 5 by keeping track of the failures of the check, so that those failures are never tried twice.

Example 6 This example illustrates the behaviour of the minimisation procedure. f, g stand for functional symbols or arity 1 and x, y, z for variables. Parentheses in terms are omitted since we deal with functions of arity 1 only. a, b, c are constants and p, q, r, s are the predicate symbols, all of arity 1 except for p which has arity 2.

- Target expression $T = \{[p(x, fy) \to r(y)], [q(fz) \to s(z)]\}$.
- Hypothesis $H = \{q(ffx) \to s(fx)\}$.
- Counterexample x as given by the *EntEQ* oracle: $p(ga, ffb), q(ffb), r(gfc) \to r(fb)$.
- After step 2, $s_x = \{p(ga, ffb), q(ffb), r(gfc), s(fb)\}$.
- After step 3, $c_x = \{r(fb)\}$.
- The first version of full counterexample is $[\{p(ga, ffb), q(ffb), r(gfc), s(fb)\}, \{r(fb)\}]$.
- Generalising terms. The list of functional terms is $[gfc, ffb, fc, fb, ga, c, b, a]$.
  - Generalise term $gfc \mapsto x_1$:
    - $[s'_x, c'_x] = [\{p(ga, ffb), q(ffb), r(x_1), s(fb)\}, \{r(fb)\}]$.
    - $rhs(s'_x c'_x) = \{r(fb)\}$.
    - $[s_x, c_x] = [\{p(ga, ffb), q(ffb), r(x_1), s(fb)\}, \{r(fb)\}]$.
    - The list of terms still to check is $[ffb, fb, ga, b, a]$.
  - Generalise term $ffb \mapsto x_2$:
    - $[s'_x, c'_x] = [\{p(ga, x_2), q(x_2), r(x_1), s(fb)\}, \{r(fb)\}]$.
    - $rhs(s'_x c'_x) = \emptyset$.
    - $[s_x, c_x] = [\{p(ga, ffb), q(ffb), r(x_1), s(fb)\}, \{r(fb)\}]$.
    - The list of terms still to check is $[fb, ga, b, a]$.
  - Generalise term $fb \mapsto x_3$:
    - $[s'_x, c'_x] = [\{p(ga, fx_3), q(fx_3), r(x_1), s(x_3)\}, \{r(x_3)\}]$.
    - $rhs(s'_x c'_x) = \{r(x_3)\}$.
    - $[s_x, c_x] = [\{p(ga, fx_3), q(fx_3), r(x_1), s(x_3)\}, \{r(x_3)\}]$.
    - The list of terms still to check is $[ga, a]$.
  - Generalise term $ga \mapsto x_4$:
    - $[s'_x, c'_x] = [\{p(x_4, fx_3), q(fx_3), r(x_1), s(x_3)\}, \{r(x_3)\}]$.
    - $rhs(s'_x c'_x) = \{r(x_3)\}$.
    - $[s_x, c_x] = [\{p(x_4, fx_3), q(fx_3), r(x_1), s(x_3)\}, \{r(x_3)\}]$.
    - No more terms to generalise and this loop finishes.
- Removing literals. The list of terms is $[x_1, x_3, x_4, fx_3]$.
  - Drop term x_1:
    - $[s'_x, c'_x] = [\{p(x_4, fx_3), q(fx_3), s(x_3)\}, \{r(x_3)\}]$.
    - $rhs(s'_x c'_x) = \{r(x_3)\}$.
    - $[s_x, c_x] = [\{p(x_4, fx_3), q(fx_3), s(x_3)\}, \{r(x_3)\}]$.
    - The list of terms still to check is $[x_3, x_4, fx_3]$.
  - Drop term x_3:
    - $[s'_x, c'_x] = [\{\}, \{\}]$.
    - $rhs(s'_x c'_x) = \emptyset$.
    - $[s_x, c_x] = [\{p(x_4, fx_3), q(fx_3), s(x_3)\}, \{r(x_3)\}]$.
    - The list of terms still to check is $[x_4, fx_3]$.

- Drop term x_4:
    - $[s'_x, c'_x] = [\{q(fx_3), s(x_3)\}, \{r(x_3)\}]$.
    - $rhs(s'_x c'_x) = \emptyset$.
    - $[s_x, c_x] = [s'_x, c'_x] = [\{p(x_4, fx_3), q(fx_3), s(x_3)\}, \{r(x_3)\}]$.
    - The list of terms still to check is $[fx_3]$.
- Drop term fx_3:
    - $[s'_x, c'_x] = [\{s(x_3)\}, \{r(x_3)\}]$.
    - $rhs(s'_x c'_x) = \emptyset$.
    - $[s_x, c_x] = [\{p(x_4, fx_3), q(fx_3), s(x_3)\}, \{r(x_3)\}]$.
    - No more terms to drop and the minimisation is finished.

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* (see Section 2.5 in Page 4).

3.2.1 Matchings

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, and hence, the hypothesis. 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 distinct 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 7 Let $[s_x, c_x] = [\{p(a, b)\}, \{q(a)\}]$ and $[s_i, c_i] = [\{p(f(1), 2)\}, \{q(f(1))\}]$. The terms appearing in s_x are $\{a, b\}$. And in s_i are $\{1, 2, f(1)\}$. The possible matchings are:

$$\begin{array}{lll} \sigma_1 = \{a - 1, b - 2\} & \sigma_3 = \{a - 2, b - 1\} & \sigma_5 = \{a - f(1), b - 1\} \\ \sigma_2 = \{a - 1, b - f(1)\} & \sigma_4 = \{a - 2, b - f(1)\} & \sigma_6 = \{a - f(1), b - 2\} \end{array}$$

Definition 14 (Extended matching) An *extended matching* is an ordinary matching with an extra column added to every entry of the matching (every entry consists of two terms in an ordinary 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.

Definition 15 (Legal matching) Let σ be an extended matching. We say 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.

Example 8 The matching σ is $\{a - c, f(a) - b, f(f(a)) - fb, g(f(f(a))) - g(f(f(c)))\}$.
The extended matching of σ is
```
[a - c => X]
[f(a) - b => Y]
[f(f(a)) - f(b) => f(Y)]
[g(f(f(a))) - g(f(f(c))) => g(f(f(X)))].
```

The terms appearing in the extension column of σ are $\{X, Y, f(Y), g(f(f(X)))\}$. The subterm $f(X)$ is not included in this set, and it is a subterm of the term $g(f(f(X)))$ appearing in the set. Therefore, this matching is not legal.

Example 9 The matching σ is $\{a - c, f(a) - b, f(f(a)) - fb\}$.
The extended matching of σ is `[a - c => X]`

```
[f(a) - b => Y]
[f(f(a)) - f(b) => f(Y)]
```

The terms appearing in the extension column of σ are $\{X, Y, f(Y)\}$. All subterms of the terms appearing in this set are also contained in it, and therefore σ is legal.

Our algorithm considers a more restricted type of matching, thus restricting the number of possible matchings for any pair of multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$.

Definition 16 (Basic 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 or equal than 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 $[s_i, c_i]$.

We construct basic matchings given $[s_x, c_x]$ and $[s_i, c_i]$ in the following way. Consider all possible matchings between the *variables* in s_x and the *terms* in s_i. 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.

Example 10 No parentheses for functions are written.

- $s_x = \{p(a, fx)\}$ with terms $\{a, x, fx\}$.
- $s_i = \{p(a, f1), p(a, 2)\}$ with terms $\{a, 1, 2, f1\}$.
- The basic matchings to consider are:
  - `[x - a]`: cannot add `[a - a]`, therefore discarded.
  - `[x - 1]`: completed with `[a - a]` and `[fx - f1]`.
  - `[x - 2]`: cannot add `[fx - f2]`, therefore discarded.
  - `[x - f1]` cannot add `[fx - ff1]`, therefore discarded.

Let $[s_x, c_x]$ and $[s_i, c_i]$ be any pair of multi-clauses, $[s_x, c_x]$ containing k variables and $[s_i, c_i]$ containing t distinct terms. There are a maximum of t^k distinct basic matchings between them, since we only combine *variables* of s_x with *terms* in s_i.

3.2.2 Pairings

We start our explanation of the pairing procedure. This procedure is described in Figure 6. The input to a pairing is a pair of multi-clauses and a basic matching between the terms appearing in them. A *legal* pairing is a pairing for which the input matching is legal. A *basic* pairing is a pairing for which the input 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. This restriction is quite strong in the sense that, for example, if the literals $p(f(f(1)))$ and $p(f(a))$ are included in s_x and s_i (respectively), then their *lgg* $p(f(X))$ will not be included even if the extended entry `[f(1) - a => X]` is in the matching. We will only include it if the

```
1. Let $[s_x, c_x]$ and $[s_i, c_i]$ be the multi-clauses to be paired.
2. Let $TBL$ be the table produced when computing the lgg of $s_x$ and $s_i$.
3. Let $\sigma$ be the basic matching between terms of $[s_x, c_x]$ and $[s_i, c_i]$.
4. Compute the extension of $\sigma$, restricted to the table $TBL$.
   If it is not possible to compute the lgg for some pair of terms in $\sigma$, then return $[\emptyset, \emptyset]$.
5. Else, let $[s, c]$ be $[lgg_{|\sigma}(s_x, s_i), lgg_{|\sigma}(s_x, c_i) \cup lgg_{|\sigma}(c_x, s_i) \cup lgg_{|\sigma}(c_x, c_i)]$.
6. Return $[s, c]$.
```

Figure 6: The pairing procedure

extended entry `[f(f(1)) - f(a) => f(X)]` appears in the matching. Similarly, suppose $p(a)$ appears in both s_x and s_i. Their *lgg* $p(a)$ will not be included unless the entry `[a - a => a]` appears in the matching.

To compute the consequent c of the pairing, the union of $lgg_{|\sigma}(s_x, c_i)$, $lgg_{|\sigma}(c_x, s_i)$ and $lgg_{|\sigma}(c_x, c_i)$ is computed. Note that in the consequent all those possible couples among $\{s_x, c_x, s_i, c_i\}$ are included except s_x, s_i, that is in the antecedent and therefore does not have to be in the consequent. The same *lgg* table is used as the one used for $lgg(s_x, s_i)$. To summarise:

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

Note that when computing any of the *lgg*s, 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 *lgg*s that are computed in a pairing: $lgg_{|\sigma}(s_x, s_i), lgg_{|\sigma}(s_x, c_i), lgg_{|\sigma}(c_x, s_i)$ and $lgg_{|\sigma}(c_x, c_i)$.

Example 11 How to compute the antecedent of a pairing:

- $s_x = \{p(a, fx)\}$ with terms $\{a, x, fx\}$.
- $s_i = \{p(a, f1), p(a, 2)\}$ with terms $\{a, 1, 2, f1\}$.
- The *lgg* of s_x and s_i is $\{p(a, fX), p(a, Y)\}$.
  The *lgg* table is
  ```
  [a - a => a]
  [x - 1 => X]
  [fx - f1 => fX]
  [fx - 2 => Y]
  ```
- From Example 10 we have only one possible basic matching, $\sigma = \{x - 1, a - a, fx - f1\}$.
  The extended matching is
  ```
  [x - 1 => X]
  [a - a => a]
  [fx - f1 => fX]
  ```
- The antecedent $s = lgg_{|\sigma}(s_x, s_i) = \{p(a, fX)\}$

3.2.3 What matchings do we consider?

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 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. The other restriction on the candidate matching consists in the fact that every one of its entries must appear in the original *lgg* table. This is mentioned in line 4 of the pairing procedure.

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 including constants, variables and functional terms; e_t for the maximum number of distinct terms in a counterexample as produced by the equivalence query oracle; 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 2.6, which is bounded by mt^t. Before starting with the proof, we give some definitions.

Definition 17 (Covering multi-clause) 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 $[s, c]$.

Definition 18 (Violating multi-clause) A multi-clause $[s, c]$ *violates* 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 violated by* $[s, c]$.

4.1 Brief description

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.

To bound the length of the sequence S the following condition is proved. Every element in S violates some clause of $U(T)$ but no two distinct elements of S violate the same clause of $U(T)$ (Lemma 23). The bound on the length of S is therefore m', the number of clauses of the transformation $U(T)$.

To see that every element in S violates some clause in $U(T)$, it is shown that all counterexamples in S are full multi-clauses w.r.t. the target expression T (Lemma 18) and that any full multi-clause must violate some clause in $U(T)$ (Corollary 7).

The fact that there are no two elements in S violating a same clause in T is proved by induction on the way S is constructed. Lemma 20 is used in this proof and it constitutes the most important lemma in our analysis. Lemma 20 states that if a minimised multi-clause $[s_x, c_x]$ violates some clause C in $U(T)$ covered by some other full multi-clause $[s_i, c_i]$, then there is a pairing, say $[s, c]$, considered by the algorithm that is going to replace $[s_i, c_i]$ in S. To show this, a matching σ is constructed and proved to be legal, basic and not discarded by the pairing procedure. This establishes that the pairing induced by σ is going to be considered by the learning algorithm. And it is shown that the conditions needed for replacing $[s_i, c_i]$, namely $rhs(s, c) \neq \emptyset$ and $size(s) \lneq size(s_i)$ are satisfied, and hence $[s_i, c_i]$ is replaced. This, together with Lemma 22 stating that if a legal pairing violates some clause C, then the clauses that originate the pairing cover C and at least one of them violates it, prove that there cannot be two different elements in S violating the same clause in $U(T)$.

Once the bound on S is established, we derive our final theorem by carefully counting the number of queries made to the oracles in every procedure. We proceed now with the analysis in detail.

4.2 Proof of correctness

Lemma 5 *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 their 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. And 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, $U_\pi(C) = ineq(s_c \cdot \hat{\theta}), s_c \cdot \hat{\theta} \rightarrow b_t \cdot \hat{\theta}$. Let θ'' be the substitution such that $\theta' = \hat{\theta} \cdot \theta''$. This θ'' exists since $\hat{\theta}$ is a mgu and θ' is also a unifier for every class in the partition by construction. The clause $U_\pi(C) = ineq(s_t), s_t \rightarrow b_t$ is falsified using the substitution θ'':

- $s_c \cdot \underbrace{\theta'}_{\hat{\theta} \cdot \theta''} \subseteq ext(I)$ implies $\underbrace{s_c \cdot \hat{\theta}}_{s_t} \cdot \theta'' \subseteq ext(I)$ implies $s_t \cdot \theta'' \subseteq ext(I)$, and
- $b_c \cdot \underbrace{\theta'}_{\hat{\theta} \cdot \theta''} \notin ext(I)$ implies $\underbrace{b_c \cdot \hat{\theta}}_{b_t} \cdot \theta'' \notin ext(I)$ implies $b_t \cdot \theta'' \notin ext(I)$.

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. And it can 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 $s_t \cdot \underbrace{\theta'' \cdot \sigma^{-1}}_{\theta} \subseteq \underbrace{ext(I) \cdot \sigma^{-1}}_{s}$.
- $ineq(s_t) \cdot \theta \subseteq ineq(s)$. 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, that 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 $b_t \cdot \underbrace{\theta'' \cdot \sigma^{-1}}_{\theta} \notin \underbrace{ext(I) \cdot \sigma^{-1}}_{s}$.

■

Example 12 This example illustrates Lemma 5. No parentheses are written, as function f is unary.

- $T = \{C\} = \{p(a, fx, y) \to q(x, y)\}$ with terms $\{a, x, y, fx\}$.
- $U(T) = \{(a \neq x \neq y \neq fx), \mathbf{p}(\mathbf{a}, \mathbf{fx}, \mathbf{y}) \to \mathbf{q}(\mathbf{x}, \mathbf{y})$, (from partition $\{\{a\}, \{x\}, \{y\}, \{fx\}\}$)
  $(a \neq y \neq fa), \mathbf{p}(\mathbf{a}, \mathbf{fa}, \mathbf{y}) \to \mathbf{q}(\mathbf{a}, \mathbf{y})$, (from partition $\{\{a, x\}, \{y\}, \{fx\}\}$)
  $(a \neq x \neq fx), \mathbf{p}(\mathbf{a}, \mathbf{fx}, \mathbf{a}) \to \mathbf{q}(\mathbf{x}, \mathbf{a})$, (from partition $\{\{x\}, \{a, y\}, \{fx\}\}$)
  $(a \neq x \neq fx), \mathbf{p}(\mathbf{a}, \mathbf{fx}, \mathbf{x}) \to \mathbf{q}(\mathbf{x}, \mathbf{x})$, (from partition $\{\{a\}, \{x, y\}, \{fx\}\}$)
  $(a \neq fa), \mathbf{p}(\mathbf{a}, \mathbf{fa}, \mathbf{a}) \to \mathbf{q}(\mathbf{a}, \mathbf{a})$, (from partition $\{\{a, x, y\}, \{fx\}\}$)
  $(a \neq x \neq fx), \mathbf{p}(\mathbf{a}, \mathbf{fx}, \mathbf{fx}) \to \mathbf{q}(\mathbf{x}, \mathbf{fx})$, (from partition $\{\{a\}, \{x\}, \{y, fx\}\}$)
  $(a \neq fa), \mathbf{p}(\mathbf{a}, \mathbf{fa}, \mathbf{fa}) \to \mathbf{q}(\mathbf{a}, \mathbf{fa})\}$ (from partition $\{\{a, x\}, \{y, fx\}\}$)
- $[s, c] = [\{p(a, fz, fz)\}, \{q(z, fz)\}]$ with terms $\{a, z, fz\}$.

Clearly, $T \models [s, c]$ since the only clause in T subsumes $[s, c]$ with the substitution $\{x \mapsto z, y \mapsto fz\}$. We construct the interpretation I:

- Domain $D_I = \{1, 2, 3, *\}$. And $\sigma = \{a \mapsto 1, z \mapsto 2, fz \mapsto 3\}$
- Function mapping for constant a: $a = 1$
  Function mapping for function f: $f1 = *, f2 = 3, f3 = *, f* = *$
- Extension $ext(I) = s \cdot \sigma = \{p(a, fz, fz)\} \cdot \{a \mapsto 1, z \mapsto 2, fz \mapsto 3\} = \{p(1, 3, 3)\}$

It holds that $I \not\models [s, c]$ with the variable assignment $\{z \mapsto 2\}$. The multi-clause $[s, c]$ interpreted in I following the variable assignment mentioned earlier is $[\{p(1, 3, 3)\}, \{q(2, 3)\}]$ and $\{p(1, 3, 3)\} \subseteq ext(I)$ but $\{q(2, 3)\} \cap ext(I) = \emptyset$.

And $I \not\models T$. T has only one clause C in our example, therefore, $I \not\models p(a, fx, y) \to q(x, y)$. To see this, consider the variable substitution $\theta' = \{x \mapsto 2, y \mapsto 3\}$. This clause interpreted in I following θ' is $p(1, 3, 3) \to q(2, 3)$ and $p(1, 3, 3) \in ext(I)$ but $q(2, 3) \notin ext(I)$. This shows falsity of this clause in I.

To choose the right clause in $U(C)$, we complete θ' with all terms in C and obtain $\{x \mapsto 2, y \mapsto 3, a \mapsto 1, fx \mapsto 3\}$. This induces the partition $\{\{a\}, \{x\}, \{y, fx\}\}$.

The clause we choose from $U(C)$ is $\underbrace{(a \neq x \neq fx)}_{ineq(s_t)}, \underbrace{p(a, fx, fx)}_{s_t} \to \underbrace{q(x, fx)}_{b_t}$.

The mgu used to obtain this clause from C is $\hat{\theta} = \{y \mapsto fx\}$. The substitution θ'' corresponding to θ' after $\hat{\theta}$ has been applied is $\theta'' = \{x \mapsto 2\}$.

The substitution θ is $\theta'' \cdot \sigma^{-1} = \{x \mapsto 2\} \cdot \{1 \mapsto a, 2 \mapsto z, 3 \mapsto fz\} = \{x \mapsto z\}$. And θ satisfies:

- $s_t \cdot \theta \subseteq s$: $s_t \cdot \theta = \{p(a, fx, fx)\} \cdot \{x \mapsto z\} = \{p(a, fz, fz)\} \subseteq s = \{p(a, fz, fz)\}$.
- $ineq(s_t) \cdot \theta \subseteq ineq(s)$: $ineq(s_t) \cdot \theta = (a \neq z \neq fz) \subseteq ineq(s) = (a \neq z \neq fz)$.

- $b_t \cdot \theta \notin s$: $b_t \cdot \theta = q(x, fx) \cdot \{x \mapsto z\} = q(z, fz) \notin s = \{p(a, fz, fz)\}$.

Lemma 6 *If a multi-clause $[s, c]$ is positive for some target expression T, $c \neq \emptyset$ and it is exhaustive w.r.t. T, then some clause of $U(T)$ must be violated by $[s, c]$.*

Proof. By Lemma 5, there is a mapping θ such that $[s, c]$ covers some clause $ineq(s_t), s_t \to b_t$ of $U(T)$ and $b_t \cdot \theta \notin s$. Since $ineq(s_t), s_t \to b_t$ is a clause in $U(T)$, we can conclude that $T \models s_t \to b_t$. This happens because by the way the transformation was constructed, there must be some clause in T subsuming $s_t \to b_t$. Therefore, $T \models s_t \cdot \theta \to b_t \cdot \theta$ and since $s_t \cdot \theta \subseteq s$, $T \models s \to b_t \cdot \theta$.

Since $[s, c]$ is exhaustive, $b_t \cdot \theta \notin s$ and $T \models s \to b_t \cdot \theta$, the literal $b_t \cdot \theta$ must be included in c. The multi-clause $[s, c]$ covers $ineq(s_t), s_t \to b_t$ via θ and $b_t \cdot \theta \in c$. Therefore, $[s, c]$ violates $ineq(s_t), s_t \to b_t$ via θ. ■

Corollary 7 *If a multi-clause $[s, c]$ is full w.r.t. some target expression T and $c \neq \emptyset$, then some clause of $U(T)$ must be violated by $[s, c]$.*

Proof. The conditions of Lemma 6 are satisfied. ■

Lemma 8 *If a full $[s, c]$ violates a clause $ineq(s_t), s_t \to b_t$ in $U(T)$, then $rhs(s, c) \neq \emptyset$.*

Proof. Since $[s, c]$ violates the clause $ineq(s_t), s_t \to b_t$ in $U(T)$, there is a substitution θ such that $s_t \cdot \theta \subseteq s$ and $b_t \cdot \theta \in c$. Let $s_c \to b_c$ be the clause in T that generated $ineq(s_t), s_t \to b_t$ in $U(T)$. This is, there is a unifying substitution σ such that $s_c \cdot \sigma = s_t$ and $b_c \cdot \sigma = b_t$. Let $\theta' = \sigma \cdot \theta$. Thus, $s_c \cdot \theta' \subseteq s$ and $b_c \cdot \theta' \in c$.

$T \models s_c \to b_c$ implies $T \models s_c \cdot \theta' \to b_c \cdot \theta'$. And since $s_c \cdot \theta' \subseteq s$, $T \models s \to b_c \cdot \theta'$. Also, $b_c \cdot \theta'$ is in c so that $b_c \cdot \theta' \in rhs(s, c) \neq \emptyset$. ■

Lemma 9 *Every multi-clause $[s_x, c_x]$ produced by the minimisation procedure is full w.r.t. the target expression T.*

Proof. To see that the multi-clause is correct it suffices to observe that every time the candidate multi-clause has been updated, the consequent part is computed as the output of the procedure *rhs*. Therefore, it must be correct.

The first version of the counterexample $[s_x, c_x]$ as produced by step 3 of the algorithm is exhaustive since c_x is computed by use of $rhs(s_x)$.

We prove that after generalising a term the resulting counterexample is also exhaustive. Let $[s_x, c_x]$ be the multi-clause before generalising t and $[s'_x, c'_x]$ after. Let the substitution θ be $\{t \mapsto x_t\}$. Then, $s_x \cdot \theta = s'_x$, $c_x \cdot \theta = c'_x$ and also $s_x = s'_x \cdot \theta^{-1}$, because x_t is a new variable that does not appear in s_x. We will see that any literal $b \notin s'_x$ implied by s'_x w.r.t. T is included in c'_x, and hence $[s'_x, c'_x]$ is exhaustive, so that $[s'_x, rhs(c'_x)]$ is exhaustive as well. Suppose, then, that $T \models s'_x \to b$ and $b \notin s'_x$. This implies that $T \models s'_x \cdot \theta^{-1} \to b \cdot \theta^{-1}$. Thus, $T \models s_x \to b \cdot \theta^{-1}$. Also, $b \cdot \theta^{-1} \notin s'_x \cdot \theta^{-1}$ implies $b \cdot \theta^{-1} \notin s_x$. By induction hypothesis, $[s_x, c_x]$ is exhaustive, therefore $b \cdot \theta^{-1} \in c_x$. And hence, $b \cdot \theta^{-1} \cdot \theta = b \in c'_x$. And any counterexample $[s_x, c_x]$ after step 4 is exhaustive.

We will show now that after dropping some term t the multi-clause still remains exhaustive. Again, let $[s_x, c_x]$ be the multi-clause before removing t and $[s'_x, c'_x]$ after removing it. It is clear that $s'_x \subseteq s_x$ and $c'_x \subseteq c_x$ since both have been obtained by removing literals only. Therefore, the only literals that could be missing in c_x are the ones in $s_x \setminus s'_x$. But we know that such literals cannot be implied by s'_x because they contain terms that do not appear in s'_x (remember that the target expression is range restricted). Therefore, after step 5 and as returned by the minimisation procedure, the counterexample $[s_x, c_x]$ is exhaustive. ■

Definition 19 (Positive counterexample) 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 $b \in c$, $H \not\models s \rightarrow b$.

Lemma 10 *All counterexamples given by the equivalence query oracle are positive w.r.t. the target T and the hypothesis H.*

Proof. Follows from the fact that only correct clauses are included in H, and hence $T \models H$. ■

Lemma 11 *Every multi-clause $[s_x, c_x]$ produced by the minimisation procedure is a positive counterexample for the target expression T and for the hypothesis H.*

Proof. To prove that $[s_x, c_x]$ is a positive counterexample we need to prove that $T \models [s_x, c_x]$, $c_x \neq \emptyset$ and for every $b \in c_x$ it holds that $H \not\models s_x \rightarrow b_x$. By Lemma 9, we know that $[s_x, c_x]$ is full, and hence correct. This implies that $T \models [s_x, c_x]$. It remains to show that H does not imply any of the clauses in $[s_x, c_x]$ and that $c_x \neq \emptyset$.

Let x be the original counterexample obtained from the equivalence oracle. This x is such that $T \models x$ but $H \not\models x$ (see Lemma 10). The antecedent of the multi-clause s_x is set to be $\{b \mid H \models \mathit{antecedent}(x) \rightarrow b\}$. Hence, $\mathit{antecedent}(x) \subseteq s_x$. We know that $\mathit{consequent}(x)$ is not included in s_x because x is a counterexample and hence $H \not\models x$. The consequent c_x is computed as $\mathit{rhs}(s_x)$. We can conclude, then, that $\mathit{consequent}(x) \in c_x$ because it is implied by and not included in s_x. Therefore, $c_x \neq \emptyset$. Also, $H \not\models [s_x, \mathit{rhs}(s_x)]$, since all literals implied by $\mathit{antecedent}(x)$ appear in s_x, and therefore in $\mathit{rhs}(s_x)$ only literals not implied by H appear. Therefore, after step 3 of the minimisation procedure, the multi-clause $[s_x, c_x]$ is a positive counterexample.

Next, we will see that after generalising some functional term t, the multi-clause still remains a positive counterexample. The multi-clause $[s_x, c_x]$ is only updated if the consequent part is nonempty, therefore, all the multi-clauses obtained by generalising will have a nonempty consequent. Let $[s_x, c_x]$ be the multi-clause before generalising t, and $[s'_x, c'_x]$ after. Assume $[s_x, c_x]$ is a positive counterexample. Let θ be the substitution $\{t \mapsto x_t\}$. As in Lemma 9, $s_x \cdot \theta = s'_x$, $c_x \cdot \theta = c'_x$, $s_x = s'_x \cdot \theta^{-1}$ and also $c_x = c'_x \cdot \theta^{-1}$. Suppose by way of contradiction that $H \models s'_x \rightarrow b'$, for some $b' \in c'_x$. Then, $H \models s'_x \cdot \theta^{-1} \rightarrow b' \cdot \theta^{-1}$. And we get that $H \models s_x \rightarrow b' \cdot \theta^{-1}$. Note that $b' \in c'_x$ implies that $b' \cdot \theta^{-1} \in c'_x \cdot \theta^{-1}$ and hence $b' \cdot \theta^{-1} \in c_x$. This contradicts our assumption stating that $[s_x, c_x]$ was a counterexample, since we have found a literal in c_x implied by s_x w.r.t. H. Thus, the multi-clause $[s_x, c_x]$ after step 4 is a positive counterexample for the target T and the hypothesis H.

Finally, we will show that after dropping some term t the multi-clause still remains a positive counterexample. As before, the multi-clause $[s_x, c_x]$ is only updated if the consequent part is nonempty, therefore, all the multi-clauses obtained by dropping will have a nonempty consequent. Let $[s_x, c_x]$ be the multi-clause before removing some of its literals, and $[s'_x, c'_x]$ after. It is the case that $s'_x \subseteq s_x$ and $c'_x \subseteq c_x$. To see this second inclusion, note that first, some literals are removed from c_x and the resulting set goes through rhs, that might remove some literals as well. Suppose $[s_x, c_x]$ is a positive counterexample. Hence, $H \not\models s_x \rightarrow b$, for every $b \in c_x$. Therefore, $H \not\models s'_x \rightarrow b$ because $s'_x \subseteq s_x$. And this is for every $b \in c'_x$, because $c'_x \subseteq c_x$ and $b \in c_x$. ■

Lemma 12 *Let $[s_x, c_x]$ be a multi-clause as generated by the minimisation procedure. If $[s_x, c_x]$ violates some clause $\mathit{ineq}(s_t), s_t \rightarrow b_t$ 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 violating $\mathit{ineq}(s_t), s_t \rightarrow b_t$, hence there must exist a substitution θ from variables in s_t into terms in s_x such that $s_t \cdot \theta \subseteq s_x$, $\mathit{ineq}(s_t) \cdot \theta \subseteq \mathit{ineq}(s_x)$ 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 4 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 $[s'_x, c'_x]$ violates $ineq(s_t), s_t \rightarrow b_t$ 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 the violation we need to show:

- $s_t \cdot \theta \cdot \theta_t \subseteq s'_x$. Easy, since 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 violated 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 $t'_2 \cdot \theta_t$ also are.

  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.
- $b_t \cdot \theta \cdot \theta_t \in c'_x$. Easy, since by hypothesis $b_t \cdot \theta \in c_x$ implies $b_t \cdot \theta \cdot \theta_t \in c_x \cdot \theta_t = c'_x$.

Hence, no variable in θ can be mapped into a functional term and θ is a variable renaming. ■

Lemma 13 *Let $[s_x, c_x]$ be a multi-clause as output by the minimisation procedure. And let $ineq(s_t), s_t \rightarrow b_t$ be any clause of $U(T)$ violated by $[s_x, c_x]$. Then, the number of distinct terms in $[s_x, b_x]$ is equal to the number of distinct terms in $ineq(s_t), s_t \rightarrow b_t$.*

Proof. Let n_x and n_t be the number of distinct terms appearing in s_x and s_t, respectively. Subterms should also be counted. The multi-clause $[s_x, c_x]$ violates $ineq(s_t), s_t \rightarrow b_t$. Therefore there is a θ mapping variables in s_t showing this violation. The substitution θ satisfies $ineq(s_t) \cdot \theta \subseteq ineq(s_x)$ as this is one of the violation conditions, and therefore a different variable are mapped into different terms of s_x by θ. By Lemma 12, we know also that every variable of s_t is mapped into a variable of s_x. Therefore, θ maps distinct variables of s_t into distinct variables of s_x. Therefore, the number of terms in s_t equals the number of terms in $s_t \cdot \theta$, since there has only been a non-unifying renaming of variables. Also, $s_t \cdot \theta \subseteq s_x$. We have to check that the remaining literals in $s_x \setminus s_t \cdot \theta$ do not include any term not appearing in $s_t \cdot \theta$.

Suppose there is a literal $l \in s_x \setminus (s_t \cdot \theta)$ containing some term, say t, not appearing in $s_t \cdot \theta$. Consider when in step 5 of the minimisation procedure the term t was checked. Let $[s'_x, c'_x]$ be the clause obtained after the removal of the literals containing t. Then, $s_t \cdot \theta \subseteq s'_x$ because all the literals in $s_t \cdot \theta$ do not contain t. Also, $b_t \cdot \theta \in c'_x$ because it does not either (T is range restricted). Therefore, the $[s'_x, c'_x]$ still violates $ineq(s_t), s_t \rightarrow b_t$. And therefore, $rhs(s'_x, c'_x) \neq \emptyset$ and such a term t cannot exist. Therefore, $n_t = n_x$ as required. ■

Corollary 14 *The number of terms of a counterexample as generated by the minimisation procedure is bounded by t, the maximum of the number of distinct terms in the target clauses.*

Proof. Follows easily from the fact that that any n_t as in the previous lemma is bounded by t, since the transformed clauses in $U(T)$ never contain more terms than the ones in T that originated them. ■

Lemma 15 *Let $[s, c]$ be any multi-clause covering some clause $ineq(s_t), s_t \to b_t$. Let n and n_t be the number of distinct terms in s and s_t, respectively. Then, $n_t \leq n$.*

Proof. Since $[s, c]$ covers the clause $ineq(s_t), s_t \to b_t$, there is a θ s.t. $ineq(s_t) \cdot \theta \subseteq ineq(s)$. Therefore, any two distinct terms t, t' of s_t appear as distinct terms $t \cdot \theta, t' \cdot \theta$ in s. And therefore, s has at least as many terms as s_t. ■

Corollary 16 *Let $ineq(s_t), s_t \to b_t$ be a clause of $U(T)$ with n_t distinct terms. Let $[s_x, c_x]$ be a multi-clause with n_x distinct terms as output by the minimisation procedure such that $[s_x, c_x]$ violates the clause $ineq(s_t), s_t \to b_t$. Let $[s_i, c_i]$ be a multi-clause with n_i terms covering the clause $ineq(s_t), s_t \to b_t$. Then $n_x \leq n_i$.*

Proof. By Lemma 13, $n_x = n_t$. By Lemma 15, $n_t \leq n_i$, hence $n_x \leq n_i$. ■

Lemma 17 *Let $[s_x, c_x]$ and $[s_i, c_i]$ be two full multi-clauses w.r.t. the target expression T. Let σ be a basic matching between the terms in s_x and s_i that is not rejected by the pairing procedure. Let $[s, c]$ be 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.*

Proof. To see that $[s, rhs(s, c)]$ is full w.r.t. T, it is sufficient to show that $[s, c]$ is exhaustive. That is, whenever $T \models s \to b$ and $b \notin s$, then $b \in c$. Suppose, then, that $T \models s \to b$ with $b \notin s$. Since $s = lgg_{|\sigma}(s_x, s_i) \subseteq lgg(s_x, s_i)$, we know that there exist θ_x and θ_i such that $s \cdot \theta_x \subseteq s_x$ and $s \cdot \theta_i \subseteq s_i$. $T \models s \to b$ implies both $T \models (s \to b) \cdot \theta_x$ and $T \models (s \to b) \cdot \theta_i$. Therefore, $T \models s \cdot \theta_x \to b \cdot \theta_x$ and $T \models s \cdot \theta_i \to b \cdot \theta_i$. Let $b_x = b \cdot \theta_x$ and $b_i = b \cdot \theta_i$. Finally, we obtain that $T \models s_x \to b_x$ and $T \models s_i \to b_i$. By assumption, $[s_x, c_x]$ and $[s_i, c_i]$ are full, and therefore $b_x \in s_x \cup c_x$ and $b_i \in s_i \cup c_i$. Also, since the same lgg table is used for all $lgg(\cdot, \cdot)$ we know that $b = lgg(b_x, b_i)$. Therefore b must appear in one of $lgg(s_x, s_i), lgg(s_x, c_i), lgg(c_x, s_i)$ or $lgg(c_x, c_i)$. But $b \notin lgg(s_x, s_i)$ since $b \notin s$ by assumption.

Note that all terms and subterms in b appear in s. If not, then it could not have been implied by s w.r.t. T, since T is range restricted. We know that σ is basic and hence legal, and therefore it contains all subterms of terms appearing in s. Thus, by restricting any of the $lgg(\cdot, \cdot)$ to $lgg_{|\sigma}(\cdot, \cdot)$, we will not get rid of b, since it is built up from terms that appear in s and hence in σ. Therefore, $b \in lgg_{|\sigma}(s_x, c_i) \cup lgg_{|\sigma}(c_x, s_i) \cup lgg_{|\sigma}(c_x, c_i) = c$ as required. ■

Lemma 18 *Every element $[s, c]$ appearing in the sequence S is full w.r.t. the target expression T.*

Proof. The sequence S is constructed by appending minimised counterexamples or by refining existing elements with a pairing with another minimised counterexample. Lemma 9 guarantees that all minimised counterexamples are full and, by Lemma 17, any basic pairing between full multi-clauses is also full. ■

Lemma 19 *Let $[s, c]$ be any pairing of the two multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$. Then, it is the case that $|s| \leq |s_i|$ and $|s| \leq |s_x|$.*

Proof. It is sufficient to observe that in s there is at most one copy of every atom s_x and s_i. This is true since the matching used to include literals in s is 1 to 1 and therefore a term can only be combined with a unique term and no duplication of literals occurs. ■

Lemma 20 *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 violates some clause $ineq(s_t), s_t \to b_t$ in $U(T)$ 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. 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:

1. $rhs(s, c) \neq \emptyset$
2. $size(s) \lneq size(s_i)$

We have assumed that there is some clause $ineq(s_t), s_t \rightarrow b_t \in U(T)$ violated by $[s_x, c_x]$ and covered by $[s_i, c_i]$. Let θ'_x be the substitution showing the violation of $ineq(s_t), s_t \rightarrow b_t$ by $[s_x, c_x]$ and θ'_i be the substitution showing the fact that $ineq(s_t), s_t \rightarrow b_t$ is 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).

Claim 1 *The matching σ as described above is 1-1 and the number of entries equals the minimum of the number of distinct terms in s_x and s_i.*

Proof. All the entries of σ have the form `[`$t \cdot \theta'_x$ `-` $t \cdot \theta'_i$ `=>` $lgg(t \cdot \theta'_x, t \cdot \theta'_i)$`]`. For σ to be 1-1 it is sufficient to see that there are no two terms t, t' of s_t generating the following entries in σ

`[`$t \cdot \theta'_x$ `-` $t \cdot \theta'_i$ `=>` $lgg(t \cdot \theta'_x, t \cdot \theta'_i)$`]` and `[`$t' \cdot \theta'_x$ `-` $t' \cdot \theta'_i$ `=>` $lgg(t' \cdot \theta'_x, t \cdot \theta'_i)$`]`

such that $t \cdot \theta'_x = t' \cdot \theta'_x$ or $t \cdot \theta'_i = t' \cdot \theta'_i$. But this is clear since $[s_x, c_x]$ and $[s_i, c_i]$ are covering $ineq(s_t), s_t \rightarrow b_t$ via θ'_x and θ'_i, respectively. Therefore $ineq(s_t) \cdot \theta'_x \subseteq ineq(s_x)$ and $ineq(s_t) \cdot \theta'_i \subseteq ineq(s_i)$. And therefore $t \cdot \theta'_x$ and $t' \cdot \theta'_x$ appear as different terms in s_x. Also, $t \cdot \theta'_i$ and $t' \cdot \theta'_i$ appear as different terms in s_i. And σ is 1-1.

By construction, the number of entries equals the number of distinct terms in s_t, that by Lemma 13 is the number of distinct terms in s_x. And by Lemma 15, $[s_i, c_i]$ contains at least as many terms as s_t. Therefore, the number of entries in σ coincides with the minimum of the number of distinct terms in s_x and s_i. □

Example 13 Consider the following example:

- $s_t = \{p(g(c), x, f(y), z)\}$, with 6 terms $\{c, g(c), x, y, f(y), z\}$.
- $s_x = \{p(g(c), x', f(y'), z), p(g(c), g(c), f(y'), c)\}$, with 6 terms $\{c, g(c), x', y', f(y'), z\}$.
- $s_i = \{p(g(c), f(1), f(f(2)), z)\}$, with 8 terms $\{c, g(c), 1, f(1), 2, f(2), f(f(2)), z\}$.
- $\theta'_x = \{x \mapsto x', y \mapsto y', z \mapsto z\}$ is a variable renaming.
- $\theta'_i = \{x \mapsto f(1), y \mapsto f(2), z \mapsto z\}$.
- The *lgg* table is

```
[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]
```

- $lgg(s_x, s_i) = \{p(g(c), X, f(Y), z), p(g(c), Z, f(Y), V)\}$.
- The matching σ is `[c - c => c]` (from constant c)
  `[g(c) - g(c) => g(c)]` (from ground term $g(c)$)
  `[x' - f(1) => X]` (from variable x)
  `[y' - f(2) => Y]` (from variable y)
  `[f(y') - f(f(2)) => f(Y)]` (from term $f(y)$)
  `[z - z => z]` (from variable z)
- $lgg_{|\sigma}(s_x, s_i) = \{p(g(c), X, f(Y), z)\}$.

We consider the pairing of $[s_x, c_x]$ and $[s_i, c_i]$ induced by σ. We have to show that this pairing is not discarded, it is basic, $rhs(s, c) \neq \emptyset$ and $size(s) \lneq size(s_i)$.

Claim 2 *The matching σ is not discarded.*

Proof. Notice that the discarded pairings are those that do not agree with the lgg of s_x and s_i, but this does not happen in this case, since σ has been constructed precisely using the lgg of some terms in s_x and s_i. □

Claim 3 *The matching σ is legal.*

Proof. A matching is legal if, by definition, the subterm of any term appearing as the lgg of the matching, also appears in some other entry of the matching. We will prove it by induction on the structure of the terms. We prove that 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.

Base case. When $t = a$, with a being some constant. The entry in σ for it is `[a - a => a]`, since $a \cdot \theta = a$, for any substitution θ if a is a constant and $lgg(a, a) = a$. The term a has no subterms, and therefore all its subterms trivially appear as entries in σ.

Base case. When $t = v$, where v is any variable in s_t. The entry for it in σ is `[`$v \cdot \theta'_x$ `-` $v \cdot \theta'_i$ `=>` $lgg(v \cdot \theta'_x, v \cdot \theta'_i)$`]`. s_x is minimised and by Lemma 12 $v \cdot \theta'_x$ must be a variable. Therefore, the lgg with anything else must also be a variable. Hence, all its subterms appear trivially since there are no subterms.

Step case. When $t = f(t_1, ..., t_l)$, where f is a function symbol of arity l and $t_1, ..., t_l$ its arguments. The entry for it in σ is

$$[f(t_1, ..., t_l) \cdot \theta'_x \texttt{ - } f(t_1, ..., t_l) \cdot \theta'_i \texttt{ => } \underbrace{lgg(f(t_1, ..., t_l) \cdot \theta'_x, f(t_1, ..., t_l) \cdot \theta'_x)}_{f(lgg(t_1 \cdot \theta'_x, t_1 \cdot \theta'_i), ..., lgg(t_l \cdot \theta'_x, t_l \cdot \theta'_i))}].$$

The entries `[`$t_j \cdot \theta'_x$ `-` $t_j \cdot \theta'_i$ `=>` $lgg(t_j \cdot \theta'_x, t_j \cdot \theta'_x)$`]`, with $1 \leq j \leq l$, are also included in σ, since all t_j are terms of s_t. By the induction hypothesis, all the subterms of every $lgg(t_j \cdot \theta'_x, t_j \cdot \theta'_x)$ are included in σ, and therefore, all the subterms of $lgg(f(t_1, ..., t_l) \cdot \theta'_x, f(t_1, ..., t_l) \cdot \theta'_x)$ are also included in σ and the step case holds. □

Claim 4 *The matching σ is basic.*

Proof. A basic matching is defined only for two multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$ such that the number of terms in s_x is less or equal than the number of terms in s_i. Corollary 16 shows that this is indeed the case. Following the definition, it should be also 1-1 and legal. Claim 1 shows it is 1-1 and by Claim 3 we know it is also legal. It is only left to see that if entry $f(t_1, ..., t_n) - g(r_1, ..., r_m)$ is in σ, then $f = g$, $n = m$ and $t_l - r_l \in \sigma$ for all $l = 1, ..., n$.

Suppose, then, that $f(t_1, ..., t_n) - g(r_1, ..., r_m)$ is in σ. By construction of σ all entries are of the form $t \cdot \theta'_x - t \cdot \theta'_i$. Thus, assume $t \cdot \theta'_x = f(t_1, ..., t_n)$ and $t \cdot \theta'_i = g(r_1, ..., r_m)$. We also know that θ'_x

is a variable renaming, therefore, the term $t \cdot \theta'_x$ is a variant of t. Therefore, the terms $f(t_1, ..., t_n)$ and t are variants. This is, t itself has the form $f(t'_1, ..., t'_n)$, where every t'_l is a variant of t_l and $t'_l \cdot \theta'_x = t_l$, where $l = 1, ..., n$. Therefore, $g(r_1, ..., r_m) = t \cdot \theta'_i = f(r_1 = t'_1 \cdot \theta'_i, ..., r_n = t'_n \cdot \theta'_i)$ and hence $f = g$ and $n = m$. We have seen that $t_l = t'_l \cdot \theta'_x$ and $r_l = t'_l \cdot \theta'_i$. By construction, σ includes the entries $t_l - r_l$, for any $l = 1, ..., n$ and our claim holds. $\square$

It remains to show that the properties (1) and (2) must be satisfied.

Let θ_x and θ_i be defined as follows. An entry in σ [$t \cdot \theta'_x$ = $t \cdot \theta'_i$ => $lgg(t \cdot \theta'_x, t \cdot \theta'_i)$] such that $lgg(t \cdot \theta'_x, t \cdot \theta'_i)$ is a variable will generate the mapping $lgg(t \cdot \theta'_x, t \cdot \theta'_i) \mapsto t \cdot \theta'_x$ in θ_x and $lgg(t \cdot \theta'_x, t \cdot \theta'_i) \mapsto t \cdot \theta'_i$ in θ_i. That is, $\theta_x = \{lgg(t \cdot \theta'_x, t \cdot \theta'_i) \mapsto t \cdot \theta'_x\}$ and $\theta_i = \{lgg(t \cdot \theta'_x, t \cdot \theta'_i) \mapsto t \cdot \theta'_i\}$, whenever $lgg(t \cdot \theta'_x, t \cdot \theta'_i)$ is a variable and t is a term in s_t.

In our example, $\theta_x = \{X \mapsto x', Y \mapsto y', z \mapsto z\}$ and $\theta_i = \{X \mapsto f(1), Y \mapsto f(2), z \mapsto z\}$.

The substitutions θ_x and θ_i are the ones that show subsumption of s_x and s_i by $lgg_{|\sigma}(s_x, s_i)$. Namely,

- $s \cdot \theta_x \subseteq s_x$. Let l be any literal in s, l has been obtained by taking the *lgg* of two literals l_x and l_i in s_x and s_i, respectively. That is, $l = lgg(l_x, l_i)$. Moreover, l only contains terms in the extension of σ, otherwise it would have been removed when restricting the *lgg*. The substitution θ_x is such that $l \cdot \theta_x = l_x$ because it "undoes" what the *lgg* does for the literals with terms in σ. And $l_x \in s_x$, therefore, the inclusion $s \cdot \theta_x \subseteq s_x$ holds.

- $s \cdot \theta_i \subseteq s_i$. Similar to previous.

Claim 5 *$rhs(s, c) \neq \emptyset$.*

Proof. To see this, it suffices by Lemma 8 to show that $[s, c]$ violates $ineq(s_t), s_t \rightarrow b_t$. We have to find a substitution θ such that $s_t \cdot \theta \subseteq s$, $ineq(s_t) \cdot \theta \subseteq ineq(s)$ and $b_t \cdot \theta \notin s$. Since $[s, c]$ is full, then $b_t \cdot \theta \in c$ and $rhs(s, c) \neq \emptyset$ as required.

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 following holds.

- $\theta \cdot \theta_x = \theta'_x$. Let v be any variable in s_t. The substitution θ maps v into $lgg(v \cdot \theta'_x, v \cdot \theta'_i)$. This is a variable, say V, since we know θ'_x is a variable renaming. The substitution θ_x contains the mapping $\underbrace{lgg(v \cdot \theta'_x, v \cdot \theta'_i)}_{V} \mapsto v \cdot \theta'_x$. And v is mapped into $v \cdot \theta'_x$ by $\theta \cdot \theta_x$.

  In our example: $\theta'_x = \{x \mapsto x', y \mapsto y', z \mapsto z\}$, and
  $\theta \cdot \theta_x = \{x \mapsto X, y \mapsto Y, z \mapsto z\} \cdot \{X \mapsto x', Y \mapsto y', z \mapsto z\}$.

- $\theta \cdot \theta_i = \theta'_i$. As in previous property.

- $s_t \cdot \theta \subseteq s = lgg_{|\sigma}(s_x, s_i)$. Let l be any literal in s_t and t be any term appearing in l. The matching σ contains the entry [$t \cdot \theta'_x$ - $t \cdot \theta'_i$ => $lgg(t \cdot \theta'_x, t \cdot \theta'_i)$], since t appears in s_t. The substitution θ contains $\{v \mapsto lgg(v \cdot \theta'_x, v \cdot \theta'_i)\}$ for every variable v appearing in s_t, therefore $t \cdot \theta = lgg(t \cdot \theta'_x, t \cdot \theta'_i)$. And $lgg(t \cdot \theta'_x, t \cdot \theta'_i)$ appears in σ. The literal $l \cdot \theta$ appears in $lgg(s_t \cdot \theta'_x, s_t \cdot \theta'_i)$ and therefore in $lgg(s_x, s_i)$ since $s_t \cdot \theta'_x \subseteq s_x$, $s_t \cdot \theta'_i \subseteq s_i$ and $\theta = \{v \mapsto lgg(v \cdot \theta'_x, v \cdot \theta'_i) \mid v \textit{ is a variable of } s_t\}$. Also, $l \cdot \theta$ appears in $lgg_{|\sigma}(s_x, s_i)$ since we have seen that any term in $l \cdot \theta$ appears in σ.

  In our example the only l we have in $s_t \cdot \theta$ is $p(g(c), x, f(y), z) \cdot \theta = p(g(c), X, f(Y), z)$. And $lgg_{|\sigma}(s_x, s_y)$ is precisely $p(g(c), X, f(Y), z)$.

- $\mathit{ineq}(s_t) \cdot \theta \subseteq \mathit{ineq}(s)$. We have to show that for any two distinct terms t, t' of s_t, the terms $t \cdot \theta$ and $t' \cdot \theta$ are also different terms in s, and therefore the inequality $t \cdot \theta \neq t' \cdot \theta$ appears in $\mathit{ineq}(s)$. By hypothesis, $\mathit{ineq}(s_t) \cdot \theta'_x \subseteq \mathit{ineq}(s_x)$. Since $\theta'_x = \theta \cdot \theta_x$, we get $\mathit{ineq}(s_t) \cdot \theta \cdot \theta_x \subseteq \mathit{ineq}(s_x)$ and so $t \cdot \theta \cdot \theta_x$ and $t' \cdot \theta \cdot \theta_x$ are different terms of s_x. If follows that $t \cdot \theta \neq t' \cdot \theta \in \mathit{ineq}(s)$, since otherwise we could have never distinguished $t \cdot \theta \cdot \theta_x$ from $t' \cdot \theta \cdot \theta_x$.

- $b_t \cdot \theta \notin s$. Suppose it is not the case and $b_t \cdot \theta \in s$. This implies that $b_t \cdot \theta \cdot \theta_x \in s \cdot \theta_x \subseteq s_x$. Since $\theta'_x = \theta \cdot \theta_x$, it follows that $b_t \cdot \theta'_x \in s_x$, contradicting the fact that $[s_x, c_x]$ violates $\mathit{ineq}(s_t), s_t \rightarrow b_t$ via θ'_x.

□

Claim 6 *$\mathit{size}(s) \lneq \mathit{size}(s_i)$.*

Proof. By way of contradiction, suppose $\mathit{size}(s) \geq \mathit{size}(s_i)$. By Lemma 19, we know that $|s| \leq |s_i|$, therefore $\mathit{size}(s) \leq \mathit{size}(s_i)$ since the *lgg* never substitutes a term by one of greater weight. Thus, it can only be that $\mathit{size}(s) = \mathit{size}(s_i)$ and $|s| = |s_i|$ (if $|s| < |s_i|$, then we would also get $\mathit{size}(s) < \mathit{size}(s_i)$ for the same reason as before). Remember that θ_i is the substitution for which $s \cdot \theta_i \subseteq s_i$. We conclude that θ_i is a variable renaming, since the sizes of s and s_i are equal and therefore $s \cdot \theta_i = s_i$. Also, $s_i \cdot \theta_i^{-1} = s$.

Remember θ'_x was a variable renaming and $\theta'_x = \theta \cdot \theta_x$. By the way they were constructed, θ and θ_x must be variable renamings, too. $s_i \cdot \theta_i^{-1} = s$ and $s \cdot \theta_x \subseteq s_x$ imply $s_i \cdot \theta_i^{-1} \cdot \theta_x \subseteq s_x$. We define $\hat{\theta}$ as $\theta_i^{-1} \cdot \theta_x$ and conclude $s_i \cdot \hat{\theta} \subseteq s_x$. Notice $\hat{\theta}$ is a variable renaming because θ_i^{-1} and θ_x are. That is, there is a variable renaming $\hat{\theta}$ such that $s_i \cdot \hat{\theta} \subseteq s_x$, $\theta_x = \theta_i \cdot \hat{\theta}$ and $\theta'_x = \theta'_i \cdot \hat{\theta}$.

Consider the literal $b_t \cdot \theta'_i$. We disprove the following two cases:

- $b_t \cdot \theta'_i \in s_i$. Since $\theta'_i = \theta \cdot \theta_i$ and $s \cdot \theta_i = s_i$, we obtain $b_t \cdot \theta \cdot \theta_i \in s \cdot \theta_i$. The substitution θ_i is just a variable renaming, hence $b_t \cdot \theta \in s$. But $b_t \cdot \theta \notin s$, since by the proof of claim 5 we know that $b_t \cdot \theta \in c$ and $s \cap c = \emptyset$.

- $b_t \cdot \theta'_i \notin s_i$. By Lemma 18, the multi-clause $[s_i, c_i]$ is full and therefore $b_t \cdot \theta'_i \in c_i$. Hence, the clause $s_i \rightarrow b_t \cdot \theta'_i$ is included in H. $H \models s_i \rightarrow b_t \cdot \theta'_i \models s_i \cdot \hat{\theta} \rightarrow b_t \cdot \theta'_i \cdot \hat{\theta} \models s_x \rightarrow b_t \cdot \theta'_x$, this last step because $s_i \cdot \hat{\theta} \subseteq s_x$ and $\theta'_x = \theta'_i \cdot \hat{\theta}$. That is, $H \models s_x \rightarrow b_t \cdot \theta'_x$. But since $b_t \cdot \theta'_x \in c_x$, this contradicts the fact that $[s_x, c_x]$ is a counterexample.

□

And this concludes the proof of Lemma 20. ■

Corollary 21 *If a counterexample $[s_x, c_x]$ is appended to S, it is because there is no element in S violating a clause in $U(T)$ that is also violated by $[s_x, c_x]$.*

Proof. Were it the case, then by Lemma 20 the first element sharing some target clause would have been replaced instead of being appended. ■

Lemma 22 *Let $[s_1, c_1]$ and $[s_2, c_2]$ be two full multi-clauses. And let $[s, c]$ be any legal pairing between them. If $[s, c]$ violates a clause $\mathit{ineq}(s_t), s_t \rightarrow b_t$, then the following holds:*

1. *Both $[s_1, c_1]$ and $[s_2, c_2]$ cover $\mathit{ineq}(s_t), s_t \rightarrow b_t$.*
2. *At least one of $[s_1, c_1]$ or $[s_2, c_2]$ violates $\mathit{ineq}(s_t), s_t \rightarrow b_t$.*

Proof. By assumption, $\mathit{ineq}(s_t), s_t \rightarrow b_t$ is violated by $[s, c]$, i.e., there is a θ such that $s_t \cdot \theta \subseteq s$, $\mathit{ineq}(s_t) \cdot \theta \subseteq \mathit{ineq}(s)$ 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 also distinct terms appearing in s.

Let σ be the 1-1 legal matching inducing the pairing. The antecedent s is defined to be $\mathit{lgg}_{|\sigma}(s_1, s_2)$, 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 $\mathit{ineq}(s_t), s_t \rightarrow b_t$ 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 $\mathit{ineq}(s_t) \cdot \theta \cdot \theta_1 \subseteq \mathit{ineq}(s_1)$ and $\mathit{ineq}(s_t) \cdot \theta \cdot \theta_2 \subseteq \mathit{ineq}(s_2)$. Observe that all top-level terms appearing in s (remember, $s = \mathit{lgg}_{|\sigma}(s_1, s_2)$) 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 $\mathit{lgg}(t \cdot \theta \cdot \theta_1, t \cdot \theta \cdot \theta_2) = t \cdot \theta$. And, 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 $\mathit{lgg}_{|\sigma}(s_1, c_2) \cup \mathit{lgg}_{|\sigma}(c_1, s_2) \cup \mathit{lgg}_{|\sigma}(c_1, c_2)$. 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 $l \in \mathit{lgg}_{|\sigma}(c_1, \cdot)$, then $l \cdot \theta_1 \in c_1$. Equivalently, if $l \in \mathit{lgg}_{|\sigma}(\cdot, c_2)$, then $l \cdot \theta_2 \in c_2$. Therefore we get that if $b_t \cdot \theta \in \mathit{lgg}_{|\sigma}(c_1, \cdot)$, then $b_t \cdot \theta \cdot \theta_1 \in c_1$ and if $b_t \cdot \theta \in \mathit{lgg}_{|\sigma}(\cdot, c_2)$, 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 $\mathit{lgg}_{|\sigma}$. 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 $\mathit{ineq}(s_t), s_t \rightarrow b_t$, one of $[s_1, c_1]$ or $[s_2, c_2]$ violates $\mathit{ineq}(s_t), s_t \rightarrow b_t$. ■

Example 14 To illustrate the fact that we need a legal pairing to be able to prove the lemma above, we present the following counterexample. That is, we present two multi-clauses $[s_1, c_1]$, $[s_2, c_2]$, a non-legal matching σ and a clause $\mathit{ineq}(s_t), s_t \rightarrow b_t$ such that the non-legal pairing of $[s_1, c_1]$ and $[s_2, c_2]$ induced by σ violates $\mathit{ineq}(s_t), s_t \rightarrow b_t$ but none of $[s_1, c_1]$ and $[s_2, c_2]$ do.

- $[s_1, c_1] = [\{p(ffa, gffa)\}, \{q(fa)\}]$ with terms $\{a, fa, ffa, gffa\}$
  $\mathit{ineq}(s_1) = (a \neq fa \neq ffa \neq gffa)$.
- $[s_2, c_2] = [\{p(fb, gffc)\}, \{q(b)\}]$ with terms $\{b, c, fb, fc, ffc, gffc\}$.
- The matching σ is
```
[a - c => X]
[fa - b => Y]
[ffa - fb => fY]
[gffa - gffc => gffX]
```
- $[s, c] = [\{p(fY, gffX)\}, \{q(Y)\}]$.
- $\mathit{ineq}(s_t), s_t \rightarrow b_t = (x \neq fx \neq ffx \neq gffx \neq y \neq fy), p(fy, gffx) \rightarrow q(y)$.
- $\theta = \{x \mapsto X, y \mapsto Y\}$.
- $\theta_1 = \{X \mapsto a, Y \mapsto fa\}$.
- $\theta \cdot \theta_1 = \{x \mapsto a, y \mapsto fa\}$.

The multi-clause $[s, c]$ violates $\mathit{ineq}(s_t), s_t \rightarrow b_t$ via $\theta = \{x \mapsto X, y \mapsto Y\}$. But $[s_1, c_1]$ does not cover $\mathit{ineq}(s_t), s_t \rightarrow b_t$ because the condition $\mathit{ineq}(s_t) \cdot \theta \cdot \theta_1 \subseteq \mathit{ineq}(s_1)$ fails to hold:

$$(x \neq fx \neq ffx \neq gffx \neq y \neq fy) \cdot \theta \cdot \theta_1 = (a \neq \boxed{\text{fa}} \neq \boxed{\text{ffa}} \neq gffa \neq \boxed{\text{fa}} \neq \boxed{\text{ffa}})$$

Lemma 23 (Invariant) *Every time the algorithm is about to make an entailment equivalence query, it is the case that every multi-clause in S violates at least one of the clauses of $U(T)$ and every clause of $U(T)$ is violated by at most one multi-clause in S. In other words, it is not possible that two distinct multi-clauses in S violate the same clause in $U(T)$ simultaneously.*

Proof. To see that every multi-clause $[s, c] \in S$ violates at least one clause of $U(T)$, it suffices to observe that by Lemma 18 all counterexamples included in S are full positive multi-clauses with $c \neq \emptyset$. We can apply Corollary 7, and conclude that $[s, c]$ violates some clause of $U(T)$.

To see that no two different multi-clauses in S violate the same clause of $U(T)$, we proceed by induction on the number of iterations of the main loop in line 3 of the learning algorithm. In the first loop the lemma holds trivially (there are no elements in S). By the induction hypothesis we assume that the lemma holds before a new iteration of the loop. We will see that after completion of that iteration of the loop the lemma must also hold. Two cases arise.

The minimised counterexample $[s_x, c_x]$ is appended to S. By Corollary 21, we know that $[s_x, c_x]$ does not violate any clause in $U(T)$ also violated by some element $[s_i, c_i]$ in S. This, together with the induction hypothesis, assures that the lemma is satisfied in this case.

Some $[s_i, c_i]$ is replaced in S. We denote the updated sequence by S' and the updated element in S' by $[s'_i, c'_i]$. The induction hypothesis claims that the lemma holds for S. We have to prove that it also holds for S' as updated by the algorithm. Assume it does not. The only possibility is that the new element $[s'_i, c'_i]$ violates some clause of $U(T)$, say $\mathit{ineq}(s_t), s_t \to b_t$ also violated by some other element $[s_j, c_j]$ of S', with $j \neq i$. The multi-clause $[s'_i, c'_i]$ is a basic pairing of $[s_x, c_x]$ and $[s_i, c_i]$, and hence it is also legal. Applying Lemma 22 we conclude that one of $[s_x, c_x]$ or $[s_i, c_i]$ violates $\mathit{ineq}(s_t), s_t \to b_t$.

Suppose $[s_i, c_i]$ violates $\mathit{ineq}(s_t), s_t \to b_t$. This contradicts the induction hypothesis, since both $[s_i, c_i]$ and $[s_j, c_j]$ violate $\mathit{ineq}(s_t), s_t \to b_t$ in $U(T)$.

Suppose $[s_x, c_x]$ violates $\mathit{ineq}(s_t), s_t \to b_t$. If $j < i$, then $[s_x, c_x]$ would have refined $[s_j, c_j]$ instead of $[s_i, c_i]$ (Lemma 20). Therefore, $j > i$. But then we are in a situation where $[s_j, c_j]$ violates a clause also covered by $[s_i, c_i]$. By repeated application of Lemma 22, all multi-clauses in position i cover $\mathit{ineq}(s_t), s_t \to b_t$ during the history of S. Consider the iteration in which $[s_j, c_j]$ first violated $\mathit{ineq}(s_t), s_t \to b_t$. This could have happened by appending the counterexample $[s_j, c_j]$, which contradicts Lemma 20 since $[s_i, c_i]$ or an ancestor of it was covering $\mathit{ineq}(s_t), s_t \to b_t$ but was not replaced. Or it could have happened by refining $[s_j, c_j]$ with a pairing of a counterexample violating $\mathit{ineq}(s_t), s_t \to b_t$. But then, by Lemma 20 again, the element in position i should have been refined, instead of refining $[s_j, c_j]$. ■

Corollary 24 *The number of elements in S is bounded by m', the number of clauses in $U(T)$.*

Proof. Suppose there are more than m' elements in S. Since every $[s, c]$ in S violates some clause in $U(T)$, then it must be the case that two different elements in S violate the same clause of $U(T)$, since there are only m' clauses in $U(T)$, which contradicts Lemma 23. ■

Lemma 25 *Let $[s_x, c_x]$ be any minimised counterexample. Then, $|s_x| + |c_x| \leq st^a$.*

Proof. By Corollary 14 there are a maximum of t terms in a minimised counterexample. And there are a maximum of st^a different literals built up from t terms. ■

Lemma 26 *The algorithm makes $O(m'st^a)$ equivalence queries.*

Proof. The sequence S has at most m' elements. After every refinement, either one literal is dropped or some term is substituted by one of less weight. This can happen $m'st^a$ (to drop literals) plus $m't$ (to replace terms) times, that is $m'(t + st^a)$. We need m' extra calls to add all the counterexamples. That makes a total of $\underline{m'}(1 + t + \underline{st^a})$. That is $O(m'st^a)$. ■

Lemma 27 *The algorithm makes $O(se_t^{a+1})$ membership queries in any run of the minimisation procedure.*

Proof. To compute the first version of full multi-clause we need to test the se_t^a possible literals built up from e_t distinct terms appearing in s_x. Therefore, we make se_t^a initial calls.

Next, we note that the first version of c_x has at most se_t^a literals. The first loop (generalisation of terms) is executed at most e_t times, one for every term appearing in the first version of s_x. In every execution, at most $|c_x| \leq se_t^a$ membership calls are made. In this loop there are a total of se_t^{a+1} calls.

The second loop of the minimisation procedure is also executed at most e_t times, one for every term in s_x. Again, since at most se_t^a calls are made in the body on this second loop, the total number of calls is bounded by se_t^{a+1}.

This makes a total of $se_t^a + 2se_t^{a+1}$, that is $O(se_t^{a+1})$. ■

Lemma 28 *Given a matching, the algorithm makes at most st^a membership queries in the computation of any basic pairing.*

Proof. The number of literals in the consequent c of a pairing of $[s_x, c_x]$ and $[s_i, c_i]$ is bounded by the number of literals in s_x plus the number of literals in c_x. By Lemma 25, this is bounded by st^a. ■

Lemma 29 *The algorithm makes $O(m's^2t^ae_t^{a+1} + m'^2s^2t^{2a+k})$ membership queries.*

Proof. The main loop is executed as many times as equivalence queries are made. In every loop, the minimisation procedure is executed once and for every element in S, a maximum of t^k pairings are made.

This is: $\underbrace{sm't^a}_{\#iterations} \times \{ \underbrace{se_t^{a+1}}_{minim.} + \underbrace{m'}_{|S|} \cdot \underbrace{t^k}_{\#pairings} \cdot \underbrace{st^a}_{pairing} \} = O(m's^2t^ae_t^{a+1} + m'^2s^2t^{2a+k})$. ■

Theorem 30 *The algorithm exactly identifies every range restricted Horn expression 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.*

Proof. Follows from Lemmas 26 and 29. Notice that the membership calls take most of the running time. ■

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 any 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 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 completely inequated range restricted Horn expressions.

Given an interpretation I and a variable substitution θ mapping variables into domain objects of I, the truth value of the ground inequality literal $t \cdot \theta \neq t' \cdot \theta$ is true in I iff $t \cdot \theta$ and $t' \cdot \theta$ are mapped into different objects of I.

Looking at the way the transformation $U(T)$ described in Section 2.6 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 4 has to be slightly modified in order to achieve learnability of this class. In this case, all examples seen or output by the oracles are fully inequated, and so are the hypotheses H presented by the algorithm. Next, we will briefly describe what these modifications are, how they affect the proof of correctness and what the new bounds are.

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.

The second (and last) modification is in the computation of a pairing. Given a matching σ and two multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$, its pairing $[s, c]$ is computed as:

$$s = ineq(lgg_{|\sigma}(s_x^p, s_i^p)) \cup lgg_{|\sigma}(s_x^p, s_i^p)$$

$$c = lgg_{|\sigma}(s_x^p, c_i) \cup lgg_{|\sigma}(c_x, s_i^p) \cup lgg_{|\sigma}(c_x, c_i).$$

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. Exactly the same matchings as in the algorithm for range restricted Horn expressions are considered. That is, they have to be basic and they have to agree with the lgg table of s_x^p and s_i^p.

The proof of correctness uses the fact that the two modifications above imply that minimised counterexamples and pairings are fully inequated. Thus, every multi-clause in the sequence S is fully inequated. The rest of the proof is very similar to the one presented in Section 4. Its main aim consists in guaranteeing that every $[s, c]$ in S violates one clause in T and that no two different elements $[s_1, c_1]$ and $[s_2, c_2]$ in S violate the same clause of T, thus bounding the length of S by m, the number of clauses in the target expression T. The bounds on the number of queries are derived in the same fashion as in Section 4. Hence, we present the following theorem. Notice the exponential dependence on t disappears.

Theorem 31 *The modified algorithm exactly identifies every fully inequated range restricted Horn expression 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.*

Complete details and proof can be found in [AK00].

6 Comparison of results

	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	$FIRRHE$	$O(mst^a)$	$O(ms^2t^ae_t^{a+1} + m^2s^2t^{2a+k})$

This table contains the results obtained in [Kha99b] and in this paper for the respective learning algorithms. *$RRHE$* stands for *Range Restricted Horn Expressions* and *$FIRRHE$* for *Fully Inequated Range Restricted Horn Expressions*.

In the case of $RRHE$, the parameter m' has been substituted by its upper bound mt^t. There is an exponential dependence on the parameters a and t. Focusing only on t, we notice that in the number of membership queries the term $t^{3t+\cdots}$ appears in the result of [Kha99b]. This has been improved to $t^{2t+k+\cdots}$ in our version and it constitutes one of the main contributions of this paper. Notice that the exponential dependence on t disappears in the case of $FIRRHE$.

References

[AK00] 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.

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

[DRB92] L. De Raedt and M. Bruynooghe. An overview of the interactive concept learner and theory revisor CLINT. In S. Muggleton, editor, *Inductive Logic Programming*. Academic Press, 1992.

[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.

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

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

[MR94] Stephen Muggleton and Luc 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.

[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.

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

[Sha91] E. Y. Shapiro. Inductive inference of theories from facts. In J. L. Lassez and G. D. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 199–255. The MIT Press, 1991.