Learning Inequated Range Restricted Horn Expressions

by

Marta Arias, Roni Khardon

Informatics Research Report EDI-INF-RR-0011

Division of Informatics **March 2000**
http://www.informatics.ed.ac.uk/

Learning Inequated Range Restricted Horn Expressions

Marta Arias, Roni Khardon

Informatics Research Report EDI-INF-RR-0011

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

Abstract :
A learning algorithm for the class of inequated range restricted Horn expressions is presented and proved correct. 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. And every clause includes in its antecedent all inequalities possible between all terms appearing in it. 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.

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

Learning Inequated 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 *inequated range restricted Horn expressions* is presented and proved correct. 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. And every clause includes in its antecedent all inequalities possible between all terms appearing in it. 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.

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 *inequated 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. And every clause includes in its antecedent all inequalities possible between all terms appearing in it. The paper shows that small modifications to the algorithm and proof of [AK00] yield the learning result. Further background and related work appear in [Ari97, RT98, RS98, MF92, Kha99a, Kha99b].

The rest of the paper is organised as follows. Section 2 gives some preliminary definitions. The learning algorithm is presented in Section 3 and proved correct in Section 4.

2 Preliminaries

2.1 Inequated 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

*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.

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. An *inequality literal* or *inequation* is a literal of the form $t_1 \neq t_2$, where t_1, t_2 are terms.

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 inequalities between terms in s^p.

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. An *inequated range restricted Horn clause* is a range restricted Horn clause that includes in its antecedent all possible inequations between terms appearing in the non-inequational literals. An *inequated range restricted Horn expression* is a conjunction of inequated 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. A ground inequality literal $t \cdot \theta \neq t' \cdot \theta$ is true iff $t \cdot \theta$ and $t' \cdot \theta$ are mapped into a different object of I.

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

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

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.

Definition 7 (Covering multi-clause) A multi-clause $[s, c]$ *covers* a Horn clause $s_t \rightarrow b_t$ if there is a mapping θ from variables in s_t into terms in s such that $s_t \cdot \theta \subseteq s$. Equivalently, we say that $s_t \rightarrow b_t$ *is covered* by $[s, c]$.

Definition 8 (Violating multi-clause) A multi-clause $[s, c]$ *violates* a Horn clause $s_t \rightarrow b_t$ if there is a mapping θ from variables in s_t into terms in s such that $s_t \rightarrow b_t$ is covered by $[s, c]$ via θ and $b_t \cdot \theta \in c$. Equivalently, we say that $s_t \rightarrow b_t$ *is violated by* $[s, c]$.

Definition 9 Let s be any set of atoms. Then $ineq(s) = \{t \neq t' \mid t, t' \textit{ are distinct terms in } s\}$. A set of atoms and inequalities s is *completely inequated* if $s^{\neq} = ineq(s^p)$. Similarly, a multi-clause is completely inequated if its antecedent is.

2.4 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.

- *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 1: The lgg algorithm

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 1.

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 1 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.

- $s_1 = \{p(a, f(b)), p(g(a, x), c), q(a)\}$
- $s_2 = \{p(z, f(2)), q(z)\}$
- 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]
```

```
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 $\bigwedge_{[s,c]\in S} \{s \to b \mid b \in c\}$.
4. Return $H$
```

Figure 2: The learning algorithm

3 The Algorithm

The algorithm keeps a sequence S of representative counterexamples. The hypothesis H is generated from this sequence, and the main task of the algorithm is to *refine* the counterexamples in 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"[3]. 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:

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

[3]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 $x_{f(\bar{t})}$.
        – Let $[s''_x, c'_x]$ be the multi-clause obtained from $[s'_x, c'_x]$ after arranging inequalities
          so the resulting clause is in fully inequated form.
        – 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 $t$.
        – Let $[s''_x, c'_x]$ be the multi-clause obtained from $[s'_x, c'_x]$ after arranging inequalities so
          the resulting clause is in fully inequated form.
        – 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 3: 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, possibly arranging superfluous inequalities in order to get a fully inequated counterexample. This is done by checking if any of the proper subterms of the term to be generalised still appears in some non-inequational literal. If not, then all inequalities including such subterm are removed, otherwise they are kept. Notice we only need to check inequalities including proper subterms of the generalised term. Then it is checked 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, arrange inequalities as in previous step 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 2 This example illustrates the behaviour of the minimisation procedure. f, g stand for functional symbols of 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 $T = \{[(x \neq y \neq fy), p(x, fy) \to r(y)], [(z \neq fz), q(fz) \to s(z)]\}$.

- Hypothesis $H = \{(x \neq fx \neq ffx), q(ffx) \to s(fx)\}$.

- Counterexample x as given by the $EntEQ$ oracle:

$$(a \neq \underline{b} \neq c \neq ga \neq \underline{fb} \neq fc \neq \underline{ffb} \neq gfc), p(ga, ffb), \underline{q(ffb)}, r(gfc) \to r(fb).$$

- After step 2:

$$s_x = \{(a \neq b \neq c \neq ga \neq fb \neq fc \neq ffb \neq gfc), p(ga, ffb), q(ffb), r(gfc), \underline{s(fb)}\}.$$

- After step 3: $c_x = rhs(s_x) = \{r(fb)\}$.

- The first version of full counterexample $[s_x, c_x]$:

$$[\{(a \neq b \neq c \neq ga \neq fb \neq fc \neq ffb \neq gfc), p(ga, ffb), q(ffb), r(gfc), s(fb)\}, \{r(fb)\}].$$

- **Generalising terms.** The list of functional terms is $[gfc, ffb, fc, fb, ga, c, b, a]$.

– Generalise term $gfc \mapsto x_1$:
— $[s'_x, c'_x] = [\{(a \neq b \neq c \neq ga \neq fb \neq fc \neq ffb \neq x_1), p(ga, ffb), q(ffb), r(x_1), s(fb)\}, \{r(fb)\}]$.
— $[s''_x, c'_x] = [\{(a \neq b \neq ga \neq fb \neq ffb \neq x_1), p(ga, ffb), q(ffb), r(x_1), s(fb)\}, \{r(fb)\}]$.
— $rhs(s''_x, c'_x) = \{r(fb)\} \neq \emptyset$.
— $[s_x, c_x] = [\{(a \neq b \neq ga \neq fb \neq ffb \neq x_1), p(ga, ffb), q(ffb), r(x_1), s(fb)\}, \{r(fb)\}]$.
— The list of terms still to check is $[ffb, fb, ga, b, a]$.

– Generalise term $ffb \mapsto x_2$:
— $[s'_x, c'_x] = [\{(a \neq b \neq ga \neq fb \neq x_2 \neq x_1), p(ga, x_2), q(x_2), r(x_1), s(fb)\}, \{r(fb)\}]$.
— $[s''_x, c'_x] = [\{(a \neq b \neq ga \neq fb \neq x_2 \neq x_1), p(ga, x_2), q(x_2), r(x_1), s(fb)\}, \{r(fb)\}]$.
— $rhs(s''_x, c'_x) = \emptyset$.
— $[s_x, c_x] = [\{(a \neq b \neq ga \neq fb \neq ffb \neq x_1), p(ga, ffb), q(ffb), r(x_1), s(fb)\}, \{r(fb)\}]$.
— The list of terms still to check is $[fb, ga, b, a]$.

– Generalise term $fb \mapsto x_3$:
— $[s'_x, c'_x] = [\{(a \neq b \neq ga \neq x_3 \neq fx_3 \neq x_1), p(ga, fx_3), q(fx_3), r(x_1), s(x_3)\}, \{r(x_3)\}]$.
— $[s''_x, c'_x] = [\{(a \neq ga \neq x_3 \neq fx_3 \neq x_1), p(ga, fx_3), q(fx_3), r(x_1), s(x_3)\}, \{r(x_3)\}]$.
— $rhs(s''_x, c'_x) = \{r(x_3)\} \neq \emptyset$.
— $[s_x, c_x] = [\{(a \neq ga \neq x_3 \neq fx_3 \neq x_1), p(ga, fx_3), q(fx_3), r(x_1), s(x_3)\}, \{r(x_3)\}]$.
— The list of terms still to check is $[ga, a]$.

– Generalise term $ga \mapsto x_4$:
— $[s'_x, c'_x] = [\{(a \neq x_4 \neq x_3 \neq fx_3 \neq x_1), p(x_4, fx_3), q(fx_3), r(x_1), s(x_3)\}, \{r(x_3)\}]$.
— $[s''_x, c'_x] = [\{(x_4 \neq x_3 \neq fx_3 \neq x_1), p(x_4, fx_3), q(fx_3), r(x_1), s(x_3)\}, \{r(x_3)\}]$.
— $rhs(s''_x, c'_x) = \{r(x_3)\} \neq \emptyset$.
— $[s_x, c_x] = [\{(x_4 \neq x_3 \neq fx_3 \neq x_1), p(x_4, fx_3), q(fx_3), r(x_1), s(x_3)\}, \{r(x_3)\}]$.
— 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:
— $[s'_x, c'_x] = [\{(x_4 \neq x_3 \neq fx_3), p(x_4, fx_3), q(fx_3), s(x_3)\}, \{r(x_3)\}]$.
— $[s''_x, c'_x] = [\{(x_4 \neq x_3 \neq fx_3), p(x_4, fx_3), q(fx_3), s(x_3)\}, \{r(x_3)\}]$.
— $rhs(s''_x, c'_x) = \{r(x_3)\} \neq \emptyset$.
— $[s_x, c_x] = [\{(x_4 \neq x_3 \neq fx_3), p(x_4, fx_3), q(fx_3), s(x_3)\}, \{r(x_3)\}]$.
— The list of terms still to check is $[x_3, x_4, fx_3]$.

– Drop term x_3:
— $[s'_x, c'_x] = [\{\}, \{\}]$.
— $[s''_x, c'_x] = [\{\}, \{\}]$.
— $rhs(s''_x, c'_x) = \emptyset$.
— $[s_x, c_x] = [\{(x_4 \neq x_3 \neq fx_3), p(x_4, fx_3), q(fx_3), s(x_3)\}, \{r(x_3)\}]$.
— The list of terms still to check is $[x_4, fx_3]$.

– Drop term x_4:
— $[s'_x, c'_x] = [\{(x_3 \neq fx_3), q(fx_3), s(x_3)\}, \{r(x_3)\}]$.
— $[s''_x, c'_x] = [\{(x_3 \neq fx_3), q(fx_3), s(x_3)\}, \{r(x_3)\}]$.
— $rhs(s''_x, c'_x) = \emptyset$.
— $[s_x, c_x] = [\{(x_4 \neq x_3 \neq fx_3), p(x_4, fx_3), q(fx_3), s(x_3)\}, \{r(x_3)\}]$.
— The list of terms still to check is $[fx_3]$.

– Drop term fx_3:
— $[s'_x, c'_x] = [\{(x_4 \neq x_3), s(x_3)\}, \{r(x_3)\}]$.
— $[s''_x, c'_x] = [\{s(x_3)\}, \{r(x_3)\}]$.
— $rhs(s''_x, c'_x) = \emptyset$.
— $[s_x, c_x] = [\{(x_4 \neq x_3 \neq fx_3), p(x_4, fx_3), q(fx_3), s(x_3)\}, \{r(x_3)\}]$.
— 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 2.4 in Page 3).

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 potentially 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. Typically, we denote a matching by the Greek letter σ.

Example 3 Consider:

$[s_x, c_x] = [\{(a \neq b), p(a, b)\}, \{q(a)\}]$	with terms $\{a, b\}$
$[s_i, c_i] = [\{(1 \neq 2 \neq f(1)), p(f(1), 2)\}, \{q(f(1))\}]$	with terms $\{1, 2, f(1)\}$

The possible matchings are:

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

Definition 13 (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 14 (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 4 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 5 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 15 (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. Notice this operation is not symmetric, and so is the construction. 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 6 No parentheses for functions are written.

- $s_x = \{(a \neq x \neq fx), p(a, fx)\}$ with terms $\{a, x, fx\}$.
- $s_i = \{(a \neq 1 \neq 2 \neq f1), p(a, f1), p(a, 2)\}$ 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

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^p and s_i^p.*
3. *Let σ be the basic matching between terms of $[s_x, c_x]$ and $[s_i, c_i]$.*
4. *Compute the extension of σ, restricted to the table TBL.*
   If it is not possible to compute the lgg for some pair of terms in σ, then return $[\emptyset, \emptyset]$.
5. *Else, let $[s, c]$ be $[ineq(lgg_{|\sigma}(s_x^p, s_i^p)) \cup lgg_{|\sigma}(s_x^p, s_i^p), lgg_{|\sigma}(s_x^p, c_i) \cup lgg_{|\sigma}(c_x, s_i^p) \cup lgg_{|\sigma}(c_x, c_i)]$.*
6. *Return $[s, c]$.*

Figure 4: The pairing procedure

We start our explanation of the pairing procedure. This procedure is described in Figure 4. 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 predicate part of the antecedent s^p of the pairing is computed as the lgg of s_x^p and s_i^p 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^p and s_i^p (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 extended entry `[f(f(1)) - f(a) => f(X)]` appears in the matching. Similarly, suppose $p(a)$ appears in both s_x^p and s_i^p. Their lgg $p(a)$ will not be included unless the entry `[a - a => a]` appears in the matching. The inequated part of the pairing's antecedent $s^{\neq}$ is computed as $s^{\neq} = ineq(s^p)$, creating thus a fully inequated multi-clause.

To compute the consequent c of the pairing, the union of $lgg_{|\sigma}(s_x^p, c_i)$, $lgg_{|\sigma}(c_x, s_i^p)$ and $lgg_{|\sigma}(c_x, c_i)$ is computed. Note that in the consequent all those possible couples among $\{s_x^p, c_x, s_i^p, c_i\}$ are included except s_x^p, s_i^p, 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 $lgg(s_x^p, s_i^p)$. To summarise:

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

Note that when computing any of the lggs, the same table is used. That is, the same pair of terms will be bound to the same expression in any of the four possible lggs that are computed in a pairing: $lgg_{|\sigma}(s_x^p, s_i^p), lgg_{|\sigma}(s_x^p, c_i), lgg_{|\sigma}(c_x, s_i^p)$ and $lgg_{|\sigma}(c_x, c_i)$.

Example 7 Computation of a pairing's antecedent.

- $s_x = \{(a \neq x \neq fx), p(a, fx)\}$ with terms $\{a, x, fx\}$.
- $s_i = \{(a \neq 1 \neq 2 \neq f1), p(a, f1), p(a, 2)\}$ with terms $\{a, 1, 2, f1\}$.
- $s_x^p = \{p(a, fx)\}$ with terms $\{a, x, fx\}$.

- $s_i^p = \{p(a, f1), p(a, 2)\}$ with terms $\{a, 1, 2, f1\}$.
- The *lgg* of s_x^p and s_i^p 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 6 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 $s = ineq(lgg_{|\sigma}(s_x^p, s_i^p)) \cup lgg_{|\sigma}(s_x^p, s_i^p) = \{(a \neq X \neq fX), p(a, fX)\}$

Lemma 1 *Let $[s_x, c_x]$ and $[s_i, c_i]$ be two fully inequated multi-clauses, σ a legal matching and $[s, c]$ the legal pairing induced by σ. Then there exist θ_x and θ_i such that $s \cdot \theta_x \subseteq s_x$ and $s \cdot \theta_i \subseteq s_i$.*

Proof. The legal matching σ includes entries of the form `[`t_x `-` t_i `=>` $lgg(t_x, t_i)$`]`, where t_x and t_i are terms in s_x and s_i, respectively. We define $\theta_x = \{lgg(t_x, t_i) \mapsto t_x\}$ and $\theta_i = \{lgg(t_x, t_i) \mapsto t_i\}$ whenever $lgg(t_x, t_i)$ is a variable. Therefore an entry `[`t_x `-` t_i `=>` $lgg(t_x, t_i)$`]` can be also viewed as `[`$t \cdot \theta_x$ `-` $t \cdot \theta_i$ `=>` t`]`, since the *lgg* respects the functional structure of terms.

Let $p(\overline{t})$ be any literal in s^p. Since s^p is the result of restricting $lgg(s_x^p, s_i^p)$ to σ, all terms in $\overline{t}$ must appear in the extension of σ. And since σ includes entries `[`$t \cdot \theta_x$ `-` $t \cdot \theta_i$ `=>` t`]`, both literals $p(\overline{t \cdot \theta_x}) = p(\overline{t}) \cdot \theta_x$ and $p(\overline{t \cdot \theta_i}) = p(\overline{t}) \cdot \theta_i$ appear in s_x^p and s_i^p, respectively. We conclude $s^p \cdot \theta_x \subseteq s_x^p$ and $s^p \cdot \theta_i \subseteq s_i^p$.

Let $t \neq t'$ be an inequation in $s^{\neq}$. Since s is fully inequated, the terms t and t' must appear in some literal in s^p, maybe as subterms of some bigger terms. The matching σ is legal and therefore its extension contains every term (and subterm) appearing in any literal in s^p. Therefore, t and t' appear in the extension of σ. The entries of σ for the two terms t and t' are `[`$t \cdot \theta_x$ `-` $t \cdot \theta_i$ `=>` t`]` and `[`$t' \cdot \theta_x$ `-` $t' \cdot \theta_i$ `=>` t'`]`. σ is 1-1, thus concluding that $t \cdot \theta_x \neq t' \cdot \theta_x$ and $t \cdot \theta_i \neq t' \cdot \theta_i$. That is, $t \cdot \theta_x$ and $t' \cdot \theta_x$ correspond to syntactically different terms in s_x^p. Moreover, $t \cdot \theta_x$ and $t' \cdot \theta_x$ also appear in literals of s_x^p, and since s_x is fully inequated, the inequation $t \cdot \theta_x \neq t' \cdot \theta_x$ appears in $s_x^{\neq}$. Hence, $s^{\neq} \cdot \theta_x \subseteq s_x^{\neq}$. An analogous analysis can be done for $s_i^{\neq}$. And we conclude $s \cdot \theta_x \subseteq s_x$ and $s \cdot \theta_i \subseteq s_i$. ■

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 *lgg* table of s_x^p and s_i^p, where $[s_x, c_x]$ and $[s_i, c_i]$ are the multi-clauses to be paired. 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.

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 T but no two distinct elements of S violate the same clause of T (Lemma 25). The bound on the length of S is therefore m, the number of clauses of the target T.

To see that every element in S violates some clause in T, it is shown that all counterexamples in S are full multi-clauses w.r.t. the target expression T (Lemma 20) and that any full multi-clause must violate some clause in 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 22 is used in this proof and it constitutes the most important lemma in our analysis. Lemma 22 states that if a minimised multi-clause $[s_x, c_x]$ violates some clause C in 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 24 stating that if a legal pairing violates some clause C, then the clauses that originate the pairing cover the C and at least one of them violates it, prove that there cannot be two different elements in S violating the same clause in 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

Definition 16 An inequated range restricted clause $s \rightarrow b$ is *non-trivial* if its antecedent is satisfiable. A multi-clause is non-trivial if its antecedent is satisfiable.

Notice that the only way that a clause can be trivial in our language is by having an inequation $t \neq t$ in its antecedent.

Lemma 2 *Let T be any Horn expression, $s \rightarrow b$ any non-trivial fully inequated clause and θ any substitution that does not unify any terms in s. It is the case that if $T \models s \rightarrow b$, then $T \models s \cdot \theta \rightarrow b \cdot \theta$ and $s \cdot \theta \rightarrow b \cdot \theta$ is non-trivial range restricted clause.*

Proof. If $s \rightarrow b$ is non-trivial and implied by T, then $s \rightarrow b$ is range restricted since T also is. The clause $s \cdot \theta \rightarrow b \cdot \theta$ is non-trivial, since no unification occurs when applying θ. It also holds that $T \models s \cdot \theta \rightarrow b \cdot \theta$, and hence $s \cdot \theta \rightarrow b \cdot \theta$ must be range restricted. ■

Our proof will consider clauses such as $s \rightarrow b$ and clauses derived from these by substitution $(s \rightarrow b) \cdot \theta$. We therefore need to make sure that θ does not unify terms t, t' for which $t \neq t'$ appears in s. Lemma 2 implies that this is sufficient. We will assume implicitly that clauses involved in the analysis are non-trivial. Notice that by definition any fully inequated clause is non-trivial.

Lemma 3 *Let $[s, c]$ be a completely inequated multi-clause covering some completely inequated clause $s_t \rightarrow b_t$ via substitution θ. Then θ does not unify terms of s_t. That is, there are no two distinct terms t, t' in s_t such that $t \cdot \theta = t' \cdot \theta$.*

Proof. The multi-clause $[s, c]$ covers $s_t \rightarrow b_t$. That is, $s_t \cdot \theta \subseteq s$. Let t, t' be any two distinct terms in s_t. The antecedent s_t is completely inequated, and hence it contains the inequality $t \neq t'$. Since $s_t \cdot \theta \subseteq s$, it follows that $t \cdot \theta \neq t' \cdot \theta \in s$. And s is fully inequated, therefore it will only contain inequation $t \cdot \theta \neq t' \cdot \theta$ if $t \cdot \theta$ and $t' \cdot \theta$ are two different terms, i.e. t and t' are not unified by θ. ■

Corollary 4 *Let T be a completely inequated range restricted Horn expression, $s_t \rightarrow b_t$ a completely inequated clause such that $T \models s_t \rightarrow b_t$, and $[s, c]$ any non-trivial and correct clause covering $s_t \rightarrow b_t$ via some substitution θ. Then, $T \models s \rightarrow b_t \cdot \theta$.*

Proof. Both s_t and s are completely inequated by hypothesis, and θ shows covering of $s_t \rightarrow b_t$ by $[s, c]$. By Lemma 3, we conclude θ is non-unifying for terms in s_t. Lemma 2 implies $T \models s_t \cdot \theta \rightarrow b_t \cdot \theta$, since by hypothesis $T \models s_t \rightarrow b_t$. We know that $s_t \cdot \theta \subseteq s$, and hence we conclude $T \models s \rightarrow b_t \cdot \theta$. ■

Lemma 5 *If $[s, c]$ is a non-trivial positive example for an inequated range restricted Horn expression T, then there is some clause $s_t \rightarrow b_t$ of T such that $s_t \cdot \theta \subseteq s$ and $b_t \cdot \theta \notin s$, where θ is some substitution mapping variables of s_t into terms of s. That is, $s_t \rightarrow b_t$ 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^p (with the corresponding new names for the terms), that is, $ext(I) = s^p \cdot \sigma$, where the top-level terms in s^p are substituted by their image in D_I given by σ.

It is easy to see that this I falsifies $[s, c]$. All inequations in s are satisfied because every term is mapped into a different object in D_I. And all atoms in s are also satisfied since $ext(I)$ has been defined precisely as s interpreted in I. But all literals $b \in c$ are not satisfied because $s \cap c = \emptyset$ and hence their interpretation in I does not appear in $ext(I)$.

And since $I \not\models [s, c]$ and $T \models [s, c]$, we conclude that $I \not\models T$. That is, there is a clause $s_t \rightarrow b_t$ in T such that $I \not\models C$ and there is a substitution θ' from variables in s_t into domain objects of I such that $s_t^p \cdot \theta' \subseteq ext(I)$, $b_t \cdot \theta' \notin ext(I)$ and every different term in s_t is interpreted as a different object in D_I, since $s_t^{\neq}$ also has to be satisfied. The set s_t is fully inequated, hence $s_t^{\neq} = ineq(s_t^p)$.

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 $ext(I)$.

Property $s_t^p \cdot \theta' \subseteq ext(I)$ implies $s_t^t \cdot \underbrace{\theta' \cdot \sigma^{-1}}_{\theta} \subseteq \underbrace{ext(I) \cdot \sigma^{-1}}_{s^p}$, i.e., $s_t^p \cdot \theta \subseteq s^p$.

The fact that $t \cdot \theta' \neq t' \cdot \theta'$ for every distinct terms t, t' of s_t implies that $t \cdot \underbrace{\theta' \cdot \sigma^{-1}}_{\theta} \neq t' \cdot \underbrace{\theta' \cdot \sigma^{-1}}_{\theta}$, since every distinct object is mapped into a distinct term of s by σ^{-1}. Hence, $s_t^{\neq} \cdot \theta \subseteq s^{\neq}$ and therefore $s_t \cdot \theta \subseteq s$ as required.

Property $b_t \cdot \theta' \notin ext(I)$ implies $b_t \cdot \underbrace{\theta' \cdot \sigma^{-1}}_{\theta} \notin \underbrace{ext(I) \cdot \sigma^{-1}}_{s}$, i.e., $b_t \cdot \theta \notin s$. ■

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

Proof. By Lemma 5, there is a mapping θ such that $[s, c]$ covers some clause $s_t \rightarrow b_t$ in T and $b_t \cdot \theta \notin s$. T is an inequated range restricted Horn expression, and hence s_t is completely inequated, and so is s by hypothesis. $s_t \rightarrow b_t$ is a clause in T, and $T \models s_t \rightarrow b_t$. Using Corollary 4, we conclude that $T \models s \rightarrow b_t \cdot \theta$.

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

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

Proof. The conditions of Lemma 6 are satisfied. ■

Lemma 8 *Every minimised counterexample $[s_x, c_x]$ 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 is 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 $rhs(s_x)$ and all possible consequents are tried out.

We will prove that after generalising a term t the resulting counterexample is still exhaustive. Let $[s_x, c_x]$ be the multi-clause before generalising t, $[s'_x, c'_x]$ the multi-clause after t has been replaced by the new variable x_t and $[s''_x, c'_x]$ after inequalities have been arranged. To see that $[s''_x, rhs(s''_x, c'_x)]$ is exhaustive it suffices to see that $[s''_x, c'_x]$ is. Let the substitution θ_t be $\{t \mapsto x_t\}$. That is:

- $[s_x, c_x]$ is full by assumption.
- $[s'_x, c'_x] = [s_x \cdot \theta_t, c_x \cdot \theta_t]$. We know that $s_x = s'_x \cdot \theta_t^{-1}$ and $c'_x = c_x \cdot \theta_t$, because x_t is a new variable that does not appear in s_x.
- $[s''_x, c'_x]$ is the clause where inequations containing terms not in ${s'_x}^p$ have been removed from s'_x. We know that $s''_x \subseteq s'_x$, since we obtain s''_x by removing inequations from s'_x.

We will now see that any literal b implied by s''_x w.r.t. T is included in c'_x, and hence $[s''_x, c'_x]$ is also exhaustive. Suppose, then, that $T \models s''_x \rightarrow b$ with $b \notin s''_x$ and b not being an inequation. Since $s''_x \subseteq s'_x$ we get $T \models s'_x \rightarrow b$ and hence $T \models \underbrace{s'_x \cdot \theta_t^{-1}}_{s_x} \rightarrow b \cdot \theta_t^{-1}$. This is true since θ_t is non-unifying, since it only maps variable x_t into a term that does not appear anywhere else in s'_x (all its occurrences have been substituted by x_t in s'_x). Thus, $T \models s_x \rightarrow b \cdot \theta_t^{-1}$. Also, $b \cdot \theta_t^{-1} \notin s_x$ since $b \notin s'_x$ by assumption. By induction hypothesis, $[s_x, c_x]$ is full and hence exhaustive, therefore $b \cdot \theta_t^{-1} \in c_x$. And hence, $\underbrace{b \cdot \theta_t^{-1} \cdot \theta_t}_{b} \in \underbrace{c_x \cdot \theta_t}_{c'_x}$ as required.

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, $[s'_x, c'_x]$ after removing it and $[s''_x, c'_x]$ after inequalities have been arranged. It is clear that $s''_x \subseteq s'_x \subseteq s_x$ and $c'_x \subseteq c_x$ since all have been obtained by removing literals only. Assume $[s_x, c_x]$ is full. Suppose $T \models s''_x \rightarrow b$. T is range restricted and hence b uses terms in s''_x only. Since $s''_x \subseteq s_x$, we conclude $T \models s_x \to b$. The literal b is not an inequation and does not contain t, therefore $b \notin s_x$. The multi-clause $[s_x, c_x]$ has been assumed to be exhaustive, and so $b \in c_x$. Moreover, b is not removed from c_x since only literals containing t are removed. Therefore, $b \in c'_x$ as required. ■

Definition 17 (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 $b \in c$, $H \not\models s \to b$.

Lemma 9 *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 10 *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 $H \not\models s_x \to b$. By Lemma 8, we know that $[s_x, c_x]$ as output by the minimisation procedure 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 9). The antecedent of the multi-clause s_x is set to be $\{b \mid H \models \mathit{antecedent}(x) \to b\}$. 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 $\mathit{antecedent}(x)$ w.r.t. H 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 still 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, $[s'_x, c'_x]$ after generalising t and $[s''_x, c'_x]$ after arranging inequalities. Assume $[s_x, c_x]$ is a positive counterexample. Let θ_t be the substitution $\{t \mapsto x_t\}$. As in Lemma 8, $s_x \cdot \theta_t = s'_x$, $c'_x = c_x \cdot \theta_t$, $s_x = s'_x \cdot \theta_t^{-1}$, $c'_x \cdot \theta_t^{-1} = c_x$ and also $s''_x \subseteq s'_x$. Suppose by way of contradiction that $H \models s''_x \to b'$, for some $b' \in c'_x$. This implies $H \models s'_x \to b'$ since $s''_x \subseteq s'_x$. Then, $H \models s'_x \cdot \theta_t^{-1} \to b' \cdot \theta_t^{-1}$. And we get that $H \models s_x \to b' \cdot \theta_t^{-1}$. Note that $b' \in c'_x$ implies that $b' \cdot \theta_t^{-1} \in \underbrace{c'_x \cdot \theta_t^{-1}}_{c_x}$. 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 $[s''_x, c'_x]$ is a counterexample.

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 removal of literals and $[s''_x, c_x]$ the multi-clause after arranging inequalities. It is the case that $s''_x \subseteq s'_x \subseteq s_x$ and $c'_x \subseteq c_x$. It holds that $s''_x \to b \models s'_x \to b \models s_x \to b$ for any literal b. For all $b \in c_x$, it holds that $H \not\models s_x \to b$, and hence $H \not\models s''_x \to b$. Therefore, for all $b \in c'_x$ it holds that $H \not\models s''_x \to b$ and $[s''_x, c'_x]$ is a counterexample. ■

Lemma 11 *Every minimised counterexample is completely inequated.*

Proof. First we note that the counterexample x produced by the equivalence oracle is completely inequated. The first version of s_x is $\{b \mid H \models \mathit{antecedent}(x) \to b\}$. Notice $\mathit{antecedent}(x)$ contains all inequations, and every literal b included uses terms already appearing in $\mathit{antecedent}(x)$, therefore no new inequalities are needed and this first version is completely inequated. After some term has been generalised, notice that inequalities are arranged in such a way that the new antecedent remains fully inequated, and the same happens when some term is dropped. Therefore, a minimised counterexample is completely inequated. ■

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 $s_t \to b_t$ of 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. The multi-clause $[s_x, c_x]$ is violating $s_t \to b_t$, hence there must exist a substitution θ from variables in s_t into terms in s_x such that $s_t \cdot \theta \subseteq s_x$ 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 $s_t \to b_t$ via $\theta \cdot \theta_t$. By hypothesis we know that $s_t \cdot \theta \subseteq s_x$. This implies that $s_t \cdot \theta \cdot \theta_t \subseteq s_x \cdot \theta_t = s'_x$. Similarly, $b_t \cdot \theta \in c_x$ implies $b_t \cdot \theta \cdot \theta_t \in c_x \cdot \theta_t = c'_x$. And hence $s_t \to b_t$ is violated by $[s'_x, c'_x]$ via $\theta \cdot \theta_t$.

It is left to show that the multi-clause $[s''_x, c'_x]$ obtained from $[s'_x, c'_x]$ still violates $s_t \to b_t$. To see this, we will use the same substitution $\theta \cdot \theta_t$. To show violation, we have to prove the two conditions $s_t \cdot \theta \cdot \theta_t \subseteq s''_x$ and $b_t \cdot \theta \cdot \theta_t \in c'_x$. The second condition has been shown already in the previous paragraph, hence we only need to make sure that $s_t \cdot \theta \cdot \theta_t \subseteq s''_x$. The sets s'_x and s''_x only differ in that s''_x may contain a few less inequalities than s'_x. Therefore, $s^p_t \cdot \theta \cdot \theta_t \subseteq s''^p_x \subseteq s''_x$. To see that $s^{\neq}_t \cdot \theta \cdot \theta_t \subseteq s''_x$, suppose t and t' are any two distinct terms in s_t, i.e., $t \neq t' \in s_t$. $s_t \cdot \theta \cdot \theta_t \subseteq s'_x$ implies $t \cdot \theta \cdot \theta_t \neq t' \cdot \theta \cdot \theta_t \in s'_x$. s_t is fully inequated, therefore t and t' appear in s^p_t. And therefore, $t \cdot \theta \cdot \theta_t$ and $t' \cdot \theta \cdot \theta_t$ appear in ${s'_x}^p$ and hence in s''^p_x. s''^p_x is also fully inequated, therefore $t \cdot \theta \cdot \theta_t \neq t' \cdot \theta \cdot \theta_t \in s''_x$. And $[s''_x, c'_x]$ violates $s_t \to b_t$, $\mathit{rhs}(s''_x, c'_x) \neq \emptyset$ and t is generalised. ■

Lemma 13 *Let $[s_x, c_x]$ be any minimised counterexample with n_x distinct terms. And let $s_t \to b_t$ be any clause of T violated by $[s_x, c_x]$ containing n_t distinct terms. Then, $n_x = n_t$.*

Proof. The multi-clause $[s_x, c_x]$ violates $s_t \to b_t$. Therefore there is a θ mapping variables in s_t showing this violation. By Lemma 11 the multi-clause $[s_x, c_x]$ is completely inequated and so is $s_t \to b_t$ since it appears in T. By Lemma 3, the substitution θ does not unify terms in s_t. 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 $l \in s_x - s_t \cdot \theta$ 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). That is, $[s'_x, c'_x]$ violates $s_t \to b_t$. By a similar argument as in Lemma 12, the multi-clause $[s''_x, c'_x]$ also violates $s_t \to b_t$. And therefore, $\mathit{rhs}(s''_x, c'_x) \neq \emptyset$ and such a term t cannot exist. We conclude $n_t = n_x$. ■

Corollary 14 *Let $[s_x, c_x]$ be any minimised counterexample containing n_x distinct terms. Then $n_x \leq t$.*

Proof. Follows from the fact that that any n_t as in the previous lemma is bounded by t. ■

Lemma 15 *Let $[s_i, c_i]$ be any completely inequated multi-clause covering some clause $s_t \rightarrow b_t$ of T. Let n_i and n_t be the number of distinct terms in s_i and s_t, respectively. Then, $n_t \leq n_i$.*

Proof. Since $[s_i, c_i]$ covers the clause $s_t \rightarrow b_t$, there is a θ s.t. $s_t \cdot \theta \subseteq s_i$. By Lemma 3 , θ does not unify terms in s_t. 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_i has at least as many terms as s_t and the result follows. ■

Corollary 16 *Let $s_t \rightarrow b_t$ be a clause of T with n_t distinct terms. Let $[s_x, c_x]$ be a minimised counterexample with n_x distinct terms such that $[s_x, c_x]$ violates the clause $s_t \rightarrow b_t$. Let $[s_i, c_i]$ be a multi-clause with n_i terms covering the clause $s_t \rightarrow b_t$. 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 $[s, c]$ be a pairing of two multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$. Then the multi-clause $[s, c]$ is completely inequated.*

Proof. Any pairing is completely inequated by construction, since its antecedent s is computed as $ineq(lgg_{|\sigma}(s_x^p, s_i^p)) \cup lgg_{|\sigma}(s_x^p, s_i^p)$. ■

Lemma 18 *Let $[s_x, c_x]$ and $[s_i, c_i]$ be two completely inequated and 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 as computed in our algorithm. 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 suffices to show that $[s, c]$ is exhaustive. That is, whenever $T \models s \rightarrow b$ and $b \notin s$, then $b \in c$. Suppose, then, that $T \models s \rightarrow b$ with $b \notin s$. By Lemma 1, there exist θ_x and θ_i such that $s \cdot \theta_x \subseteq s_x$ and $s \cdot \theta_i \subseteq s_i$. All s, s_i and s_x are completely inequated by hypothesis and by Lemma 17. Hence, those inclusions imply that both θ_x and θ_i do not unify terms of s. And it follows from Lemma 2 that $T \models s \rightarrow b$ implies both $T \models s \cdot \theta_x \rightarrow b \cdot \theta_x$ and $T \models s \cdot \theta_i \rightarrow b \cdot \theta_i$. Let $b_x = b \cdot \theta_x$ and $b_i = b \cdot \theta_i$. Finally, we obtain that $T \models s_x \rightarrow b_x$ and $T \models s_i \rightarrow b_i$. 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 and s fully inequated and hence non-trivial. 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, $b \in lgg_{|\sigma}(s_x, c_i) \cup lgg_{|\sigma}(c_x, s_i) \cup lgg_{|\sigma}(c_x, c_i)$. Notice also that b is an atom, and therefore $b \in lgg_{|\sigma}(s_x^p, c_i) \cup lgg_{|\sigma}(c_x, s_i^p) \cup lgg_{|\sigma}(c_x, c_i) = c$. ■

Lemma 19 *Every element $[s, c]$ appearing in the sequence S is completely inequated.*

Proof. The sequence S is constructed by appending minimised counterexamples or by refining existing elements with a pairing with another minimised counterexample. Lemma 11 guarantees that all minimised counterexamples are fully inequated and by Lemma 17, any basic pairing between fully inequated multi-clauses is also fully inequated. ■

Lemma 20 *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 8 guarantees that all minimised counterexamples are full and by Lemma 18, any basic pairing between full multi-clauses is also full. ■

Lemma 21 *Let $[s, c]$ be any pairing of the two fully inequated 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. From Lemma 1 we know that there are θ_x and θ_i for which $s \cdot \theta_x \subseteq s_x$ and $s \cdot \theta_i \subseteq s_i$. Moreover, s is fully inequated by Lemma 17, and Lemma 3 guarantees therefore that θ_x and θ_i do not unify terms in s. That is, no literals in s^p or $s^{\neq}$ are unified and the result follows. ■

Lemma 22 *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 $s_t \rightarrow b_t$ in 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 $[s_i, c_i]$ for which the two properties $rhs(s, c) \neq \emptyset$ and $size(s) \lneq size(s_i)$ hold.

We have assumed that there is some clause $s_t \rightarrow b_t \in T$ violated by $[s_x, c_x]$ and covered by $[s_i, c_i]$. Let θ'_x be the substitution showing the violation of $s_t \rightarrow b_t$ by $[s_x, c_x]$ and θ'_i be the substitution showing the fact that $s_t \rightarrow b_t$ is covered by $[s_i, c_i]$. Thus it holds that $s_t \cdot \theta'_x \subseteq s_x$, $b_t \cdot \theta'_x \in c_x$ and $s_t \cdot \theta'_i \subseteq s_i$.

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. Lemma 19 states both s_i and s_x are completely inequated. By Lemma 3, θ'_x and θ'_i do not unify terms, and hence the matching σ is 1-1, since it cannot be the case that $t \cdot \theta'_x = t' \cdot \theta'_x$ or $t \cdot \theta'_i = t' \cdot \theta'_i$, for any two distinct terms t, t' in s_t.

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 (that in this case is guaranteed to be s_x). □

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 $\texttt{[}v \cdot \theta'_x \texttt{ - } v \cdot \theta'_i \texttt{ => } lgg(v \cdot \theta'_x, v \cdot \theta'_i)\texttt{]}$. By Lemma 12, s_x is minimised and $v \cdot \theta'_x$ 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

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

The entries $\texttt{[}t_j \cdot \theta'_x \texttt{ - } t_j \cdot \theta'_i \texttt{ => } lgg(t_j \cdot \theta'_x, t_j \cdot \theta'_x)\texttt{]}$, 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. That 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. □

It remains to show that properties $rhs(s, c) \neq \emptyset$ and $size(s) \lneq size(s_i)$ are satisfied. Let θ_x and θ_i be the two substitutions defined in Lemma 1. 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. It holds that $s \cdot \theta_x \subseteq s_x$ and $s \cdot \theta_i \subseteq s_i$.

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

Proof. To see this, it suffices to show that $[s, c]$ violates $s_t \rightarrow b_t$. We have to find a substitution θ such that $s_t \cdot \theta \subseteq s$ and $b_t \cdot \theta \in c$.

Let θ be the substitution that maps all variables in s_t to their corresponding expression assigned in the extension of σ. That is, $\theta = \{v \mapsto lgg(v \cdot \theta'_x, v \cdot \theta'_i) \mid v \textit{ is a variable of } s_t\}$. Notice that θ is a variable renaming because θ'_x is and the lgg when a variable is involved is also a variable.

- $\theta \cdot \theta_x = \theta'_x$. 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 $\theta \cdot \theta_x$.

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

- $s_t \cdot \theta \subseteq s$. Let l be any literal in s_t^p 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 $lgg_{|\sigma}(s_x^p, s_i^p) = s^p \subseteq s$ since we have seen that any term in $l \cdot \theta$ appears in σ. Hence, $l \cdot \theta \in s$ and $s_t^p \cdot \theta \subseteq s$. Let $t \neq t'$ be any inequation in s_t. The set s_t is fully inequated, and therefore, t, t' appear in some non-equational literals l, l' in s_t. And therefore, $l \cdot \theta, l' \cdot \theta$ appear in s. Notice $t \cdot \theta$ and $t' \cdot \theta$ are distinct terms since θ is a variable renaming. By construction, s is fully inequated and therefore $t \cdot \theta \neq t' \cdot \theta \in s^{\neq} \subseteq s$. We have seen that both inequational literals and atoms in $s_t \cdot \theta$ appear in s and hence $s_t \cdot \theta \subseteq s$ as required.

- $b_t \cdot \theta \notin s$. 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 $b_t \cdot \theta'_x \in s_x$. This contradicts the fact that $[s_x, c_x]$ violates $s_t \to b_t$ via θ'_x.

Since $[s, c]$ is full, $T \models s_t \cdot \theta \to b_t \cdot \theta$, $s_t \cdot \theta \subseteq s$, θ is non-unifying and $b_t \cdot \theta \notin s$ we conclude that $b_t \cdot \theta \in c$. And $rhs(s, c) \neq \emptyset$ as required. □

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

Proof. By way of contradiction, suppose $size(s) \geq size(s_i)$. By Lemma 21, we know that $|s| \leq |s_i|$, therefore $size(s) \leq size(s_i)$ since the lgg never substitutes a term by one of greater weight. Thus, it can only be that $size(s) = size(s_i)$ and $|s| = |s_i|$ (if $|s| < |s_i|$, then we would also get $size(s) < size(s_i)$ 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 $s \cdot \theta_i = s_i$, and hence s_x contains a variable renaming of s_i. That is, there is a substitution $\hat{\theta}$ 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 20, the multi-clause $[s_i, c_i]$ is full and therefore $b_t \cdot \theta'_i \in c_i$. Hence, the clause $s_i \to b_t \cdot \theta'_i$ is included in H. This implies that $H \models s_i \to b_t \cdot \theta'_i$. The substitution $\hat{\theta}$ is a variable renaming and hence $H \models s_i \cdot \hat{\theta} \to b_t \cdot \underbrace{\theta'_i \cdot \hat{\theta}}_{\theta'_x}$ and since $s_i \cdot \hat{\theta} \subseteq s_x$, we obtain that $H \models s_x \to b_t \cdot \theta'_x$. But since $b_t \cdot \theta'_x \in c_x$, this contradicts the fact that $[s_x, c_x]$ is a counterexample.

□

To summarise, we have constructed a 1-1, legal and basic matching σ that induces the legal pairing $[s, c]$ that will be considered by the algorithm and for which the two properties needed for refining the multi-clause $[s_i, c_i] \in S$ hold. That is, $[s_i, c_i]$ is refined as required. ■

Corollary 23 *If a counterexample $[s_x, c_x]$ is appended to S, it is because there is no element in S violating a clause in T that is also violated by $[s_x, c_x]$.*

Proof. Were it the case, then by Lemma 22 the first element sharing some target clause would have been replaced instead of being appended. ■

Lemma 24 *Let $[s_1, c_1]$ and $[s_2, c_2]$ be two completely inequated full multi-clauses. And let $[s, c]$ be any legal pairing between them. If $[s, c]$ violates a fully inequated clause $s_t \rightarrow b_t$, then the following holds:*

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

Proof. By assumption, $s_t \rightarrow b_t$ is violated by $[s, c]$, i.e., there is a θ such that $s_t \cdot \theta \subseteq s$ and $b_t \cdot \theta \in c$. Let σ be the 1-1 legal matching inducing the pairing. By Lemma 1, there exist substitutions θ_1 and θ_2 such that $s \cdot \theta_1 \subseteq s_1$ and $s \cdot \theta_2 \subseteq s_2$.

We claim that $[s_1, c_1]$ and $[s_2, c_2]$ cover $s_t \rightarrow b_t$ via $\theta \cdot \theta_1$ and $\theta \cdot \theta_2$, respectively. Notice that $s_t \cdot \theta \subseteq s$, and therefore $s_t \cdot \theta \cdot \theta_1 \subseteq s \cdot \theta_1 \subseteq s_1$. The same holds for s_2.

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

Lemma 25 (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 T and every clause of 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 T simultaneously.*

Proof. To see that every multi-clause $[s, c] \in S$ violates at least one clause of T, it suffices to observe that by Lemma 20 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 T.

To see that no two different multi-clauses in S violate the same clause of 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 23, we know that $[s_x, c_x]$ does not violate any clause in 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 $[s_i', c_i']$. We have to prove that the lemma holds for S' as updated by the learning algorithm. Assume it does not. The only possibility is that the new element $[s_i', c_i']$ violates some clause of T, say $s_t \rightarrow b_t$ 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 24 we conclude that one of $[s_x, c_x]$ or $[s_i, c_i]$ violates $s_t \rightarrow b_t$.

Suppose $[s_i, c_i]$ violates $s_t \rightarrow b_t$. This contradicts the induction hypothesis, since both $[s_i, c_i]$ and $[s_j, c_j]$ violate $s_t \rightarrow b_t$ in T.

Suppose $[s_x, c_x]$ violates $s_t \rightarrow b_t$. If $j < i$, then $[s_x, c_x]$ should have refined $[s_j, c_j]$ instead of $[s_i, c_i]$ (Lemma 22). 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 24, all multi-clauses in position i cover $s_t \rightarrow b_t$ during the history of S. Consider the iteration in which $[s_j, c_j]$ first violated $s_t \rightarrow b_t$. This could have happened by appending the counterexample $[s_j, c_j]$, which contradicts Lemma 22 since $[s_i, c_i]$ or an ancestor of it was covering $s_t \to b_t$ but was not replaced. Or it could have happened by refining $[s_j, c_j]$ with a pairing of a counterexample violating $s_t \to b_t$. But then, by Lemma 22 again, the element in position i should have been refined, instead of refining $[s_j, c_j]$. ■

Corollary 26 *The number of elements in S is bounded by m, the number of clauses in T.*

Proof. Suppose there are more than m elements in S. Since every $[s, c]$ in S violates some clause in T, then it must be the case that two different elements in S violate the same clause of T, since there are only m clauses in T, contradicting Lemma 25. ■

Lemma 27 *Let $[s_x, c_x]$ be any minimised counterexample. Then, $|s_x| + |c_x| \le st^a$.*

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 28 *The algorithm makes $O(mst^a)$ 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 mst^a (to drop literals) plus mt (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 $O(mst^a)$. ■

Lemma 29 *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 $|c_x| \le se_t^a$ 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 30 *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 27, this is bounded by st^a. ■

Lemma 31 *The algorithm makes $O(ms^2t^ae_t^{a+1} + m^2s^2t^{2a+k})$ 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.

That is: $\underbrace{smt^a}_{\#iterations} \times \{ \underbrace{se_t^{a+1}}_{minim.} + \underbrace{m}_{|S|} \cdot \underbrace{t^k}_{\#pairings} \cdot \underbrace{st^a}_{pairing} \} = O(ms^2t^ae_t^{a+1} + m^2s^2t^{2a+k})$. ■

Theorem 32 *The algorithm exactly identifies every range restricted Horn expression making $O(mst^a)$ equivalence queries and $O(ms^2t^ae_t^{a+1} + m^2s^2t^{2a+k})$ membership queries. The running time is polynomial in the number of membership queries.*

Proof. Follows from Lemmas 28 and 31. Notice that the membership calls take most of the running time. ■

References

[AK00] M. Arias and R. Khardon. A New Algorithm for Learning Range Restricted Horn Expressions. Technical Report EDI-INF-RR-0010, Division of Informatics, University of Edinburgh, March 2000.

[Ari97] Hiroki Arimura. Learning acyclic first-order Horn sentences from entailment. In *Proceedings of the International Conference on 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.