Learning Closed Horn Expressions[1]

Marta Arias and Roni Khardon

Electrical Engineering and Computer Science, Tufts University
161 College Avenue, Medford, MA 02155, USA

E-mail: {marias,roni}@eecs.tufts.edu

The paper studies the learnability of Horn expressions within the framework of *learning from entailment*, where the goal is to exactly identify some pre-fixed and unknown expression by making queries to *membership* and *equivalence* oracles. It is shown that a class that includes both *Range Restricted Horn Expressions* (where terms in the conclusion also appear in the condition of a Horn clause) and *Constrained Horn Expressions* (where terms in the condition also appear in the conclusion of a Horn clause) is learnable. This extends previous results by showing that a larger class is learnable with better complexity bounds. A further improvement in the number of queries is obtained when considering the class of Horn expressions with inequalities on all syntactically distinct terms.

Key Words: Computational learning theory, Inductive logic programming, Horn expressions, algorithms, queries

1. INTRODUCTION

This paper considers the problem of learning an unknown first order expression T (often called *target* expression) from examples of clauses that T entails or does not entail. This type of learning framework is known as *learning from entailment*. Frazier & Pitt [6] formalised learning from entailment using equivalence queries and membership queries and showed the learnability of propositional Horn expressions. Generalising this result to the first order setting is of clear interest. Indeed, several works have been done following this line [9, 3, 20, 19, 10, 11, 2] obtaining algorithms that work for certain subsets of Horn expressions.

Learning first order Horn expressions has become a fundamental problem in *Inductive Logic Programming* [15]. Theoretical results have shown that learning from examples only is feasible for very restricted classes [4] and that, in fact, learnability becomes intractable when slightly more general classes are considered [5]. To

[1]This work has been done at the University of Edinburgh supported by EPSRC Grant GR/M21409, and at Tufts University supported by NSF Grant IIS-0099446.

tackle this problem, learners have been equipped with the ability to ask questions. It is the case that with this ability larger classes can be learned. In this paper, the questions that the learner is allowed to ask are *membership* and *equivalence* queries. While our work is purely theoretical, there are systems that are able to learn using equivalence and membership queries (MIS [23], CLINT [18], for example). Some of the techniques developed in this framework have been adapted for systems that learn from examples only [21, 12].

We present an algorithm to learn certain subsets of Horn expressions. The algorithm is related to the ones in [10, 11], which learn Range Restricted Horn expressions. The algorithms in [10, 11] and here use 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 than those in [10, 11], 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 [10, 11] uses direct products of models we use an operation based on the *lgg* (least general generalisation [17]). The use of *lgg* seems a more natural and intuitive technique to use for learning from entailment, and it has been used before, both in theoretical and applied work [3, 20, 19, 14]. The class of *Closed Horn Expressions* shown to be learnable here, includes both the class of *Range Restricted Horn Expressions*, the class of *Constrained Horn Expressions* and their union[2]. In addition, the complexity of the algorithm is better than that of the algorithm in [10, 11].

We extend our results to the class of *Fully Inequated Closed 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 closed Horn expressions. The more restricted nature of this class allows for better bounds to be derived.

The rest of the paper is organised as follows. Section 2 gives some preliminary definitions. The learning algorithm is presented in Section 3 and proved correct in Section 4. The results are extended to the fully inequated case in Section 5. Finally, Section 6 compares the results obtained in this paper with previous results and includes further discussion of the result and related work.

2. PRELIMINARIES

We consider a subset of the class of universally quantified expressions in first order logic. In the learning problem, a pre-fixed known and finite signature of the language is assumed. 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 $x_1, x_2, x_3, \ldots$ is used to construct expressions.

[2]This extends preliminary work in [2], which showed learnability of Range Restricted Horn Expressions only.

Definitions of first order languages can be found in standard texts, e.g. [13]. Here we briefly introduce the necessary constructs. A variable is a *term* of depth 0. If $t_1, \ldots, t_n$ 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.

Let X be a term, or set of terms, or atom, or set of atoms. The set $Terms(X)$ is the set of terms and subterms appearing in X.

Let P be a set of predicates together with their arities, and X a term, or set of terms, or atom, or set of atoms. The set $Atoms_P(X)$ is the set of atoms built from predicate symbols in P (of the correct arity) and terms in $Terms(X)$.

EXAMPLE 2.1. Suppose $P = \{p/2, q/1\}$ and r is a predicate of arity 1.

- $Terms(f(x, g(a))) = \{x, a, g(a), f(x, g(a))\}$
- $Atoms_P(r(f(1))) = \{p(1,1), p(1, f(1)), p(f(1), 1), p(f(1), f(1)), q(1), q(f(1))\}$

A *clause* is a disjunction of literals where all variables are 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 *definite* if it has exactly one positive literal.

A *Range Restricted Horn clause* $s \rightarrow b$ 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. That is, $Terms(b) \subseteq Terms(s)$. A *Range Restricted Horn Expression* is a conjunction of Range Restricted Horn clauses.

A *Constrained Horn clause* $s \rightarrow b$ is a definite Horn clause in which every term appearing in its antecedent also appears in its consequent, possibly as a subterm of another term. That is, $Terms(s) \subseteq Terms(b)$. A *Constrained Horn Expression* is a conjunction of Constrained 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* (also called *structure* or *model*) I includes a domain D which is a 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 the predicate 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 positive literal $p \cdot \theta$ is true in I if it appears in the extension of I. A negative literal is true in I if its negation is not.

A Horn clause $C = p_1 \wedge ... \wedge p_n \rightarrow p_{n+1}$ is true in a given interpretation I, denoted $I \models C$ if 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.

A *multi-clause* is a pair of the form $[s, c]$, where both s and c are sets of atoms 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 atoms 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 $[s_c, \{b_c\}]$.

EXAMPLE 2.2. We represent multi-clauses using set notation: e.g., the multi-clause $[\{p(x, f(a)), q(y)\}, \{r(a), r(f(a)\}]$ is interpreted as the logical expression

$$(p(x, f(a)) \wedge q(y) \rightarrow r(a)) \wedge (p(x, f(a)) \wedge q(y) \rightarrow r(f(a))).$$

The definition of the sets $Atoms_P$ and $Terms$ is extended to include clauses and multi-clauses as the input argument in the natural way. That is: $Terms(s \rightarrow b) = Terms(s \cup \{b\})$ and $Terms([s, c]) = Terms(s \cup c)$. Similarly, $Atoms_P(s \rightarrow b) = Atoms_P(s \cup \{b\})$ and $Atoms_P([s, c]) = Atoms_P(s \cup c)$.

A multi-clause $[s, c]$ is *range restricted* if $Terms(c) \subseteq Terms(s)$; it is *constrained* if $Terms(s) \subseteq Terms(c)$.

A logical expression T *implies* (or *logically entails*) a multi-clause $[s, c]$ if it implies all of its single clause components. That is, $T \models [s, c]$ if $T \models \bigwedge_{b \in c} s \rightarrow b$.

The *size* of a term is the number of occurrences of variables plus twice the number of occurrences of function symbols (including constants). The *size* of an atom is the sum of the sizes of the (top-level) terms it contains plus 1. The *size* of a set of atoms is the sum of sizes of atoms in it.

Let s_1, s_2 be two sets of atoms. 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. Equivalently, s_2 is a *instance* of s_1.

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

DEFINITION 2.1. A *derivation* of a clause $C = A \rightarrow a$ from a Horn expression T is a finite directed acyclic graph G with the following properties. Nodes in G are atoms possibly containing variables. The node a is the unique node of out-degree zero. For each node b in G, let $Pred(b)$ be the set of nodes b' in G with edges from b' to b. Then, for every node b in G, either $b \in A$ or $Pred(b) \rightarrow b$ is an instance of a clause in T. A derivation G of C from T is *minimal* if no proper subgraph of G is also a derivation of C from T. A minimal derivation G of a clause $C = A \rightarrow a$ from a Horn expression T is said to be *trivial* if all nodes b of G are contained in $A \cup \{a\}$, otherwise it is *nontrivial*.

THEOREM 2.1. *Let T be any Horn expression and C be a Horn clause which is not a tautology. If $T \models C$, then there is a minimal derivation of C from T.*

Proof. As proved by the *Subsumption Theorem for SLD-resolution* (Theorem 7.10 in [16]), there is a SLD-resolution of C from T. By induction on the depth of the SLD-resolution tree we can show how to transform any SLD-resolution into a derivation graph of C from T. Therefore, there is a derivation graph of C from T which guarantees that there is a minimal one. ∎

DEFINITION 2.2. A class $\mathcal{C}$ of Horn Expressions is *closed* if for every pair of atoms b and b', every set of atoms s and every Horn expression $T \in \mathcal{C}$, if b' is used in a minimal derivation of $s \to b$ from T, then $b' \in Atoms_P(s \to b)$.

LEMMA 2.1. *The following classes are closed: $RRHE$, the class of Range Restricted Horn Expressions, $COHE$ the class of Constrained Horn Expressions and $RRCOHE$ the class $RRHE \cup COHE$.*

Proof. For $RRHE$: if b' appears in any derivation of $T \models s \to b$, where T is a range restricted Horn expression and s is a set of atoms, then obviously, $T \models s \to b'$. T is range restricted and therefore b' is made out of terms in s only. Thus, $b' \in Atoms_P(s) \subseteq Atoms_P(s \to b)$.

For $COHE$: consider any minimal derivation of $s \to b$ from a constrained Horn expression T. If b' appears in the derivation, then, since T is constrained, b' must be made out of terms in b only. Thus, $b' \in Atoms_P(b) \subseteq Atoms_P(s \to b)$.

For $RRCOHE$ the property follows immediately since $RRCOHE$ is the disjoint union of $RRHE$ and $COHE$. ∎

Notice that any expression in $RRCOHE$ is either a range restricted Horn expression or a constrained Horn expression. This is *not* the class of expressions whose clauses are either range restricted or constrained. In the class considered here we do not allow expressions with mixed types of clauses.

DEFINITION 2.3. A multi-clause $[s, c]$ is *correct* w.r.t. a Horn expression T if $T \models [s, c]$. A multi-clause $[s, c]$ is *closed* w.r.t. a Horn expression T if for all $b \in Atoms_P(s \cup c) \setminus s$ such that $T \models s \to b$, $b \in c$. A multi-clause $[s, c]$ is *full* if it is correct and closed.

2.1. Most General Unifier

Let Σ be a finite set of expressions (here by "expressions" we mean terms or atoms). 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.*

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.

The *disagreement set* of a finite set of expressions 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 2.3. $\Sigma = \{p(x, \underline{y}, v), p(x, \underline{f(g(a))}, x), p(x, \underline{f(z)}, f(a))\}$. Its disagreement set is $\{y, f(g(a)), f(z)\}$.

ALGORITHM 1 (THE UNIFICATION ALGORITHM).

1. Let Σ be the set of expressions to be unified.
2. Set k to 0 and σ_0 to $\emptyset$, the empty substitution.
3. **Repeat** until $\Sigma \cdot \sigma_k$ is a singleton
4.   Let D_k be the disagreement set for $\Sigma \cdot \sigma_k$.
5.   **If** there exist x, t in D_k s. t. x is a variable not occurring in t
6.   **Then** set $\sigma_{k+1} = \sigma_k \cdot \{x \mapsto t\}$.
7.   **Else** report that Σ is not unifiable and stop.
8. **Return** σ_k.

THEOREM 2.2 (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. See [13]. ■

2.2. Least General Generalisation

The algorithm proposed uses the *least general generalisation* or *lgg* operation [17]. This operation computes a generalisation of two sets of literals. It works as follows.

The *lgg* of two terms $f(s_1, ..., s_n)$ and $g(t_1, ..., t_m)$ is defined as the term

$$f(lgg(s_1, t_1), ..., lgg(s_n, t_n))$$

if $f = g$ and $n = m$. Otherwise, it is a new variable x, where x stands for the *lgg* of that pair of terms throughout the computation of the *lgg*. This information is kept in what we call the *lgg* table.

The *lgg* of two *compatible* atoms $p(s_1, ..., s_n)$ and $p(t_1, ..., t_n)$ is the atom

$$p(lgg(s_1, t_1), ..., lgg(s_n, t_n)).$$

The *lgg* is only defined for compatible atoms, that is, atoms with the same predicate symbol and arity.

The *lgg* of two *compatible* positive literals l_1 and l_2 is the *lgg* of the underlying atoms. The *lgg* of two *compatible* negative literals l_1 and l_2 is the negation of the *lgg* of the underlying atoms. Two literals are compatible if they share predicate symbol, arity and sign.

The lgg of two sets of literals s_1 and s_2 is the set

$$\{lgg(l_1, l_2) \mid (l_1, l_2) \text{ are two compatible literals of } s_1 \text{ and } s_2\}.$$

It is important to note that all lggs share the same table.

EXAMPLE 2.4. Let $s_1 = \{p(a, f(b)), p(g(a, x), c), q(a)\}$.
Let $s_2 = \{p(z, f(2)), q(z)\}$.
Their lgg is $lgg(s_1, s_2) = \{p(X, f(Y)), p(Z, V), q(X)\}$.
The lgg table produced during the computation of $lgg(s_1, s_2)$ is

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

2.3. The Learning Model

We consider the model of *exact learning from entailment* [6]. 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.

3. THE ALGORITHM

Before presenting the algorithm, we define some operations. Suppose that the class $\mathcal{C}$ is closed. Suppose that $H, T \in \mathcal{C}$. Then we define:

- $TClosure_T([s, c]) = [s, \{b \in Atoms_P(s \cup c) \setminus s \mid T \models s \to b\}]$
- $HClosure_H([s, c]) = [\{b \in Atoms_P(s \cup c) \mid H \models s \to b\}, c]$
- $rhs_T(s, c) = \{b \in c \mid T \models s \to b\}$

The algorithm computes these operations for the case when T is the target expression and H is a hypothesis. In practice, we do not know what the target expression T is, but we can use the $EntMQ$ oracle to compute $TClosure_T$ and rhs_T. Since T always refers to the target expression, we omit the "$_T$" subscript and write:

- $TClosure([s, c]) = [s, \{b \in Atoms_P(s \cup c) \setminus s \mid EntMQ(s \to b) = Yes\}]$
- $rhs(s, c) = \{b \in c \mid EntMQ(s \to b) = Yes\}$

Notice that, in general, the computation of $HClosure$ might not be feasible. However, in our case, we will show that this can be done with a polynomial number of subsumption tests by forward chaining. This is due to the fact that we only check for atoms in the polynomially bounded set $Atoms_P(s \cup c)$ as potential consequents. We will incrementally construct the set of consequents ($CONS$ in the algorithm), starting with the antecedent s. The algorithm is as follows:

ALGORITHM 2 (THE $HClosure(s, c)$ PROCEDURE).

1. $CONS = s$.
2. **Repeat** until no more atoms are added to $CONS$
3.   **For** every atom b in $Atoms_P(s \cup c) \setminus CONS$ **do**
4.   **If** clause $CONS \rightarrow b$ is subsumed by a clause $C \in H$
5.   **Then** Set $CONS = CONS \cup \{b\}$.
6. **Return** $[CONS, c]$

LEMMA 3.1. *Algorithm 2 computes the set $HClosure(s, c)$.*

Proof. Take any atom $b \in HClosure(s, c)$. By Theorem 2.1, there is a derivation of $s \rightarrow b$ from H. The previous algorithm searches through all possible closed derivations, therefore it will eventually reach the node b in the corresponding derivation, and b will be included in the set $CONS$. Soundness of forward chaining guarantees that atoms not in $HClosure(s, c)$ are never added to the set $CONS$. ■

We finally present our learning algorithm.

ALGORITHM 3 (THE LEARNING ALGORITHM).

1. Set S to be the empty sequence and H to be the empty hypothesis.
2. **Repeat** until $EntEQ(H)$ returns "Yes":
3.   Minimise the counterexample x - use calls to $EntMQ$
    Let $[s_x, c_x]$ be the minimised counterexample produced.
4.   Find the first $[s_i, c_i] \in S$ such that there is a basic pairing $[s, c]$ of
    $[s_i, c_i]$ and $[s_x, c_x]$ satisfying:
        (i) $rhs(s, c) \neq \emptyset$ and
        (ii) $size(s) \lneq size(s_i)$ or ($size(s) = size(s_i)$ and $size(c) \lneq size(c_i)$)
5.   **If** such an $[s_i, c_i]$ is found
6.   **Then** replace it by the multi-clause $[s, rhs(s, c)]$
7.   **Else** append $[s_x, c_x]$ to S
8.   Set H to $\bigwedge_{[s,c] \in S} \{s \rightarrow b \mid b \in c\}$
9. **Return** H

The algorithm follows pretty much the structure of the algorithm in [6] for the propositional case. It keeps a sequence S of representative multi-clauses. The hypothesis H is generated from this sequence, and the main task of the algorithm is to *refine* the counterexamples in S in order to get a more accurate hypothesis in each iteration of the main loop (line 2) until hypothesis and target expression coincide.

There are two basic operations on counterexamples that need to be explained in detail. These are *minimisation* (line 3), that takes a counterexample as given by the equivalence oracle and produces a positive, full counterexample; and *pairing* (line 4), that takes two counterexamples and generates a series of candidate counterexamples. The counterexamples obtained by combination of previous ones (by *pairing* them) are the candidates to refine the sequence S.

3.1. Minimising the counterexample

The minimisation procedure has to transform a counterexample clause $A \to a$ as generated by the equivalence query oracle into a multi-clause counterexample $[s_x, c_x]$ ready to be handled by the learning algorithm.

ALGORITHM 4 (THE MINIMISATION PROCEDURE).

1. Let $A \to a$ be the counterexample obtained by the $EntEQ$ oracle.
2. Set $[s_x, c_x]$ to $TClosure(HClosure([A, \{a\}]))$.
3. **For** every functional term t in $s_x \cup c_x$, in decreasing order of size **do**
4. Let $[s'_x, c'_x]$ be the multi-clause obtained from $[s_x, c_x]$ after substituting all occurrences of the term t by a new variable x_t
5. **If** $rhs(s'_x, c'_x) \neq \emptyset$ **then** set $[s_x, c_x]$ to $[s'_x, rhs(s'_x, c'_x)]$
6. **For** every term t in $s_x \cup c_x$, in increasing order of size **do**
7. Let $[s'_x, c'_x]$ be the multi-clause obtained after removing from $[s_x, c_x]$ all those atoms containing t
8. **If** $rhs(s'_x, c'_x) \neq \emptyset$ **then** set $[s_x, c_x]$ to $[s'_x, rhs(s'_x, c'_x)]$
9. **Return** $[s_x, c_x]$

EXAMPLE 3.1. This example illustrates the behaviour of the minimisation procedure. Parentheses are omitted; function f is unary. T consists of the single clause $p(a, fx) \to q(x)$. We start with the counterexample $[p(a, f1), q(2), r(1) \to q(1)]$ as obtained after step 2 of the minimisation procedure. In the third column of the table, correct atoms in the consequent appear with a box around them. If no atom is correct, the multi-clause is not positive and the counterexample is not updated.

$[s_x, c_x]$		*After generalising term*
$[p(a, f1), q(2), r(1) \to q(1)]$	$f1 \mapsto X$	$[p(a, X), q(2), r(1) \to q(1)]$
$[p(a, f1), q(2), r(1) \to q(1)]$	$1 \mapsto X$	$[p(a, fX), q(2), r(X) \to \boxed{q(X)}]$
$[p(a, fX), q(2), r(X) \to q(X)]$	$2 \mapsto Y$	$[p(a, fX), q(Y), r(X) \to \boxed{q(X)}]$
$[p(a, fX), q(Y), r(X) \to q(X)]$	$a \mapsto Z$	$[p(Z, fX), q(Y), r(X) \to q(X)]$
$[s_x, c_x]$		*After dropping term*
$[p(a, fX), q(Y), r(X) \to q(X)]$	X	$[q(Y) \to]$
$[p(a, fX), q(Y), r(X) \to q(X)]$	Y	$[p(a, fX), r(X) \to \boxed{q(X)}]$
$[p(a, fX), r(X) \to q(X)]$	a	$[r(X) \to q(X)]$
$[p(a, fX), r(X) \to q(X)]$	fX	$[r(X) \to q(X)]$
$[p(a, fX), r(X) \to q(X)]$		

Notice that the minimised counterexample is very similar to the target clause. In fact, it is the case that every minimised counterexample contains a syntactic variant of one of the target clauses (Lemma 4.10). However, it may still contain extra atoms that the minimisation procedure is unable to get rid of – like $r(X)$ in Example 3.1 – these will have to disappear in some other way: *pairing*.

3.2. Pairings

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

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

3.2.1. Matchings

A *matching* is a set whose elements are pairs of terms $t_x - t_i$, where t_x and t_i are terms in $s_x \cup c_x$ and $s_i \cup c_i$, respectively. Usually, we denote a matching by the Greek letter σ. A matching σ should include all the terms in one of $s_x \cup c_x$ or $s_i \cup c_i$, more formally: $|\sigma| = min(|Terms(s_x \cup c_x)|, |Terms(s_i \cup c_i)|)$. 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.

EXAMPLE 3.2. Let $[s_x, c_x]$ be $[\{p(a,b)\}, \{q(a)\}]$ with terms $\{a, b\}$. Let $[s_i, c_i]$ be $[\{p(f(1),2)\}, \{q(f(1))\}]$ with terms $\{1, 2, f(1)\}$. The 6 possible 1-1 matchings are:

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

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

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

EXAMPLE 3.3. Parentheses are omitted as functions f and g are unary. Let σ_1 be $\{a-c, fa-b, ffa-fb, gffa-gffc\}$ and $\sigma_2 = \{a-c, fa-b, ffa-fb\}$. The matching σ_1 is not legal, since the term fX is not present in its extension column and it is a subterm of $gffX$, which is present. The matching σ_2 is legal.

Extended σ_1	*Extended* σ_2
`[a - c => X]`	`[a - c => X]`
`[fa - b => Y]`	`[fa - b => Y]`
`[ffa - fb => fY]`	`[ffa - fb => fY]`
`[gffa - gffc => gffX]`	

Our algorithm considers yet a more restricted type of matching. A *basic matching* σ is a 1-1, legal matching between two multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$. This operation is asymmetric and the order in which the arguments is given is relevant. It is only defined if the number of distinct terms in $[s_x, c_x]$ (first argument) is smaller or equal than the number of distinct terms in $[s_i, c_i]$ (second argument). It restricts how the functional structure of the terms is matched. More formally, if entry $f(t_1, ..., t_n) - t \in \sigma$, then $t = f(r_1, ..., r_n)$ and $t_i - r_i \in \sigma$ for all $i = 1, ..., n$. As we show below, a basic matching maps all variables in $[s_x, c_x]$ to terms in $[s_i, c_i]$ and then adds the remaining entries following the functional structure of the terms in $[s_x, c_x]$. Therefore an entry `[x - f(y)]` might be included in a basic pairing but an entry `[f(y) - x]` cannot (terms on the left belong to $[s_x, c_x]$, terms on the right to $[s_i, c_i]$).

The following procedure shows how to construct basic matchings between multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$.

Algorithm 5 (How to Construct Basic Matchings).

1. Match every *variable* in $s_x \cup c_x$ to a different *term* in $s_i \cup c_i$.
   Every possibility will potentially yield to a basic matching
   between $[s_x, c_x]$ and $[s_i, c_i]$
2. Complete all potential basic matchings by adding the functional
   terms in $s_x \cup c_x$ to the basic matchings as follows:
3. **For** every potential basic matching created in step 1 **do**
4.   Consider all functional terms in $s_x \cup c_x$ in an upwards fashion,
       beginning with simpler terms:
5.   **For** every term $f(t_1, ..., t_n)$ in $s_x \cup c_x$ such that all
         $[t_i - r_i]$ (with $i = 1, ..., n$) appear in the basic
         matching already **do**
6.   Add a new entry $[f(t_1, ..., t_n) - f(r_1, ..., r_n)]$
7.   **If** $f(r_1, ..., r_n)$ does not appear in $s_i \cup c_i$ or the term
           $f(r_1, ..., r_n)$ has been used already
8.   **Then** discard the matching

Example 3.4. Let $s_x \cup c_x$ contain the terms $\{a, x, fx\}$ and $s_i \cup c_i$ the terms $\{a, 1, 2, f1\}$. No parentheses for functions are written. The algorithm starts by matching variables in $s_x \cup c_x$ to terms in $s_i \cup c_i$. Then, it matches functional terms in $s_x \cup c_x$ using the constraints described in the procedure above. This computation is described in the table below.

Terms	*Matching 1*	*Matching 2*	*Matching 3*	*Matching 4*
x	`[x - a]`	`[x - 1]`	`[x - 2]`	`[x - f1]`
a	NO! `[a - a]`	`[a - a]`	`[a - a]`	`[a - a]`
fx	DISCARDED	`[fx - f1]`	NO! `[fx - f2]`	NO! `[fx - ff1]`
	DISCARDED	OK	DISCARDED	DISCARDED

The table is interpreted as follows. In the first column we have the terms in $s_x \cup c_x$ as how they would be considered by our algorithm. In the columns thereafter, we have all potential matchings. The last row indicates which of the matchings has been discarded. The entries on top of the "OK" matchings contain the matching's pairs.

Notice that we have only 1 basic matching between the set of terms $\{a, x, fx\}$ and $\{a, 1, 2, f1\}$. Compare this with the 24 different 1-1 matchings that would be considered by previous algorithms. This difference grows with the complexity of the functional structure in the examples.

LEMMA 3.2. *The procedure described above finds all basic matchings between the two input multi-clauses and only basic matchings are produced.*

Proof. First, we will show that every matching constructed by the procedure is basic. It is 1-1 because after step 1 the matchings are 1-1, and the new pairs added in step 2 are checked not to be included in the matchings already. It is legal because only terms which have all of its subterms included in the matching are added. It is basic because functional structure is respected when adding a new pair.

Secondly, we will show that every basic matching will be found by the procedure. First notice that matchings including the combination of a pair [functional term in $s_x \cup c_x$ - variable in $s_i \cup c_i$] is not permitted, since subterms of the functional term in s_x have to be included in the matching and they would not have any possible legal term to be matched to because a variable has no subterms. Therefore, the only possibility involving variables is [variable in s_x - term in s_i]. All these are found in step 1 of the procedure and appropriately completed in step 2. ■

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 the restrictions imposed. By doing so we avoid testing too many pairings and hence avoid making unnecessary calls to the oracles. One of the restrictions has already been mentioned, it consists in considering basic pairings only, as opposed to considering every possible matching. This reduces the t^t possible distinct matchings to only t^k distinct *basic* pairings. Notice that there are a maximum of t^k basic matchings between $[s_x, c_x]$ with k variables and $[s_i, c_i]$ with t terms, since we only combine *variables* of s_x with *terms* in s_i. The other restriction on the candidate matching consists in the fact that every one of its entries must appear in the original *lgg* table, as we will see in the next section.

3.2.2. Pairings

Pairing is an operation that takes two multi-clauses and a matching between its terms and produces another multi-clause. We say that the pairing is *induced* by the matching it is fed as input. A *legal pairing* is a pairing for which the inducing matching is legal; a *basic pairing* is one for which the inducing matching is basic.

The antecedent s of the pairing is computed as the lgg of s_x and s_i restricted to the matching σ inducing it; we denote this by $lgg_{|\sigma}(s_x, s_i)$. An atom is included in the pairing only if all of its top-level terms appear as entries in the extended matching. This restriction is quite strong in the sense that, for example, if an atom $p(f(x))$ appears in both s_x and s_i then their lgg $p(f(x))$ will not be included unless the entry `[f(x) - f(x) => f(x)]` appears in the matching. In case `[x - x => x]` appears but `[f(x) - f(x) => f(x)]` does not, the atom $p(f(x))$ is ignored. We only consider matchings that are subsets of the *lgg* table.

The consequent c of the pairing is computed as the union of the sets $lgg_{|_\sigma}(s_x, c_i)$, $lgg_{|_\sigma}(c_x, s_i)$ and $lgg_{|_\sigma}(c_x, c_i)$. Note that in the consequent all the possible *lgg*s of pairs among $\{s_x, c_x\}$ and $\{s_i, c_i\}$ are included except $lgg_{|_\sigma}(s_x, s_i)$, which constitutes the antecedent.

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. The paring between $[s_x, c_x]$ and $[s_i, c_i]$ induced by σ is computed as follows:

Algorithm 6 (The Pairing Procedure).

1. Set s to $lgg_{|_\sigma}(s_x, s_i)$
2. Set c to $lgg_{|_\sigma}(s_x, c_i) \cup lgg_{|_\sigma}(c_x, s_i) \cup lgg_{|_\sigma}(c_x, c_i)$
3. Return $[s, c]$

Example 3.5. The table below describes two examples. Both examples have the same terms as in Example 3.4, so there is only one basic matching. Ex. 3.5.1 shows how to compute a pairing. Ex. 3.5.2 shows that a basic matching may be rejected if it does not agree with the *lgg* table (entries `[x - 1 => X]` and `[fx - f1 => fX]` do not appear in the *lgg* table).

	Example 3.5.1	*Example 3.5.2*
s_x	$\{p(a, fx)\}$	$\{p(a, fx)\}$
s_i	$\{p(a, f1), p(a, 2)\}$	$\{q(a, f1), p(a, 2)\}$
$lgg(s_x, s_i)$	$\{p(a, fX), p(a, Y)\}$	$\{p(a, Y)\}$
lgg table	`[a - a => a]`	`[a - a => a]`
	`[x - 1 => X]`	`[fx - 2 => Y]`
	`[fx - f1 => fX]`	
	`[fx - 2 => Y]`	
basic σ	`[a-a=>a]`	`[a-a=>a]`
	`[x-1=>X]`	`[x-1=>X]`
	`[fx-f1=>fX]`	`[fx-f1=>fX]`
$lgg_{|_\sigma}(s_x, s_i)$	$\{p(a, fX)\}$	PAIRING REJECTED

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

A note for potential implementations. In practice, when trying to construct basic pairings between s_x and s_i it is better to consider as entries for the matching those entries appearing in the lgg table only. That is, when combining multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$, one would first compute the $lgg(s_x, s_i)$ and record the lgg table. The next step would be to construct basic pairings using the entries in the lgg table. Instead of considering any pair between terms of s_x and s_i, the choice would be restricted to those pairs of terms present in the lgg table. The advantage of this method is that subsets of the lgg table that constitute a basic matching are systematically constructed. This implies that there is no need to check whether a given basic matching agrees with the lgg table and only subsets of the lgg table are generated. This consideration is not reflected in the bounds for the worst case analysis. However, it should constitute an important speedup in practice.

4. PROOF OF CORRECTNESS

Before going into the details of the proof of correctness, we describe the transformation $U(T)$ performed on a target expression T. It extends the transformation described in [10] (where expressions were function-free) and it serves analogous purposes.

4.1. Transforming the target expression

This transformation is never computed by the learning algorithm; it is only used in the analysis of the proof of correctness. The transformation introduces new clauses and adds some inequalities to every clause's antecedent. This avoids unification of terms in the transformed clauses. Related work in [22] also uses inequalities in clauses, although the learning algorithm and approach are completely different.

The idea is to create from every clause C in T the set of clauses $U(C)$. Every clause in $U(C)$ corresponds to the original clause C with its terms unified in a unique way, different from every other clause in $U(C)$. Every possible unification of terms of C are covered by one of the clauses in $U(C)$. The clauses in $U(C)$ will only be satisfied if the terms are unified in exactly that way.

ALGORITHM 7 (THE TRANSFORMATION ALGORITHM).

Set $U(T)$ to be the empty expression
For every clause $C = s_c \rightarrow b_c$ in T **do**
    For every partition of $Terms(C)$ $\pi = \{\pi_1, \pi_2, ..., \pi_l\}$ **do**
        Let A_π be the set of atoms $\{A(t_1, ..., t_l) \mid \forall i : 1 \leq i \leq l : t_i \in \pi_i\}$
        Let σ_π be an mgu of A_π.
        If no mgu exists **or** there are $\pi_i \neq \pi_j$ s.t. $\pi_i \cdot \sigma_\pi = \pi_j \cdot \sigma_\pi$
        Then discard the partition
        Else
            Set $U_\pi(C) = ineq(C \cdot \sigma), s_c \cdot \sigma \rightarrow b_c \cdot \sigma$
            Set $U(T) = U(T) \wedge U_\pi(C)$
Return $U(T)$.

We construct $U(T)$ from T by considering every clause separately. For a clause C in T we generate a set of clauses $U(C)$. To do that, we consider all partitions of the set of terms in $Terms(C)$; 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

$U(C) = \bigwedge_{\pi \in ValidPartitions(Terms(C))} U_\pi(C)$. The set $ValidPartitions(Terms(C))$ captures those partitions for which a simultaneous unifier of all of its classes exists and partitions whose representatives are all different. The use of A_π provides the simultaneous mgu; uniqueness of representatives is tested on line 6 in the transformation algorithm. We call a *representative* of a class π_i the only element in $\pi_i \cdot \sigma_\pi$, where σ_π is a mgu for the set A_π as described in the algorithm above.

EXAMPLE 4.1. Let C be $p(f(x), f(y), g(z)) \rightarrow q(x, y, z)$. The terms appearing in C are $\{x, y, z, f(x), f(y), g(z)\}$. We consider some possible partitions:

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

$$A_\pi = \begin{cases} A(x, z, f(x), g(z)) \\ A(x, z, f(y), g(z)) \\ A(y, z, f(x), g(z)) \\ A(y, z, f(y), g(z)) \end{cases}$$

A mgu for A_π is $\sigma_\pi = \{y \mapsto x\}$. Therefore,

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

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

$$A_{\pi'} = \begin{cases} A(x, f(x), f(y)) \\ A(x, g(z), f(y)) \\ A(y, f(x), f(y)) \\ A(y, g(z), f(y)) \\ A(z, f(x), f(y)) \\ A(z, g(z), f(y)) \end{cases}$$

There is no mgu for the set $A_{\pi'}$, therefore this partition does not contribute to the transformation $U(C)$.

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

$$A_{\pi''} = \begin{cases} A(x, z, f(x), f(y), g(z)) \\ A(y, z, f(x), f(y), g(z)) \end{cases}$$

A mgu for $A_{\pi''}$ is $\sigma_{\pi''} = \{y \mapsto x\}$. However, this partition is discarded because the representatives for classes π_3 and π_4 coincide: $\pi_3 \cdot \sigma_\pi = \{f(x)\} = \pi_4 \cdot \sigma_\pi$. Notice that the partition π covers the case when the terms $f(x)$ and $f(y)$ are unified into the same term, so adding this clause would introduce repeated clauses in the transformation.

We write the fully inequated clause "$ineq(s_t \rightarrow b_t), s_t \rightarrow b_t$" as "$\neq (s_t \rightarrow b_t)$". The following facts hold for T and its transformation $U(T)$.

LEMMA 4.1. *If an expression T has m clauses, then the number of clauses in its transformation $U(T)$ is at most mt^k, where t (k, resp.) is the maximum number of different terms (variables, resp.) in any clause in T.*

Proof. It suffices to see that any clause C produces at most t^k clauses in $U(C)$. We will show that if π and π' are two partitions that are not discarded by the transformation algorithm and $\sigma_\pi = \sigma_{\pi'}$, then $\pi = \pi'$. Suppose, then, that π and π' are two successful partitions such that $\sigma_\pi = \sigma_{\pi'}$. Let t and t' be two distinct terms of C in the same class in π. Notice that since σ_π is a unifier for A_π, t and t' have the same representative. Therefore, these two terms have to fall into the same class in π' (otherwise π' would be rejected). Since the same argument also holds in the opposite direction (i.e. from π' to π) we conclude that for all terms t, t' of C, t and t' are placed in the same class in π if and only they are placed in the same class in π'. Hence, $\pi = \pi'$. Finally, the bound follows since there are at most t^k substitutions mapping the at most k variables into the at most t terms. ■

LEMMA 4.2. *$T \models U(T)$.*

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

COROLLARY 4.1. *If $U(T) \models C$, then $T \models C$. Also, if $U(T) \models [s, c]$, then $T \models [s, c]$.*

However, the inverse implication $U(T) \models T$ of Lemma 4.2 does not hold. To see this, consider the following example.

EXAMPLE 4.2. We present an expression T, its transformation $U(T)$ and an interpretation I such that $I \models U(T)$ but $I \not\models T$. The expression T is $\{p(a, f(a)) \rightarrow q(a)\}$ and its transformation $U(T) = \{(a \neq f(a)), p(a, f(a)) \rightarrow q(a)\}$. The interpretation I has domain $D_I = \{1\}$; the only constant $a = 1$; the only function $f(1) = 1$ and the extension $ext(I) = \{p(1, 1)\}$.

$I \not\models T$ 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.

4.2. Some definitions

During the analysis, s will stand for the cardinality of P, the set of predicate symbols in the language; a for the maximal arity of the predicates in P; k for the maximum number of distinct variables in a clause of T; t for the maximum number of distinct terms in a clause of T; e_t for the maximum number of distinct terms in a counterexample; m for the number of clauses of the target expression T; m' for the number of clauses of the transformation of the target expression $U(T)$.

DEFINITION 4.1. A multi-clause $[s, c]$ *covers* a clause $\neq (s_t \rightarrow b_t)$ if there is a mapping θ from variables in $s_t \cup \{b_t\}$ into terms in $Terms(s \cup c)$ such that $s_t \cdot \theta \subseteq s$, $ineq(s_t \cup \{b_t\}) \cdot \theta \subseteq ineq(s \cup c)$ and $b_t \cdot \theta \in Atoms_P(s \cup c)$. Equivalently, we say that $\neq (s_t \rightarrow b_t)$ *is covered* by $[s, c]$.

The condition $ineq(s_t \cup \{b_t\}) \cdot \theta \subseteq ineq(s \cup c)$ establishes that the substitution θ is non-unifying, i.e., it does not unify terms in $s_t \rightarrow b_t$ in the sense that two distinct terms in $s_t \rightarrow b_t$ will remain distinct after applying the substitution θ.

DEFINITION 4.2. A multi-clause $[s, c]$ *captures* a clause $\neq (s_t \rightarrow b_t)$ if there is a mapping θ from variables in s_t into terms in s such that $\neq (s_t \rightarrow b_t)$ is covered by $[s, c]$ via θ and $b_t \cdot \theta \in c$. Equivalently, we say that $\neq (s_t \rightarrow b_t)$ *is captured by* $[s, c]$.

4.3. Brief description of the proof of correctness

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 captures some clause of $U(T)$ but no two distinct elements of S capture the same clause of $U(T)$ (Lemma 4.17). 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 captures 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 4.7) and that any full multi-clause must capture some clause in $U(T)$ (Corollary 4.2).

To see that no two distinct elements of S capture the same clause of $U(T)$, two important properties are established in the proof. Lemma 4.16 shows that if a counterexample $[s_x, c_x]$ captures some clause of $U(T)$ which is covered by some $[s_i, c_i]$ then the algorithm will replace $[s_i, c_i]$ with one of their basic pairings. Lemma 4.15 shows that a basic pairing cannot capture a clause not captured by either of the original clauses. These properties are used in Lemma 4.17 to prove uniqueness of captured clauses.

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.

4.4. Properties of substitutions

Our proof of correctness relies partly on some basic properties of substitutions. Here we list all of the properties used. However, they might not be explicitly referenced in the proof.

Let θ (and subscripted variations of it) be substitutions, S and s two sets of atoms and θ_N a non-unifying substitution (w.r.t. $s \rightarrow b$). With a non-unifying substitution (w.r.t. some expression Σ) we mean that if t, t' are two distinct terms in Σ, then the terms $t \cdot \theta_N$ and $t' \cdot \theta_N$ are distinct terms as well.

LEMMA 4.3.

1. *If* $b \in s$, *then* $b \cdot \theta \in s \cdot \theta$.
2. *If* $b \notin s$, *then* $b \cdot \theta_N \notin s \cdot \theta_N$.

3. *If* $b \in S \setminus s$, *then* $b \cdot \theta \in S \cdot \theta \setminus s \cdot \theta$ *unless* $b \cdot \theta \in s \cdot \theta$.
4. *If* $b \in S \setminus s$, *then* $b \cdot \theta_N \in S \cdot \theta_N \setminus s \cdot \theta_N$.
5. *If* $\theta = (\theta_1 \cdot \theta_2)$ *and* $t \cdot \theta \neq t' \cdot \theta$, *then* $t \cdot \theta_1 \neq t' \cdot \theta_1$.
6. *If* $T \models s \to b$, *then* $T \models s \cdot \theta \to b \cdot \theta$.

Proof. We prove some of the properties. For Property 2., suppose that $b \notin s$. The substitution θ_N is non-unifying for s and b, therefore, distinct terms in b remain distinct after applying θ_N. Therefore we can reverse θ_N, and we conclude that if $b \cdot \theta_N \in s \cdot \theta_N$ then $b \in s$. Hence, $b \cdot \theta_N \notin s \cdot \theta_N$. Property 3. is straightforward, and with Property 2., it implies that Property 4. holds. For Property 5., notice that if $t \cdot \theta_1 = t' \cdot \theta_1$, then θ cannot distinguish the terms t and t'. ■

4.5. Properties of full multi-clauses

The next two lemmas use properties of derivation graphs to improve over the model construction argument given in a preliminary version of the paper [2] which only holds for Range Restricted expressions.

LEMMA 4.4. *If* $[s, c]$ *is subsumed by a clause* C, *then* $[s, c]$ *captures some clause in* $U(C)$.

Proof. By assumption, $C = s_c \to b_c$ subsumes $[s, c]$. That is, there is a substitution θ such that $s_c \cdot \theta \subseteq s$ and $b_c \cdot \theta \in c$. To see which clause in $U(C)$ is captured by $[s, c]$ consider the partition π defined by the way terms in $s_c \cup \{b_c\}$ are unified by the substitution θ. More precisely, two distinct terms t, t' appearing in $s_c \cup \{b_c\}$ fall into the same class of π if and only if $t \cdot \theta = t' \cdot \theta$. The proof proceeds by arguing that the clause $U_\pi(C)$ appears in $U(C)$ and that $[s, c]$ captures $U_\pi(C)$.

We observe that θ is a unifier for $A_\pi = \{A(t_1, ..., t_l) \mid \forall i : 1 \leq i \leq l : t_i \in \pi_i\}$. Thus, a *mgu* σ_π exists. Therefore, $\theta = \sigma_\pi \cdot \hat{\theta}$ for some substitution $\hat{\theta}$. The transformation procedure rejects a partition π when some of the following conditions hold. Either A_π is not unifiable (however, we have seen it is) or the representatives of two distinct classes are equal. The second condition does not hold because $\pi_i \cdot \sigma_\pi = \pi_j \cdot \sigma_\pi$ (with $i \neq j$) implies $\pi_i \cdot \theta = \pi_j \cdot \theta$, which is not true by the way π was constructed.

Finally, we show that $[s, c]$ captures $U_\pi(C) = (\neq (s_t \to b_t))$ via $\hat{\theta}$. Notice that $s_c \cdot \sigma_\pi = s_t$ and $b_c \cdot \sigma_\pi = b_t$. We need to check (1) $s_t \cdot \hat{\theta} \subseteq s$, (2) $ineq(s_t \cup \{b_t\}) \cdot \hat{\theta} \subseteq ineq(s \cup c)$ and (3) $b_t \cdot \hat{\theta} \in c$. Condition (1) is easy: $s_t \cdot \hat{\theta} = s_c \cdot \sigma_\pi \cdot \hat{\theta} = s_c \cdot \theta \subseteq s$ by hypothesis. For (2), let t, t' be two different terms in $s_t \cup \{b_t\}$. It is sufficient to check that $t \cdot \hat{\theta}, t' \cdot \hat{\theta}$ are also different terms (i.e., $\hat{\theta}$ does not unify them). Let t_c, t'_c be the two terms in C such that $t_c \cdot \sigma_\pi = t$ and $t'_c \cdot \sigma_\pi = t'$. Since $t \neq t'$, it follows that t_c, t'_c belong to a different class of π (otherwise σ_π would have unified them). Therefore, by construction, $t_c \cdot \theta \neq t'_c \cdot \theta$. Equivalently, $t_c \cdot \sigma_\pi \cdot \hat{\theta} \neq t'_c \cdot \sigma_\pi \cdot \hat{\theta}$ and hence $t \cdot \hat{\theta} \neq t' \cdot \hat{\theta}$ as required. Condition (3) is like (1). ■

LEMMA 4.5. *If a multi-clause* $[s, c]$ *is correct for some closed target expression* T, $c \neq \emptyset$ *and it is closed w.r.t.* T, *then some clause of* $U(T)$ *must be captured by* $[s, c]$.

Proof. Fix any $b \in c$. Clearly, $T \models s \to b$ (since we have assumed $[s,c]$ correct). Consider a minimal derivation graph G of $s \to b$ from T. By Theorem 2.1 such a graph exists. We start with atom b in the graph and consider $Pred(b)$, the set of atoms that have an edge ending at b. If any of the atoms b' in $Pred(b)$ does not appear in s, then we take b' as our next b. Notice that $b' \notin s$ implies $b' \in c$, since $[s,c]$ is closed. We iterate until we find an atom $b' \in c$ such that $Pred(b') \subseteq s$. By construction of derivation graphs, the clause $Pred(b') \to b'$ must be an instance of some clause C in T. Equivalently, C subsumes $Pred(b') \to b'$ and therefore it also subsumes $[s,c]$ because $Pred(b') \subseteq s$ and $b' \in c$. Using Lemma 4.4 we conclude that some clause in $U(T)$ is captured by $[s,c]$. ■

COROLLARY 4.2. *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 captured by $[s,c]$.*

LEMMA 4.6. *If $[s,c]$ captures some clause of $U(T)$, then $rhs(s,c) \neq \emptyset$.*

Proof. The fact that $[s,c]$ captures some clause of $U(T)$ implies that there is a clause $s_c \to b_c$ in T and a substitution θ such that $s_c \cdot \theta \subseteq s$ and $b_c \cdot \theta \in c$. Clearly, $T \models s_c \to b_c \models s_c \cdot \theta \to b_c \cdot \theta$ and hence the atom $b_c \cdot \theta \in c$ survives the rhs operation. ■

COROLLARY 4.3. *If $[s,c]$ is a full multi-clause w.r.t. T and $c \neq \emptyset$, then $rhs(s,c) \neq \emptyset$.*

4.6. Properties of minimised multi-clauses

This section includes properties of minimised multi-clauses as produced by the minimisation procedure. Throughout the proof, we will refer to the minimised multi-clause as $[s_x, c_x]$.

Lemma 4.10 shows that every minimised counterexample contains a syntactic variant of some clause in $U(T)$, excluding inequalities. This is an important property and it is responsible for one of the main improvements in the bounds.

DEFINITION 4.3. 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 atoms $b \in c$, $H \not\models s \to b$.

LEMMA 4.7. *Every minimised $[s_x, c_x]$ is full w.r.t. the target expression T.*

Proof. We proceed by induction on the updates of $[s_x, c_x]$ during computation of the minimisation procedure. Our base case is the first version of the counterexample $[s_x, c_x]$ as produced by step 2 of the algorithm. This multi-clause is full, since it is the output of function $TClosure$ that produces full multi-clauses by definition.

To see that the final 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.

To see that the final multi-clause is closed, we prove first that after generalising a term the resulting counterexample is closed. Let $[s_x, c_x]$ be the multi-clause before generalising t and $[s'_x, c'_x]$ after. Let the substitution θ_t be $\{x_t \mapsto t\}$. Then, $s'_x \cdot \theta_t = s_x$ and $c_x = c'_x \cdot \theta_t$, because x_t is a new variable that does not appear in $[s_x, c_x]$. By way of contradiction, suppose that some atom $b \in Atoms_P(s'_x \cup c'_x) \setminus s'_x$ such that $T \models s'_x \to b$ is not in c'_x. Notice that the substitution θ_t is non-unifying w.r.t. $s'_x \cup c'_x$, and therefore using properties 2. and 4. in Lemma 4.3 we conclude that $b \cdot \theta_t \in Atoms_p(s_x \cup c_x) \setminus s_x$ and $b \cdot \theta_t \notin c_x$. Since $T \models s_x \to b \cdot \theta_t$, this contradicts our (implicit) induction hypothesis stating that $[s_x, c_x]$ is closed, since the atom $b \cdot \theta_t$ would be missing. Hence, any counterexample $[s_x, c_x]$ after step 3 is closed.

We will show now that after dropping some term t the multi-clause still remains closed. 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 only removing atoms. By the induction hypothesis, the only atoms that could be missing are atoms in $c_x \setminus c'_x$ and $s_x \setminus s'_x$. Since for the closure of $[s'_x, c'_x]$ we only consider atoms in $Atoms_P(s'_x \cup c'_x)$ and these atoms do not contain t (all occurrences have been removed), the removed atoms cannot be missing because they all contain t. Therefore, after step 6 and as returned by the minimisation procedure, the counterexample $[s_x, c_x]$ is closed. ■

LEMMA 4.8. *All counterexamples given by the equivalence query oracle are positive w.r.t. the target T and the hypothesis H.*

Proof. The algorithm makes sure that all clauses in H are correct (lines 3 and 6 of Algorithm 3 and lines 2, 5 and 8 of Algorithm 4). Therefore, $T \models H$. ■

LEMMA 4.9. *Every minimised $[s_x, c_x]$ is a positive counterexample w.r.t. target T and 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 $H \not\models s_x \to b_x$. By Lemma 4.7, 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 $A \to a$ be the original counterexample obtained from the equivalence oracle. This $A \to a$ is such that $T \models A \to a$ but $H \not\models A \to a$ (by Lemma 4.8), and therefore a will not be included in the antecedent of the first $[s_x, c_x]$ by $HClosure$ because it is not implied by H. However, a is included in c_x because $a \in Atoms_P(A \to a)$ and $T \models A \to a$. Thus, $c_x \neq \emptyset$ after step 2 of the minimisation procedure. Moreover, the call to the procedure $HClosure$ guarantees that every atom implied by H will be put into the antecedent s_x, leaving no space for any atom implied by H to be put into the consequent c_x by $TClosure$. Thus, after step 2, $[s_x, c_x]$ is a 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_t be the substitution $\{x_t \mapsto t\}$. As in Lemma 4.7, $s'_x \cdot \theta_t = s_x$ and $c'_x \cdot \theta_t = c_x$. Suppose by way of contradiction that $H \models s'_x \to b'$, for some $b' \in c'_x$. Then, $H \models s'_x \cdot \theta_t \to b' \cdot \theta_t$. And we get that $H \models s_x \to b' \cdot \theta_t$. Note that $b' \in c'_x$ implies that $b' \cdot \theta_t \in c_x$. This contradicts our assumption stating that $[s_x, c_x]$ was a counterexample. Thus, the multi-clause $[s_x, c_x]$ after step 3 is a positive 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 atoms, and $[s'_x, c'_x]$ after. It is the case that $s'_x \subseteq s_x$ and $c'_x \subseteq c_x$. Assume $[s_x, c_x]$ is a positive counterexample. Then, for all $b \in c_x : H \not\models s_x \to b$. Since $c'_x \subseteq c_x$, it holds that for all $b \in c'_x : H \not\models s_x \to b$. Since $s'_x \subseteq s_x$, we obtain that for all $b \in c'_x : H \not\models s'_x \to b$. Thus, the multi-clause $[s_x, c_x]$ after step 6 is a positive counterexample. ■

LEMMA 4.10. *If a minimised $[s_x, c_x]$ captures some clause $\neq (s_t \to b_t)$ of $U(T)$, then it must be via some substitution θ such that θ is a variable renaming, i.e., θ maps distinct variables of s_t into distinct variables of s_x only.*

Proof. $[s_x, c_x]$ is capturing $\neq (s_t \to b_t)$, hence there must exist a substitution θ from variables in $s_t \cup \{b_t\}$ into terms in $s_x \cup c_x$ such that $s_t \cdot \theta \subseteq s_x$, $ineq(s_t \cup \{b_t\}) \cdot \theta \subseteq ineq(s_x \cup c_x)$ 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 $s_t \cup \{b_t\}$ into a functional term t of $s_x \cup c_x$ (i.e. $v \cdot \theta = t$). Consider the generalisation of the term t in step 3 of the minimisation procedure. We will see that the term t should have been generalised and substituted by the new variable x_t.

Suppose, then that $[s_x, c_x]$ is the multi-clause previous to generalising t and $[s'_x, c'_x]$ after. We generalise the term t to the fresh variable x_t. Consider the substitution θ' defined as $\theta \setminus \{v \mapsto t\} \cup \{v \mapsto x_t\}$. The substitution θ' behaves like θ on all terms except for variable v. We will see that $[s'_x, c'_x]$ captures $\neq (s_t \to b_t)$ via θ' and hence $rhs(s'_x, c'_x) \neq \emptyset$ (Lemma 4.6). Therefore t must be generalised to the variable x_t.

To see that $[s'_x, c'_x]$ captures $\neq (s_t \to b_t)$ via θ' we need to show (1) $s_t \cdot \theta' \subseteq s'_x$, (2) $b_t \cdot \theta' \in c'_x$ and (3) $ineq(s_t \cup \{b_t\}) \cdot \theta' \subseteq ineq(s'_x \cup c'_x)$. For (1), consider any atom b of s_t. We observe the following: after substitution θ': $b(...v...) \Rightarrow b(...x_t...)$, and after substitution θ and generalising t: $b(...v...) \Rightarrow b(...t...) \Rightarrow b(...x_t...)$. The part of the "dots" in the previous expressions is identical for both lines, since θ and θ' behave equally for terms different than v. Moreover, the fact that θ does not unify terms in $s_t \cup \{b_t\}$ assures that the rest of terms will differ from t and x_t after applying θ or θ'. Therefore, we get that $b \cdot \theta' \in s'_x$ iff $b \cdot \theta \in s_x$ and since $s_t \cdot \theta \subseteq s_x$, Property

(1) follows. Property (2) is identical to Property (1). For (3), let t, t' be two distinct terms of $s_t \cup \{b_t\}$. We have to show that $t \cdot \theta'$ and $t' \cdot \theta'$ are two different terms of $s'_x \cup c'_x$ and therefore their inequality appears in $ineq(s'_x \cup c'_x)$. It is easy to see that they are terms of $s'_x \cup c'_x$ since by previous properties $(s_t \cup \{b_t\}) \cdot \theta' \subseteq (s'_x \cup c'_x)$. Now, let θ_t be the substitution $\{x_t \to t\}$ and notice that $\theta = \theta' \cdot \theta_t$. Since θ does not unify terms in $s_t \cup \{b_t\}$, then none of θ' and θ_t do. Therefore, $t \cdot \theta' \neq t' \cdot \theta'$ as required. ■

4.7. Properties of the number of terms in minimised examples

LEMMA 4.11. *Let $[s_x, c_x]$ be a multi-clause as output by the minimisation procedure. Let $\neq (s_t \to b_t)$ be a clause of $U(T)$ captured by $[s_x, c_x]$. Then, the number of distinct terms in $[s_x, c_x]$ is equal to the number of distinct terms in $\neq (s_t \to b_t)$.*

Proof. Let n_x and n_t be the number of distinct terms appearing in $[s_x, c_x]$ and $s_t \to b_t$, respectively. Subterms should also be counted. The multi-clause $[s_x, c_x]$ captures $\neq (s_t \to b_t)$. Therefore there is a substitution θ satisfying $ineq(s_t \cup \{b_t\}) \cdot \theta \subseteq ineq(s_x \cup c_x)$. Thus, different variables in $s_t \to b_t$ are mapped into different terms of $s_x \cup c_x$ by θ. By Lemma 4.10, we know also that every variable of s_t, b_t is mapped into a variable of s_x, c_x. Therefore, θ maps distinct variables of s_t, b_t into distinct variables of s_x, c_x. Therefore, the number of terms in s_t, b_t equals the number of terms in $(s_t \cup \{b_t\}) \cdot \theta$, since there has only been a non-unifying renaming of variables. Also, $s_t \cdot \theta \subseteq s_x$ and $b_t \cdot \theta \in c_x$. We have to check that the remaining atoms in $(s_x \setminus s_t \cdot \theta) \cup (c_x \setminus b_t \cdot \theta)$ do not include any term not appearing in $(s_t \cup \{b_t\}) \cdot \theta$.

Suppose there is an atom $l \in (s_x \setminus s_t \cdot \theta) \cup (c_x \setminus b_t \cdot \theta)$ containing some term, say t, not appearing in $(s_t \cup \{b_t\}) \cdot \theta$. Consider when in step 6 of the minimisation procedure the term t was checked as a candidate to be removed. Let $[s'_x, c'_x]$ be the clause obtained after the removal of the atoms containing t. Then, $s_t \cdot \theta \subseteq s'_x$ and $b_t \cdot \theta \in c'_x$ because all the atoms in $(s_t \cup \{b_t\}) \cdot \theta$ do not contain t. Moreover, $ineq(s_t \cup \{b_t\}) \cdot \theta \subseteq ineq(s'_x \cup c'_x)$. To see this, take any two terms $t \neq t'$ from $s_t \to b_t$. The terms $t \cdot \theta$ and $t' \cdot \theta$ appear in $s'_x \cup c'_x$ because they contain terms in $(s_t \cup b_t) \cdot \theta$ only (so they are not removed). Further, since $t \cdot \theta \neq t' \cdot \theta$ in $s_x \cup c_x$ and $\{t \cdot \theta, t' \cdot \theta\} \subseteq (s'_x \cup c'_x) \subseteq (s_x \cup c_x)$ we conclude that $t \cdot \theta \neq t' \cdot \theta$ in $s'_x \cup c'_x$. Thus, $[s'_x, c'_x]$ still captures $\neq (s_t \to b_t)$. And therefore, $rhs(s'_x, c'_x) \neq \emptyset$ and such a term t cannot exist. We conclude that $n_t = n_x$. ■

COROLLARY 4.4. *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.*

LEMMA 4.12. *Let $[s, c]$ be a multi-clause covering some $\neq (s_t \to b_t)$. Let n and n_t be the number of distinct terms in $s \cup c$ and $s_t \cup \{b_t\}$, respectively. Then, $n_t \leq n$.*

Proof. Since $[s, c]$ covers the clause $\neq (s_t \rightarrow b_t)$, there is a θ s.t. $ineq(s_t \cup \{b_t\}) \cdot \theta \subseteq ineq(s \cup c)$. Therefore, any two distinct terms of $s_t \cup \{b_t\}$ appear as distinct terms in $s \cup c$. And therefore, $[s, c]$ has at least as many terms as $s_t \rightarrow b_t$. ■

COROLLARY 4.5. *Let $\neq (s_t \rightarrow b_t)$ 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 $[s_x, c_x]$ captures the clause $\neq (s_t \rightarrow b_t)$. Let $[s_i, c_i]$ be a multi-clause with n_i terms covering the clause $\neq (s_t \rightarrow b_t)$. Then $n_x \leq n_i$.*

4.8. Properties of pairings

LEMMA 4.13. *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 closed. That is, whenever $T \models s \rightarrow b$ and $b \in Atoms_P(s \cup c) \setminus s$, then $b \in c$. Suppose, then, that $T \models s \rightarrow b$ with $b \in Atoms_P(s \cup c) \setminus s$. 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 $s \cdot \theta_i \subseteq s_i$. $T \models s \rightarrow b$ 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 $b_i \in s_i \cup c_i$ because $b_x \in Atoms_P(s_x \cup c_x)$ and $b_i \in Atoms_P(s_i \cup c_i)$ (remember that $b \in Atoms_P(s \cup c)$). 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 $s \cup c$, because $b \in Atoms_P(s \cup c)$. We know that σ is basic and hence legal, and therefore it contains all subterms of terms appearing in $s \cup c$. 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 $s \cup c$ 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 4.14. *Let $[s, c]$ be a pairing of two multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$. Then, it is the case that $|s| \leq |s_i|$ and $|s \cup c| \leq |s_i \cup c_i|$.*

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

LEMMA 4.15. *Let $[s_1, c_1]$ and $[s_2, c_2]$ be two full multi-clauses w.r.t. some Horn expression T. Let $[s, c]$ be any legal pairing between them. The following holds:*

1. *If $[s, c]$ covers a clause $\neq (s_t \rightarrow b_t)$ in $U(T)$, then both $[s_1, c_1]$ and $[s_2, c_2]$ cover $\neq (s_t \rightarrow b_t)$.*

2. *If* $[s, c]$ *captures a clause* $\neq (s_t \rightarrow b_t)$ *in* $U(T)$*, then at least one of* $[s_1, c_1]$ *or* $[s_2, c_2]$ *captures* $\neq (s_t \rightarrow b_t)$*.*

Proof. Condition 1. By assumption, $\neq (s_t \rightarrow b_t)$ is covered by $[s, c]$, i.e., there is a θ such that $s_t \cdot \theta \subseteq s$, $ineq(s_t \cup \{b_t\}) \cdot \theta \subseteq ineq(s \cup c)$ and $b_t \cdot \theta \in Atoms_P(s \cup c)$. This implies that if t, t' are two distinct terms of $s_t \cup \{b_t\}$, then $t \cdot \theta$ and $t' \cdot \theta$ are distinct terms appearing in $s \cup c$. Let σ be the 1-1 legal matching inducing the pairing. The antecedent s is defined to be $lgg_{|\sigma}(s_1, s_2)$, and therefore there exist substitutions θ_1 and θ_2 such that $s \cdot \theta_1 \subseteq s_1$ and $s \cdot \theta_2 \subseteq s_2$. We claim that $[s_1, c_1]$ and $[s_2, c_2]$ cover $\neq (s_t \rightarrow b_t)$ via $\theta \cdot \theta_1$ and $\theta \cdot \theta_2$, respectively. We will prove this for $[s_1, c_1]$ only, the proof for $[s_2, c_2]$ is identical. 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 $s_t \cdot \theta \cdot \theta_1 \subseteq s_1$. We show now that $ineq(s_t \cup \{b_t\}) \cdot \theta \cdot \theta_1 \subseteq ineq(s_1 \cup c_1)$. Observe that all top-level terms appearing in $s \cup c$ also appear as one entry of the matching σ, because otherwise they could not have survived the restriction imposed by σ. Further, since σ is legal, all subterms of terms of $s \cup c$ also appear as an entry in σ. Let t, t' be two distinct terms appearing in $s_t \cup \{b_t\}$. Since $(s_t \cup \{b_t\}) \cdot \theta \subseteq s \cup c$ and σ includes all terms appearing in $s \cup c$, the distinct terms $t \cdot \theta$ and $t' \cdot \theta$ appear as the lgg of distinct entries in σ. These entries have the form `[`$t \cdot \theta \cdot \theta_1$ `-` $t \cdot \theta \cdot \theta_2$ `=>` $t \cdot \theta$`]`, since $lgg(t \cdot \theta \cdot \theta_1, t \cdot \theta \cdot \theta_2) = t \cdot \theta$. Since σ is 1-1, we know that $t \cdot \theta \cdot \theta_1 \neq t' \cdot \theta \cdot \theta_1$. Finally, we need to show that $b_t \cdot \theta \cdot \theta_1 \in Atoms_P(s_1 \cup c_1)$. Notice that $s \cdot \theta_1 \subseteq s_1$ and $c \cdot \theta_1 \subseteq (s_1 \cup c_1)$. Therefore, $s_t \cup \{b_t\} \cdot \theta \subseteq s \cup c$ implies $s_t \cup \{b_t\} \cdot \theta \cdot \theta_1 \subseteq (s \cup c) \cdot \theta_1 \subseteq s_1 \cup c_1$. Thus, $b_t \cdot \theta \cdot \theta_1 \in Atoms_P(s_1 \cup c_1)$ as required.

Condition 2. By hypothesis, $b_t \cdot \theta \in c$ and c is defined to be $lgg_{|\sigma}(s_1, c_2) \cup lgg_{|\sigma}(c_1, s_2) \cup lgg_{|\sigma}(c_1, c_2)$. Observe that all these lggs share the same table, so the same pairs of terms will be mapped into the same expressions. Observe also that the substitutions θ_1 and θ_2 are defined according to this table, so that if any atom $l \in lgg_{|\sigma}(c_1, \cdot)$, 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 $\neq (s_t \rightarrow b_t)$, one of $[s_1, c_1]$ or $[s_2, c_2]$ captures $\neq (s_t \rightarrow b_t)$. ■

It is crucial for Lemma 4.15 that the pairing involved is *legal.* It is indeed possible for a *non-legal* pairing to capture some clause that is not even covered by some of its originating multi-clauses, as the next example illustrates.

EXAMPLE 4.3. In this example we present two multi-clauses $[s_1, c_1]$ and $[s_2, c_2]$, a non-legal matching σ and a clause $\neq (s_t \rightarrow b_t)$ such that the non-legal pairing induced by σ captures $\neq (s_t \rightarrow b_t)$ but none of $[s_1, c_1]$ and $[s_2, c_2]$ do.

- $[s_1, c_1] = [p(ffa, gffa) \rightarrow q(fa)]$ with terms $\{a, fa, ffa, gffa\}$
$ineq(s_1) = (a \neq fa \neq ffa \neq gffa)$.
- $[s_2, c_2] = [p(fb, gffc) \rightarrow q(b)]$ with terms $\{b, c, fb, fc, ffc, gffc\}$.
- The matching σ is `[a - c => X]`
  `[fa - b => Y]`

```
[ffa - fb => fY]
[gffa - gffc => gffX]
```

- $[s,c] = [p(fY, gffX) \to q(Y)]$.
- $\underbrace{(x \neq fx \neq ffx \neq gffx \neq y \neq fy)}_{ineq(s_t)}, \underbrace{p(fy, gffx)}_{s_t} \to \underbrace{q(y)}_{b_t}$.
- $\theta = \{x \mapsto X, y \mapsto Y\}$.
- $\theta_1 = \{X \mapsto a, Y \mapsto fa\}$.
- $\theta \cdot \theta_1 = \{x \mapsto a, y \mapsto fa\}$.

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

$$\underbrace{(a \neq \boxed{\text{fa}} \neq \boxed{\text{ffa}} \neq gffa \neq \boxed{\text{fa}} \neq \boxed{\text{ffa}})}_{(x \neq fx \neq ffx \neq gffx \neq y \neq fy) \cdot \theta \cdot \theta_1} \not\subseteq \underbrace{(a \neq fa \neq ffa \neq gffa)}_{ineq(s_1)}$$

COROLLARY 4.6. *Let $[s_1, c_1], [s_2, c_2], [s_3, c_3], ..., [s_k, c_k], ...$ be a sequence of full multi-clauses such that every multi-clause $[s_{i+1}, c_{i+1}]$ is a legal pairing between the previous multi-clause $[s_i, c_i]$ in the sequence and some other full multi-clause $[s'_i, c'_i]$, for $i \geq 1$. Suppose some $[s_k, c_k]$ in the sequence covers a clause $\neq (s_t \to b_t)$. Then, all previous $[s_i, c_i]$ in the sequence (where $i < k$), must cover the clause $\neq (s_t \to b_t)$, too.*

4.9. Properties of the sequence S

COROLLARY 4.7. *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 4.7 guarantees that all minimised counterexamples are full and, by Lemma 4.13, any basic pairing between full multi-clauses is also full. ∎

LEMMA 4.16. *Let S be the sequence $[[s_1, c_1], [s_2, c_2], ..., [s_k, c_k]]$. If a minimised counterexample $[s_x, c_x]$ is produced such that it captures some clause $\neq (s_t \to b_t)$ 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)$rhs(s,c) \neq \emptyset$ and (2)$size(s) \lneq size(s_i)$ or ($size(s) = size(s_i)$ and $size(c) \lneq size(c_i)$).

We have assumed that there is some clause $\neq (s_t \to b_t) \in U(T)$ captured by $[s_x, c_x]$ and covered by $[s_i, c_i]$. Let θ'_x be the substitution showing that $\neq (s_t \to b_t)$

is captured by $[s_x, c_x]$ and θ'_i the substitution showing that $\neq (s_t \rightarrow b_t)$ is covered by $[s_i, c_i]$. Thus the following properties hold:

- $s_t \cdot \theta'_x \subseteq s_x$
- $ineq(s_t \cup \{b_t\}) \cdot \theta'_x \subseteq ineq(s_x \cup c_x)$
- $b_t \cdot \theta'_x \in c_x$
- $b_t \cdot \theta'_x \in Atoms_P(s_x \cup c_x)$
- $s_t \cdot \theta'_i \subseteq s_i$
- $ineq(s_t \cup \{b_t\}) \cdot \theta'_i \subseteq ineq(s_i \cup c_i)$
- $b_t \cdot \theta'_i \in Atoms_P(s_i \cup c_i)$

We construct a matching σ that includes all entries

$$[t \cdot \theta'_x \texttt{ - } t \cdot \theta'_i \texttt{ => } lgg(t \cdot \theta'_x, t \cdot \theta'_i)]$$

such that t is a term appearing in $s_t \cup \{b_t\}$ (one entry for every distinct term).

EXAMPLE 4.4. Consider the following:

- $s_t = \{p(g(c), x, f(y), z)\}$.

With terms $c, g(c), x, y, f(y)$ and z.

- $s_x = \{p(g(c), x', f(y'), z), p(g(c), g(c), f(y'), c)\}$.

With terms $c, g(c), x', y', f(y')$ and z.

- $s_i = \{p(g(c), f(1), f(f(2)), z)\}$.

With terms $c, g(c), 1, f(1), 2, f(2), f(f(2))$ and z.

- The substitution $\theta'_x = \{x \mapsto x', y \mapsto y', z \mapsto z\}$ and it is a variable renaming.
- The substitution $\theta'_i = \{x \mapsto f(1), y \mapsto f(2), z \mapsto z\}$.
- The $lgg(s_x, s_i)$ is $\{p(g(c), X, f(Y), z), p(g(c), Z, f(Y), V)\}$ and it produces the following lgg table.

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

- The extended matching σ is

$$\begin{array}{rcl} c & \Rightarrow & \texttt{[c - c => c]} \\ g(c) & \Rightarrow & \texttt{[g(c) - g(c) => g(c)])} \\ x & \Rightarrow & \texttt{[x' - f(1) => X]} \\ y & \Rightarrow & \texttt{[y' - f(2) => Y]} \\ f(y) & \Rightarrow & \texttt{[f(y') - f(f(2)) => f(Y)]} \\ z & \Rightarrow & \texttt{[z - z => z]} \end{array}$$

- The pairing induced by σ is $lgg_{|_\sigma}(s_x, s_i) = \{p(g(c), X, f(Y), z)\}$.

Claim. The matching σ as described above is 1-1 and the number of entries equals the minimum of the number of distinct terms in $s_x \cup c_x$ and $s_i \cup c_i$.

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

$$
\begin{array}{l}
[t \cdot \theta'_x \texttt{ - } t \cdot \theta'_i \texttt{ => } lgg(t \cdot \theta'_x, t \cdot \theta'_i)] \\
[t' \cdot \theta'_x \texttt{ - } t' \cdot \theta'_i \texttt{ => } lgg(t' \cdot \theta'_x, t \cdot \theta'_i)]
\end{array}
$$

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 $\neq (s_t \to b_t)$ via θ'_x and θ'_i, respectively. Therefore $ineq(s_t \cup \{b_t\}) \cdot \theta'_x \subseteq ineq(s_x \cup c_x)$ and $ineq(s_t \cup \{b_t\}) \cdot \theta'_i \subseteq ineq(s_i \cup c_i)$. And therefore $t \cdot \theta'_x$ and $t' \cdot \theta'_x$ appear as different terms in $s_x \cup c_x$. Also, $t \cdot \theta'_i$ and $t' \cdot \theta'_i$ appear as different terms in $s_i \cup c_i$. Thus σ is 1-1.

By construction, the number of entries equals the number of distinct terms in $s_t \cup \{b_t\}$, that by Lemma 4.11 is the number of distinct terms in $s_x \cup c_x$. And by Lemma 4.12, $[s_i, c_i]$ contains at least as many terms as $s_t \cup \{b_t\}$. Therefore, the number of entries in σ coincides with the minimum of the number of distinct terms in $s_x \cup c_x$ and $s_i \cup c_i$. ■

Claim. The matching σ is legal.

Proof. A matching is legal if the subterms of any term appearing as the lgg of the matching, also appear in some other entries of the matching. We will prove it by induction on the structure of the terms. We prove that if t is a term in $s_t \cup \{b_t\}$, then the term $lgg(t \cdot \theta'_x, t \cdot \theta'_i)$ and all its subterms appear somewhere in the extension 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 $s_t \cup \{b_t\}$. The entry for it in σ is $[v \cdot \theta'_x \texttt{ - } v \cdot \theta'_i \texttt{ => } lgg(v \cdot \theta'_x, v \cdot \theta'_i)]$. $[s_x, c_x]$ is minimised and by Lemma 4.10 $v \cdot \theta'_x$ must be a variable. Therefore, its 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 $[t_j \cdot \theta'_x \texttt{ - } t_j \cdot \theta'_i \texttt{ => } lgg(t_j \cdot \theta'_x, t_j \cdot \theta'_x)]$, with $1 \leq j \leq l$, are also included in σ, since all t_j are terms of $s_t \cup \{b_t\}$. 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. 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 $s_x \cup c_x$ is less or equal than the number of terms in $s_i \cup c_i$. Corollary 4.5 shows that this is indeed the case. Following the definition, it should be also 1-1 and legal. The claims above show that it is 1-1 and that it is also legal. It is only left to see that it is basic: if entry $f(t_1, ..., t_n) - t$ is in σ, then $t = f(r_1, ..., r_n)$ and $t_l - r_l \in \sigma$ for all $l = 1, ..., n$.

Suppose, then, that $f(t_1, ..., t_n) - t$ is in σ. By construction of σ all entries are of the form $\hat{t} \cdot \theta'_x - \hat{t} \cdot \theta'_i$, where $\hat{t}$ is a term in $s_t \cup \{b_t\}$. Thus, assume $\hat{t} \cdot \theta'_x = f(t_1, ..., t_n)$ and $\hat{t} \cdot \theta'_i = t$. We also know that θ'_x is a variable renaming, therefore, the term $\hat{t} \cdot \theta'_x$ is a variant of $\hat{t}$. Therefore, the terms $f(t_1, ..., t_n)$ and $\hat{t}$ are variants. That is, $\hat{t}$ itself has the form $f(t'_1, ..., t'_n)$, where every t'_j is a variant of t_j and $t'_j \cdot \theta'_x = t_j$, where $j = 1, ..., n$. Therefore, $t = \hat{t} \cdot \theta'_i = f(r_1 = t'_1 \cdot \theta'_i, ..., r_n = t'_n \cdot \theta'_i)$ as required. We have seen that $t_j = t'_j \cdot \theta'_x$ and $r_j = t'_j \cdot \theta'_i$. By construction, σ includes the entries $t_j - r_j$, for any $j = 1, ..., n$ and our claim holds. ■

The claims above show that the matching σ is a good matching in the sense that it will be one of the matchings constructed by the algorithm. Now we consider the pairing of $[s_x, c_x]$ and $[s_i, c_i]$ induced by σ. Notice that this pairing (call it $[s, c]$) will not be discarded by our algorithm. 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 $[s_x, c_x]$ and $[s_i, c_i]$.

It is left to show that conditions for replacement in the algorithm hold. The following two claims show that this is indeed the case.

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

Proof. 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 $s_t \cup \{b_t\}$.

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

- $s \cdot \theta_x \subseteq s_x$. Let l be an atom in s, l has been obtained by taking the lgg of two atoms 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 atoms 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.

Let θ be the substitution that maps all variables in $s_t \cup \{b_t\}$ to their corresponding expression assigned in the extension of σ. That is, θ maps any variable v of $s_t \cup \{b_t\}$ to the term $lgg(v \cdot \theta'_x, v \cdot \theta'_i)$. In our example, $\theta = \{x \mapsto X, y \mapsto Y, z \mapsto z\}$.

The proof that $rhs(s,c) \neq \emptyset$ consists in showing that $\neq (s_t \to b_t)$ is captured by $[s,c]$ via θ. Then we apply Lemma 4.6 and conclude that $rhs(s,c) \neq \emptyset$.

The following properties hold:

• $\theta \cdot \theta_x = \theta'_x$. Let v be a variable in $s_t \cup \{b_t\}$. 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.

To see how $\neq (s_t \to b_t)$ is captured by $[s,c]$ via θ:

• $s_t \cdot \theta \subseteq s = lgg_{|\sigma}(s_x, s_i)$. Let l be an atom in s_t. We show that $l \cdot \theta$ is in $lgg(s_x, s_i)$ and that it is not removed by the restriction to σ. Let t be a term appearing in l. The matching σ contains the entry

$$\texttt{[}t \cdot \theta'_x \texttt{ - } t \cdot \theta'_i \texttt{ => } lgg(t \cdot \theta'_x, t \cdot \theta'_i)\texttt{]},$$

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 $s_t \cup \{b_t\}$ (and thus for every variable in s_t), therefore $t \cdot \theta = lgg(t \cdot \theta'_x, t \cdot \theta'_i)$. Indeed, $lgg(t \cdot \theta'_x, t \cdot \theta'_i)$ appears in σ. The atom $l \cdot \theta$ 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 $\theta = \{v \mapsto lgg(v \cdot \theta'_x, v \cdot \theta'_i) \mid v \text{ is a variable of } s_t\}$. 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 $\{p(g(c), X, f(Y), z)\}$.

• $ineq(s_t \cup \{b_t\}) \cdot \theta \subseteq ineq(s \cup c)$. We have to show that for any two distinct terms t, t' of $s_t \cup \{b_t\}$, the terms $t \cdot \theta$ and $t' \cdot \theta$ are also different terms in $s \cup c$, and therefore the inequality $t \cdot \theta \neq t' \cdot \theta$ appears in $ineq(s \cup c)$. By hypothesis, $ineq(s_t \cup \{b_t\}) \cdot \theta'_x \subseteq ineq(s_x \cup c_x)$. Since $\theta'_x = \theta \cdot \theta_x$, we get $ineq(s_t \cup \{b_t\}) \cdot \theta \cdot \theta_x \subseteq ineq(s_x \cup c_x)$ and so $t \cdot \theta \cdot \theta_x$ and $t' \cdot \theta \cdot \theta_x$ are different terms of $s_x \cup c_x$. From Property 5. in Lemma 4.3 it follows that $t \cdot \theta \neq t' \cdot \theta \in ineq(s \cup c)$.

• $b_t \cdot \theta \in c$. By hypothesis, $b_t \cdot \theta'_x \in c_x$. Also, $b_t \cdot \theta'_i \in Atoms_P(s_i \cup c_i)$ implies (because $[s_i, c_i]$ is full), that $b_t \cdot \theta'_i \in s_i \cup c_i$. Notice that $b_t \cdot \theta = lgg_{|\sigma}(b_t \cdot \theta'_x, b_t \cdot \theta'_i)$ by construction. Therefore $b_t \cdot \theta \in c = lgg_{|\sigma}(s_x, c_i) \cup lgg_{|\sigma}(c_x, s_i) \cup lgg_{|\sigma}(c_x, c_i)$ as required.

■

Claim. $size(s) \lneq size(s_i)$ or $(size(s) = size(s_i)$ and $size(c) \lneq size(c_i))$.

Proof. By Lemma 4.14, 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. Notice that the *lgg* substitutes variables for functional terms. According to our definition of size, variables weigh less than functional terms, therefore the size of a generalisation will be at most the size of the instance that has been generalised. We cover all possible cases: if $size(s) \lneq size(s_i)$, then the condition is true. If $size(s) = size(s_i)$, then we know by Lemma 4.14 that $|s \cup c| \leq |s_i \cup c_i|$. Since $|s| = |s_i|$, we conclude that $|c| \leq |c_i|$, and hence $size(c) \leq size(c_i)$ by the same argument as above. Thus, $s \cdot \theta_i = s_i$ and $s_i \cdot \theta_i^{-1} = s$. Again, we split the proof into two cases. The case when $size(c) \lneq size(c_i)$ satisfies the condition. For the case when $size(c) = size(c_i)$, we have that the multi-clauses $[s, c]$ and $[s_i, c_i]$ are equal up to variable renaming. We will elaborate this case a little more and will arrive to a contradiction, finishing our proof. The following facts hold:

- Since $[s, c]$ and $[s_i, c_i]$ are variable renamings, $c \cdot \theta_i = c_i$ and $c_i \cdot \theta_i^{-1} = c$.
- By the previous claim, it holds that $b_t \cdot \theta \in c$ and therefore there exists a b_i s.t. $b_i = b_t \cdot \theta \cdot \theta_i \in c_i$.
- The substitutions θ_i and θ'_x are variable renamings, and (by previous claim) $\theta'_x = \theta \cdot \theta_x$, therefore the substitution $\hat{\theta} = \theta_i^{-1} \cdot \theta_x$ is well defined and is a variable renaming.
- It follows that $s_i \cdot \hat{\theta} \subseteq s_x$ and $b_i \cdot \hat{\theta} = \underbrace{b_t \cdot \theta \cdot \theta_i}_{b_i} \cdot \underbrace{\theta_i^{-1} \cdot \theta_x}_{\hat{\theta}} = b_t \cdot \theta \cdot \theta_x = b_t \cdot \theta'_x \in c_x$ (by assumption).

Therefore, $H \models s_i \rightarrow b_i \models s_i \cdot \hat{\theta} \rightarrow b_i \cdot \hat{\theta} \models s_x \rightarrow b_x$ (where $b_x = b_t \cdot \theta'_x \in c_x$) contradicting the fact that $[s_x, c_x]$ is a counterexample. ■

This completes the proof for the lemma. ■

COROLLARY 4.8. *If a counterexample $[s_x, c_x]$ is appended to S, it is because there is no element in S capturing a clause in $U(T)$ that is also captured by $[s_x, c_x]$.*

LEMMA 4.17. *Every time the algorithm is about to make an equivalence query, it is the case that every multi-clause in S captures at least one of the clauses of $U(T)$ and every clause of $U(T)$ is captured by at most one multi-clause in S.*

Proof. All multi-clauses included in S are full by Corollary 4.7. By construction, their consequents are non-empty so that we can apply Corollary 4.2, and conclude that all counterexamples in S capture some clause of $U(T)$.

An induction on the number of iterations of the main loop in line 2 of the learning algorithm shows that no two different multi-clauses in S capture the same clause of $U(T)$. 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 4.8, we know that $[s_x, c_x]$ does not capture any clause in $U(T)$ also captured 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 $[s'_i, c'_i]$ captures some clause of $U(T)$, say $\neq (s_t \to b_t)$ also captured 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 4.15 we conclude that one of $[s_x, c_x]$ or $[s_i, c_i]$ captures $\neq (s_t \to b_t)$.

Suppose $[s_i, c_i]$ captures $\neq (s_t \to b_t)$. This contradicts the induction hypothesis, since both $[s_i, c_i]$ and $[s_j, c_j]$ appear in S and capture $\neq (s_t \to b_t)$ in $U(T)$.

Suppose $[s_x, c_x]$ captures $\neq (s_t \to b_t)$. If $j < i$, then $[s_x, c_x]$ would have refined $[s_j, c_j]$ instead of $[s_i, c_i]$ (Lemma 4.16). Therefore, $j > i$. But then we are in a situation where $[s_j, c_j]$ captures a clause also covered by $[s_i, c_i]$. By Corollary 4.6, all multi-clauses in position i cover $\neq (s_t \to b_t)$ during the history of S. Consider the iteration in which $[s_j, c_j]$ first captured $\neq (s_t \to b_t)$. This could have happened by appending the counterexample $[s_j, c_j]$, which contradicts Lemma 4.16 since $[s_i, c_i]$ or an ancestor of it was covering $\neq (s_t \to b_t)$ but was not replaced. Or it could have happened by refining $[s_j, c_j]$ with a pairing of a counterexample capturing $\neq (s_t \to b_t)$. But then, by Lemma 4.16 again, the element in position i should have been refined, instead of refining $[s_j, c_j]$. ■

4.10. Deriving the complexity bounds

Recall that m' stands for the number of clauses in the transformation $U(T)$ and that by Lemma 4.1, $m' \leq mt^k$, where t (k, resp.) is the maximum number of terms (variables, resp.) in any clause in T. By Lemma 4.17 the number of clauses in $U(T)$ bounds the number of elements in S, and therefore:

COROLLARY 4.9. *The number of elements in S is bounded by m'.*

What follows is a detailed account of the number of queries made in every procedure.

LEMMA 4.18. *If $[s_x, c_x]$ is a minimised counterexample, then, $|s_x| + |c_x| \leq st^a$.*

Proof. By Corollary 4.4, there are a maximum of t terms in a minimised counterexample. There are a maximum of st^a different atoms built up from t terms. ■

LEMMA 4.19. *The algorithm makes $O(m'st^a)$ equivalence queries.*

Proof. Notice that any set of atoms containing t distinct terms can be generalised at most t times. This is because after generalising a term into a variable, it cannot be further generalised. The sequence S has at most m' elements. The following actions can happen after refining a multi-clause in S (possibly combined): either (1) one atom is dropped from the antecedent, or (2) an atom moves from antecedent to consequent, or (3) an atom is dropped from the consequent, or (4) some term is generalised. This can happen $m'st^a$ times for (1), $m'st^a$ times for (2), $m'st^a$

times for (3), and $m't$ times for (4), that is $m'(t+3st^a)$ in total. We need m' extra calls to add all the counterexamples. In total $\underline{m'}(1+t+3\underline{st^a})$, that is $O(m'st^a)$. ■

Lemma 4.20. *The algorithm makes $O(se_t^{a+1})$ membership queries during the minimisation procedure.*

Proof. To compute the first version of the full multi-clause we need to test the se_t^a possible atoms 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 atoms. 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 4.21. *Given a matching, the algorithm makes at most st^a membership queries during the computation of a basic pairing.*

Proof. The number of atoms in the consequent c of a pairing of $[s_x, c_x]$ and $[s_i, c_i]$ is bounded by the number of atoms in s_x plus the number of atoms in c_x. By Lemma 4.18, this is bounded by st^a. ■

Lemma 4.22. *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:

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

■

We arrive to our main result.

Theorem 4.1. *The algorithm exactly identifies every closed 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. Furthermore, the running time is polynomial in ${m'}^2 + s^2 + t^k + t^a + e_t^a$.*

We conclude that the classes $RRHE$, $COHE$ and $RRCOHE$ are learnable using our algorithm. Since by Lemma 4.1 we know that $m' \leq mt^k$, we obtain:

Corollary 4.10. *The algorithm exactly identifies every closed Horn expression making $O(mst^{a+k})$ equivalence queries and $O(ms^2t^{a+k}e_t^{a+1} + m^2s^2t^{2a+3k})$ membership queries. Furthermore, the running time is polynomial in $m^2 + s^2 + t^k + t^a + e_t^a$.*

5. FULLY INEQUATED CLOSED HORN EXPRESSIONS

Clauses in this class can contain a new type of atom, that we call *inequation* or *inequality* and has the form $t \neq t'$, where both t and t' are terms. Inequated clauses may contain any number of inequalities in its antecedent. Let s be a conjunction of atoms and inequations. Then, s^p denotes the conjunction of atoms in s and $s^{\neq}$ the conjunction of inequalities in s. That is $s = s^p \wedge s^{\neq}$. We say s is *completely inequated* if $s^{\neq}$ contains all possible inequations between distinct terms in s^p, i.e., if $s^{\neq} = ineq(s^p)$. A clause $s \rightarrow b$ is completely inequated if $s = ineq(s^p \cup \{b\}) \wedge s^p$. No inequalities are allowed in the consequent. Similarly, a multi-clause $[s, c]$ is completely inequated if $s = ineq(s^p \cup c) \wedge s^p$. A *fully inequated Closed Horn expression* is a conjunction of fully inequated closed Horn clauses.

Looking at the way the transformation $U(T)$ described in Section 4.1 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. As an example, take the clause $human(father(x)) \wedge human(mother(x)) \rightarrow human(x)$. The intended meaning is clearly that $x \neq faher(x) \neq mother(x)$, and hence this clause is fully inequated. We will see that the learning algorithm described in Section 3 has to be slightly modified in order to achieve learnability of this class.

The first modification is in the minimisation procedure. It can be the case that after generalising or dropping some terms (as 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 pairing procedure. Given a matching σ and two multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$, its pairing $[s, c]$ is computed in the new algorithm as:

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

Notice that inequations in the original multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$ are ignored. The pairing is computed only for the atomic information, and finally the fully inequated pairing is constructed by adding all the inequations needed. This can be done safely because the algorithm only deals with fully inequated clauses. The proof of correctness is very similar to the one presented here. Complete details and proof for the case of Range Restricted Horn Expressions can be found in [1].

THEOREM 5.1. *The modified algorithm identifies fully inequated closed Horn expressions making $O(mst^a)$ calls to the equivalence oracle and $O(ms^2t^ae_t^{a+1} + m^2s^2t^{2a+k})$ to the membership oracle. Furthermore, the running time is polynomial in $m^2 + s^2 + t^k + t^a + e_t^a$.*

Let the class $FIRRHE$ be the class of fully inequated range restricted Horn expressions, $FICOHE$ the class of fully inequated constrained Horn expressions and $FIRRCOHE$ their union. We conclude that the classes $FIRRHE$, $FICOHE$ and $FIRRCOHE$ are learnable using the modified algorithm.

6. CONCLUSIONS

The paper introduced a new algorithm for learning closed Horn expressions (CHE) and established the learnability of fully inequated closed Horn expressions ($FICHE$). The structure of the algorithm is similar to previous ones, but it uses carefully chosen operations that take advantage of the structure of functional terms in examples. This in turn leads to an improvement of worst case bounds on the number of queries required, which is one of the main contributions of the paper. The following table contains the results obtained in [11] for range restricted Horn Expressions ($RRHE$) and in this paper for Closed Horn Expressions. This paper extends [2] where similar bounds were obtained for $RRHE$.

	Class	$EntEQ$	$EntMQ$
Result in [11]	$RRHE$	$O(mst^{t+a})$	$O(ms^2t^{t+a}e_t^{a+1} + m^2s^2t^{3t+2a})$
Our result	CHE	$O(mst^{k+a})$	$O(ms^2t^{k+a}e_t^{a+1} + m^2s^2t^{3k+2a})$
Our result	$FICHE$	$O(mst^a)$	$O(ms^2t^ae_t^{a+1} + m^2s^2t^{k+2a})$

Notice that we significantly improve previous results by removing the exponential dependence of the number of queries on the number of terms. However, we still remain exponential on the number of variables. The bounds are further improved for the case of $FICHE$. This may be significant as in many cases, while inequalities are not explicitly written, the intention is that different terms denote different objects.

The reduction in the number of queries goes beyond worst case bounds. The restriction that pairings are both basic *and* agree with the lgg table is quite strong and reduces the number of pairings and hence queries. This is not reflected in our analysis but we believe it will make a difference in practice. Similarly, the bound $m' \leq mt^k$ on $|U(T)|$ is quite loose, as a large proportion of partitions will be discarded if T includes functional structure.

Another important difference is that the proof in [11] assumes that the number of function symbols is finite. Our proof holds even when the set of function symbols is infinite or unknown, as long as examples have finite descriptions.

It is interesting to compare this result to other similar efforts in [9, 21, 3, 20, 19]. The results in [9, 21] rely on the fact that no chaining or self-resolution is possible between rules. Thus subsumption and implication are the same and it is easy to know which examples to combine in the generalisation process. The results in [3, 20] allow recursion and chaining but assume the expressions are acyclic in terms of chaining order, and that an additional query is allowed which indicates this order; in addition [3] assumes constrained expressions and [20] assumes range restricted expressions. So both results are covered by our algorithm as special cases. On the other hand their complexity is lower than in our case. In particular they are polynomial in the number of variables whereas our algorithm is exponential. It would be interesting to find out whether such reduced complexity is possible without the use of additional query types. One way to explore this question is to study the query complexity of the problem (ignoring computational complexity) by using the notion of certificates [8, 7]. The result in [19] goes beyond constrained clauses by allowing additional length bounded terms in clause bodies, but uses "subsumption-queries" to decide how to combine examples. If we allow such terms in our setting we must include them in the intermediate term set currently captured by the set $Atoms_P([s,c])$. Unfortunately, several crucial steps in our proof require that this set does not use additional terms. It remains to be seen whether such a generalisation is possible.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewer whose comments helped improve our complexity results.

REFERENCES

1. 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.
2. M. Arias and R. Khardon. A new algorithm for learning range restricted Horn expressions. In *Proceedings of the 10th International Conference on Inductive Logic Programming*, pages 21–39. Springer-Verlag, 2000. LNAI 1866.
3. Hiroki Arimura. Learning acyclic first-order Horn sentences from entailment. In *Proceedings of the International Conference on ALT*, Sendai, Japan, 1997. Springer-Verlag. LNAI 1316.
4. W. Cohen. PAC-learning recursive logic programs: Efficient algorithms. *Journal of Artificial Intelligence Research*, 2:501–539, 1995.
5. W. Cohen. PAC-learning recursive logic programs: Negative results. *Journal of Artificial Intelligence Research*, 2:541–573, 1995.
6. 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.
7. T. Hegedus. On generalized teaching dimensions and the query complexity of learning. In *Proceedings of the 8th Annual Conference on Computational Learning Theory (COLT'95)*, pages 108–117, New York, NY, USA, July 1995. ACM Press.
8. Lisa Hellerstein, Krishnan Pillaipakkamnatt, Vijay Raghavan, and Dawn Wilkins. How many queries are needed to learn? *Journal of the ACM*, 43(5):840–862, September 1996.
9. Charles David Page Jr. Anti-unification in constraint logics: Foundations and applications to learnability in first-order logic, to speed-up learning, and to deduction. Technical Report UIUCDCS-R-93-1820, University of Illinois at Urbana-Champaign, Department of Computer Science, 1993.
10. R. Khardon. Learning function free Horn expressions. *Machine Learning*, 37:241–275, 1999.
11. 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.
12. Roni Khardon. Learning Horn expressions with LogAn-H. In *Proceedings of the International Conference on Machine Learning*, pages 471–478, 2000.
13. J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987.
14. S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.
15. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19 & 20:629–680, May 1994.
16. S. Nienhuys-Cheng and R. De Wolf. *Foundations of Inductive Logic Programming.* Springer-verlag, 1997. LNAI 1228.
17. G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.
18. L. De Raedt and M. Bruynooghe. An overview of the interactive concept-learner and theory revisor CLINT. In S. Muggleton, editor, *Inductive Logic Programming*, pages 163–192. Academic Press, 1992.

19. 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.

20. 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.

21. C. Reddy and P. Tadepalli. Learning Horn definitions: Theory and an application to planning. *New Generation Computing*, 17:77–98, 1999.

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

23. E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.