J. Symbolic Computation (1987) **4**, 233–254

Fast Algorithms for Testing Unsatisfiability of Ground Horn Clauses with Equations

JEAN H. GALLIER

Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, U.S.A.

(*Received* 26 *April* 1986)

This paper presents two fast algorithms for testing the unsatisfiability of a set of ground Horn clauses with or without equational atomic formulae. If the length of the set H of Horn clauses (viewed as the string obtained by concatenating the clauses in H) is n, it is possible to design an algorithm running in time $O(n \log(n))$. These algorithms are obtained by generalising the concept of *congruence closure* to ground Horn clauses. The basic idea behind these algorithms is that the congruence closure induced by a set of ground Horn clauses can be obtained by interleaving steps in which an equational congruence closure is computed, and steps in which an implicational type of closure is computed.

1. Introduction

This paper presents two fast algorithms for testing the unsatisfiability of a set of ground Horn clauses with or without equational atomic formulae. If the length of the set of Horn clauses (viewed as the string obtained by concatenating the clauses in H) is n, then algorithm 1 runs in time $O(n^2)$ and storage $O(n)$, and algorithm 2 runs in time $O(n \log^2(n)/\log(k))$ and storage $O(kn)$, for any k chosen in advance. However, after implementing these algorithms, as Nelson and Oppen (1980), we found that in practice, algorithm 1 runs faster than algorithm 2 (the same observation has also been reported to the author by Dexter Kozen). These algorithms are obtained by combining the methods used in two other algorithms:

(1) The linear-time algorithm of Dowling and Gallier (1984) for testing the satisfiability of a set of propositional Horn clauses.
(2) The congruence closure algorithms of Kozen (1976, 1977*a*), Nelson and Oppen (1980) and Downey *et al.* (1980).

The crucial idea is that the concept of a *congruence closure* can be generalised to sets of ground Horn clauses. In this generalisation, two graphs are used. The first graph $GT(H)$, similar to the graph used in the congruence closure method (Kozen, 1976, 1977*a*; Nelson and Oppen, 1980; Oppen, 1980) represents subterm dependencies. As in Gallier (1986), an extra node $\top$ (*the constant* **true**) is added to take care of non-equational atomic formulae. The second graph $GC(H)$ (similar to the graph used in Dowling and Gallier (1984)) represents implications induced by the clauses.

This research was partially supported by the National Science Foundation under Grant No. DCR-86-07156.

0747–7171/87/050233+22 $03.00/0

Now, a set H of ground Horn clauses induces a relation E on the set of nodes of the graph $GT(H)$ defined as follows:

For every clause in H consisting of an atomic (positive) formula B:

(1) If B is an atomic formula $Pt_1 \ldots t_n$, then $(Pt_1 \ldots t_n, \top) \in E$.
(2) If B is an equation $t_1 \doteq t_2$, then $(t_1, t_2) \in E$.

Then, a certain kind of congruence closure $\overset{*}{\leftrightarrow}_E$ of E with respect to the graph $GT(H)$ can be defined. The crucial fact about this congruence is that H is unsatisfiable iff there is some negative clause $:- A_1, \ldots, A_n \in H$, such that, for every i, $1 \leq i \leq n$, if A_i is of the form $Pt_1 \ldots t_k$, then $Pt_1 \ldots t_k \overset{*}{\leftrightarrow}_E \top$, else if A_i is of the form $t_1 \doteq t_2$, then $t_1 \overset{*}{\leftrightarrow}_E t_2$.

In order to compute this congruence closure, both graphs $GT(H)$ and $GC(H)$ are used. Roughly speaking, the graph $GT(H)$ is used to propagate congruence resulting from purely equational reasons; the graph $GC(H)$ is used to propagate congruence resulting from purely implicational reasons. The algorithms presented in this paper are obtained by interleaving an equational congruence closure algorithm and an implicational closure algorithm.

It is actually not trivial to interleave an equational congruence closure algorithm and an implicational closure algorithm and achieve an $O(n \log(n))$-time performance (the best time-complexity of equational congruence closure algorithms known so far (Downey *et al.*, 1980)). The difficulty is that every time two equivalence classes are merged (during the equational congruence closure), it is necessary to propagate the information that pairs of nodes in this new class are congruent to the implicational graph. The naïve method runs in time $O(n^2)$, but it is possible to design a propagation algorithm running in time $O(n \log(n))$ using a balancing scheme akin to Tarjan's (1975) "weighting rule". We now define the graphs $GT(H)$ and $GC(H)$ and the notion of congruence closure.

2. Congruences Associated with Sets of Horn Clauses

Let H be a set of ground Horn clauses, possibly with equational atoms. First, we make the following observation. If we view our language as a two-sorted language in which there is a special sort *bool*, a constant $\top$ interpreted as **true**, and for every structure, the domain BOOL of sort *bool* is the set of truth values {**true**, **false**}, every atomic formula $Pt_1 \ldots t_k$ is logically equivalent to the equation $(Pt_1 \ldots t_k \equiv \top)$, in the sense that $Pt_1 \ldots t_k \equiv (Pt_1 \ldots t_k \equiv \top)$ is valid.

But then, this means that $\equiv$ behaves semantically exactly as the identity relation on BOOL. Hence, we can treat $\equiv$ as the equality symbol $\doteq_{bool}$ of sort *bool*, and interpret it as the identity on BOOL.

Hence, every set H of Horn clauses is equivalent to a set H' of Horn clauses over a two-sorted language, in which every atomic formula $Pt_1 \ldots t_k$ is replaced by the equation $Pt_1 \ldots t_k \equiv \top$. In the sequel, we assume that sets of Horn clauses have been preprocessed as explained above. In fact, our method applies to any many-sorted language with a finite number of sorts, including the special sort *bool*.

2.1. THE GRAPH $GT(H)$

The graph $GT(H)$ represents subterm dependencies, and it is used to propagate congruential information. This graph was first defined by Kozen (under a different name) to study the properties of finitely presented algebras (Kozen, 1976, 1977*a*, *b*, 1981).

DEFINITION 2.1. Given a set H of ground Horn clauses over a many-sorted language, let $TERM(H)$ be the set of all subterms of terms occurring in the atomic formulae in H. Let $S(H)$ be the set of sorts of all terms in $TERM(H)$. For every sort s in $S(H)$, let $TERM(H)_s$ be the set of all terms of sort s in $TERM(H)$. Note that by the definition, each set $TERM(H)_s$ is non-empty. Let Σ be the $S(H)$-ranked alphabet consisting of all constant and function symbols occurring in $TERM(H)$. Every symbol $f \in \Sigma$ has a *rank* $\rho(f) = (s_1 \ldots s_n, s)$, where s_i is the type of the ith argument of f if f is not a constant, and $\rho(f) = (e, s)$ if f is a constant. In both cases, s is the type of f. The graph $GT(H)$ has the set $TERM(H)$ as its set of nodes, and its edges and the function Λ labelling its nodes are defined as follows:

For every node t in $TERM(H)$, if t is a constant, then $\Lambda(t) = t$, else t is of the form $fy_1 \ldots y_k$ and $\Lambda(t) = f$.

For every node t in $TERM(H)$, if t is of the form $fy_1 \ldots y_k$, then t has exactly k successors $y_1, \ldots, y_k$, else t is a constant and it is a terminal node of $GT(H)$.

Given a node $u \in TERM(H)$, if $\rho(\Lambda(u)) = (s_1 \ldots s_n, s)$, $n > 0$, then the ith successor of u is denoted by $u[i]$. For every $s \in S(H)$, let $E_s = \{(r, t) \mid r \doteq_s t \in H\}$, and let E be the $S(H)$-indexed family $(E_s)_{s \in S(H)}$.

2.2. THE GRAPH $GC(H)$

The graph GC represents implicational information, and was defined in Dowling and Gallier (1984).

DEFINITION 2.2. The nodes of the graph $GC(H)$ are the atomic formulae occurring in all clauses in the set H, plus the special nodes $\top$ and $\perp$ (where $\perp$ is the constant interpreted as **false**). The edges and the function Δ labelling the edges of $GC(H)$ are defined as follows:

For every clause C of the form $B :- A_1, \ldots, A_n$ in H, for every i, $1 \leq i \leq n$, there is an edge from B to A_i labelled with C.

For every clause N of the form $:- A_1, \ldots, A_n$ in H, for every i, $1 \leq i \leq n$, there is an edge from $\perp$ to A_i labelled with N.

For every clause C of the form B, there is one edge from B to $\top$ labelled with C.

Note that since every atomic formula B is an equation $t_1 \doteq t_2$ (where t_2 may be $\top$), every node of the graph $GC(H)$ corresponds to a unique pair of nodes in the graph $GT(H)$.

2.3. CONGRUENCE CLOSURE

The crucial concept in showing the decidability of unsatisfiability for ground equational Horn clauses is a certain kind of equivalence relation on the graph $GT(H)$ called a congruence.

DEFINITION 2.3. Given the graph $GT(H)$ associated with the set H of ground Horn clauses, an $S(H)$-indexed family R of relations R_s over $TERM(H)_s$ is a *congruence on* $GT(H)$ iff:

(1) Each R_s is an equivalence relation.
(2) For every pair $(u, v) \in TERM(H)^2$, if $\Lambda(u) = \Lambda(v)$, $\rho(\Lambda(u)) = (s_1, \ldots, s_n, s)$, and for every i, $1 \leq i \leq n$, $u[i] R_{s_i} v[i]$, then $u R_s v$.

(3) For every pair (u, v) of nodes in $TERM(H)^2$ corresponding to a node $u \doteq_s v$ in the graph $GC(H)$:
  (i) If $u \doteq_s v \in H$, then $uR_s v$.
  (ii) If $u \doteq_s v$ is the head of a clause $u \doteq_s v$: $-u_1 \doteq_{s_1} v_1, \ldots, u_n \doteq_{s_n} v_n$ in H, and for every i, $1 \leq i \leq n$, $u_i R_{s_i} v_i$, then $uR_s v$.

In particular, note that any two nodes such that $u \doteq_s v$ is a clause are congruent.

2.4. A METHOD FOR TESTING UNSATISFIABILITY

The key to the method is that the least congruence on $GT(H)$ containing E exists, and that there is an algorithm for computing it. Indeed, assume that this least congruence $\overset{*}{\leftrightarrow}_E$ containing E (called the *congruence closure of E*) exists and has been computed. Then, the following result holds.

THEOREM 2.4 (Soundness and completeness). *Let H be a set of ground Horn clauses (with equality), let* $E_s = \{(r, t) \mid r \doteq_s t \in H\}$, *and let E be the S(H)-indexed family* $(E_s)_{s \in S(H)}$. *If* $\overset{*}{\leftrightarrow}_E$ *is the congruence closure on GT(H) of E, then*

$$H \text{ is unsatisfiable iff for some clause } :-u_1 \doteq_{s_1} v_1, \ldots, u_n \doteq_{s_n} v_n \text{ in } H,$$
$$\text{for every } i,\ 1 \leq i \leq n, \text{ we have } u_i \overset{*}{\leftrightarrow}_E v_i.$$

PROOF. The proof is obtained by combining and generalising the techniques used in lemmas 10.6.2. and 10.6.4 of Gallier (1986) (with some corrections). Let $\mathscr{D}$ be the subset of H consisting of the set of definite clauses in H. Let

$$\mathscr{E} = \{r \doteq_s t \mid (r, t) \in E_s,\ s \in S(H)\}.$$

Note that $\mathscr{E} \subseteq \mathscr{D}$.

First, we show that the $S(H)$-indexed family R of relations R_s on $TERM(H)$ defined such that

$$tR_s u \quad \text{iff} \quad \mathscr{D} \models t \doteq_s u,$$

is a congruence on $GT(H)$ containing E. The details are straightforward and are left to the reader.

Since $\overset{*}{\leftrightarrow}_E$ is the least congruence on $GT(H)$ containing E, for any terms $r, t \in TERM(H)_s$,

$$\text{if } r \overset{*}{\leftrightarrow}_E t, \quad \text{then} \quad \mathscr{D} \models r \doteq_s t.$$

Then, if for some negative clause $:-u_1 \doteq_{s_1} v_1, \ldots, u_n \doteq_{s_n} v_n$ in H, we have $u_i \overset{*}{\leftrightarrow}_E v_i$ for every i, $1 \leq i \leq n$, then $\mathscr{D} \models u_1 \doteq_{s_1} v_1 \wedge \ldots \wedge u_n \doteq_{s_n} v_n$ holds, which implies that the set $\mathscr{D} \cup \{: -u_1 \doteq_{s_1} v_1, \ldots, u_n \doteq_{s_n} v_n\}$ is unsatisfiable. Consequently, H is unsatisfiable.

Conversely, assume that there is no negative clause $:-u_1 \doteq_{s_1} v_1, \ldots, u_n \doteq_{s_n} v_n$ in H such that, $u_i \overset{*}{\leftrightarrow}_E v_i$ for every i, $1 \leq i \leq n$. We shall construct a model $\mathbf{M}$ of H.

First, we make the $S(H)$-indexed family $TERM(H)$ into a many-sorted Σ-algebra $\mathbf{H}$. The difficulty involved in choosing the right algebra structure is that $\overset{*}{\leftrightarrow}_E$ must be a congruence on this algebra. This is not obvious because $TERM(H)$ is not closed under the term constructors, that is, for some terms $t_1, \ldots, t_n \in TERM(H)$ and some function symbols f, $ft_1 \ldots t_n \notin TERM(H)$. Hence, we have to be careful in defining the term value of $f_{\mathbf{H}}(t_1, \ldots, t_n)$. If there exist other terms $r_1, \ldots, r_n \in TERM(H)$ such that $fr_1 \ldots r_n \in TERM(H)$, and $t_i \overset{*}{\leftrightarrow}_E r_i$, for every i, $1 \leq i \leq n$, the value of $f_{\mathbf{H}}(t_1, \ldots, t_n)$ cannot be defined arbitrarily. If we want $\overset{*}{\leftrightarrow}_E$ to be a congruence on **H**, we must define $f_{\mathbf{H}}(t_1, \ldots, t_n)$ so that $f_{\mathbf{H}}(t_1, \ldots, t_n) \overset{*}{\leftrightarrow}_E f_{\mathbf{H}}(r_1, \ldots, r_n)$. The same difficulty exists for predicate symbols. These difficulties are overcome in the following two definitions.

For each sort $s \neq bool$ in $S(H)$, each constant t of sort s is interpreted as the term t itself. For every function symbol f in Σ of rank $(w_1 \ldots w_k, s)$, with $s \neq bool$, for every k terms $y_1, \ldots, y_k$ in $TERM(H)$, each y_i being of sort w_i, $1 \leq i \leq k$,

$$f_{\mathbf{H}}(y_1, \ldots, y_k) = \begin{cases} fy_1 \ldots y_k & \text{if } fy_1 \ldots y_k \in TERM(H)_s; \\ fz_1 \ldots z_k & \text{if } fy_1 \ldots y_k \notin TERM(H)_s \text{ and there are terms} \\ & z_1, \ldots, z_k \text{ such that, } y_i \overset{*}{\leftrightarrow}_E z_i, \text{ and} \\ & fz_1 \ldots z_k \in TERM(H)_s; \\ t_0 & \text{otherwise, where } t_0 \text{ is some arbitrary term} \\ & \text{chosen in } TERM(H)_s. \end{cases}$$

For every predicate symbol P of rank $(w_1 \ldots w_k, bool)$, for every k terms

$$y_1, \ldots, y_k \in TERM(H),$$

each y_i being of sort w_i, $1 \leq i \leq k$,

$$P_{\mathbf{H}}(y_1, \ldots, y_k) = \begin{cases} \mathbf{T} & \text{if } Py_1 \ldots y_k \in TERM(H) \text{ and } Py_1 \ldots y_k \overset{*}{\leftrightarrow}_E \top; \\ \mathbf{T} & \text{if } Py_1 \ldots y_k \notin TERM(H), \text{ there are terms } z_1, \ldots, z_k \\ & \text{such that, } y_i \overset{*}{\leftrightarrow}_E z_i,\ Pz_1 \ldots z_k \in TERM(H), \\ & \text{and } Pz_1 \ldots z_k \overset{*}{\leftrightarrow}_E \top; \\ \mathbf{F} & \text{otherwise.} \end{cases}$$

Next, we prove that $\overset{*}{\leftrightarrow}_E$ is an algebra congruence on **H**. There are two main cases:

Case 1: For every function symbol f in Σ of rank $(w_1 \ldots w_k, s)$, with $s \neq bool$, for every k pairs of terms $(y_1, z_1), \ldots, (y_k, z_k)$, with y_i, z_i of sort w_i, $1 \leq i \leq k$, if $y_i \overset{*}{\leftrightarrow}_E z_i$, then:

(i) If $fy_1 \ldots y_k$ and $fz_1 \ldots z_k$ are both in $TERM(H)$, then

$$f_{\mathbf{H}}(y_1, \ldots, y_k) = fy_1 \ldots y_k, \quad \text{and} \quad f_{\mathbf{H}}(z_1, \ldots, z_k) = fz_1 \ldots z_k,$$

and since $\overset{*}{\leftrightarrow}_E$ is a congruence on $GT(H)$, we have $fy_1 \ldots y_k \overset{*}{\leftrightarrow}_E fz_1 \ldots z_k$. Hence,

$$f_{\mathbf{H}}(y_1, \ldots, y_k) \overset{*}{\leftrightarrow}_E f_{\mathbf{H}}(z_1, \ldots, z_k).$$

(ii) $fy_1 \ldots y_k \notin TERM(H)$, or $fz_1 \ldots z_k \notin TERM(H)$, but there are some terms $z'_1, \ldots, z'_k \in TERM(H)$, such that, $y_i \overset{*}{\leftrightarrow}_E z'_i$ and $fz'_1 \ldots z'_k \in TERM(H)$. Since $y_i \overset{*}{\leftrightarrow}_E z_i$, there are also terms $z''_1, \ldots, z''_k \in TERM(H)$ such that, $z_i \overset{*}{\leftrightarrow}_E z''_i$ and $fz''_1 \ldots z''_k \in TERM(H)$. Then,

$$f_{\mathbf{H}}(y_1, \ldots, y_k) = fz'_1 \ldots z'_k \quad \text{and} \quad f_{\mathbf{H}}(z_1, \ldots, z_k) = fz''_1 \ldots z''_k.$$

Since $y_i \overset{*}{\leftrightarrow}_E z_i$, we have, $z'_i \overset{*}{\leftrightarrow}_E z''_i$, and so, $fz'_1 \ldots z'_k \overset{*}{\leftrightarrow}_E fz''_1 \ldots z''_k$, that is,

$$f_{\mathbf{H}}(y_1, \ldots, y_k) \overset{*}{\leftrightarrow}_E f_{\mathbf{H}}(z_1, \ldots, z_k).$$

(iii) If neither $fy_1 \ldots y_k$ nor $fz_1 \ldots z_k$ is in $TERM(H)$ and (ii) does not hold, then

$$f_H(y_1, \ldots, y_k) = f_{\mathbf{H}}(z_1, \ldots, z_k) = t_0$$

for some chosen term t_0 in $TERM(H)$, and we conclude using the reflexivity of $\overset{*}{\leftrightarrow}_E$.

Case 2: For every predicate symbol P of rank $(w_1, \ldots w_k, bool)$, for every k pairs of terms $(y_1, z_1), \ldots, (y_k, z_k)$, with y_i, z_i of sort w_i, $1 \leq i \leq k$, if $y_i \overset{*}{\leftrightarrow}_E z_i$, then:

(i) $Py_1 \dots y_k \in TERM(H)$ and $Pz_1 \dots z_k \in TERM(H)$. Since $y_i \overset{*}{\leftrightarrow}_E z_i$ and $\overset{*}{\leftrightarrow}_E$ is a graph congruence, $Py_1 \dots y_k \overset{*}{\leftrightarrow}_E Pz_1 \dots z_k$. Hence, $Py_1 \dots y_k \overset{*}{\leftrightarrow}_E \top$ iff $Pz_1 \dots z_k \overset{*}{\leftrightarrow}_E \top$, that is, $P_{\mathbf{H}}(y_1, \dots, y_k) = \mathbf{T}$ iff $P_{\mathbf{H}}(z_1, \dots, z_k) = \mathbf{T}$.

(ii) $Py_1 \dots y_k \notin TERM(H)$ or $Pz_1 \dots z_k \notin TERM(H)$. In this case, $P_{\mathbf{H}}(y_1, \dots, y_k) = \mathbf{T}$ implies that there are terms $z'_1, \dots, z'_k \in TERM(H)$ such that, $y_i \overset{*}{\leftrightarrow}_E z'_i$, $Pz'_1 \dots z'_k \in TERM(H)$, and $Pz'_1 \dots z'_k \overset{*}{\leftrightarrow}_E \top$. Since $y_i \overset{*}{\leftrightarrow}_E z_i$, we also have $z_i \overset{*}{\leftrightarrow}_E z'_i$. Since $Pz'_1 \dots z'_k \in TERM(H)$, and $Pz'_1 \dots z'_k \overset{*}{\leftrightarrow}_E \top$, we have, $P_{\mathbf{H}}(z_1, \dots, z_k) = \mathbf{T}$. The same argument shows that if $P_{\mathbf{H}}(z_1, \dots, z_k) = \mathbf{T}$, then $P_{\mathbf{H}}(y_1, \dots, y_k) = \mathbf{T}$. Hence, we have shown that $P_{\mathbf{H}}(y_1, \dots, y_k) = \mathbf{T}$ iff $P_{\mathbf{H}}(z_1, \dots, z_k) = \mathbf{T}$.

This concludes the proof that $\overset{*}{\leftrightarrow}_E$ is a congruence on the algebra $\mathbf{H}$.

Let $\mathbf{M}$ be the quotient of the algebra $\mathbf{H}$ by the congruence $\overset{*}{\leftrightarrow}_E$. We claim that for every term $t \in TERM(H)$, $t_{\mathbf{M}} = [t]$, the congruence class of t. This is easily shown by induction and is left as an exercise. By the definition of $\mathbf{M}$ as the quotient of $\mathbf{H}$ by $\overset{*}{\leftrightarrow}_E$, we also have the following property: For any two terms $u, v \in TERM(H)_s$,†

$$\mathbf{M} \models u \doteq_s v \quad \text{iff} \quad u \overset{*}{\leftrightarrow}_E v. \tag{$*$}$$

We now prove that $\mathbf{M}$ is a model of H.

For every clause $u \doteq_s v \in H$, we have $(u, v) \in E_s$, and since $\overset{*}{\leftrightarrow}_E$ is a congruence containing E, we have $u \overset{*}{\leftrightarrow}_E v$. But then, by $(*)$, we have $\mathbf{M} \models u \doteq_s v$.

For every clause $u \doteq_s v :- u_1 \doteq_{s_1} v_1, \dots, u_n \doteq_{s_n} v_n$ in H, if $\mathbf{M} \models u_i \doteq_{s_i} v_i$ for every i, $1 \le i \le n$, by $(*)$, we have $u_i \overset{*}{\leftrightarrow}_E v_i$ for every i, $1 \le i \le n$. Since $\overset{*}{\leftrightarrow}_E$ is a congruence on $GT(H)$, we have $u \overset{*}{\leftrightarrow}_E v$. By $(*)$, this is equivalent to $\mathbf{M} \models u \doteq_s v$. Hence,

$$\mathbf{M} \models u \doteq_s v :- u_1 \doteq_{s_1} v_1, \dots, u_n \doteq_{s_n} v_n.$$

Finally, given any negative clause $:- u_1 \doteq_{s_1} v_1, \dots, u_n \doteq_{s_n} v_n$ in H, recall that it is assumed that we cannot have $u_i \overset{*}{\leftrightarrow}_E v_i$ for every i, $1 \le i \le n$. Then, for some i, $1 \le i \le n$, u_i and v_i are not congruent modulo $\overset{*}{\leftrightarrow}_E$, and by $(*)$, this implies that $\mathbf{M} \not\models u_i \doteq_{s_i} v_i$, that is, $\mathbf{M} \models \neg u_i \doteq_{s_i} v_i$. But this implies that

$$\mathbf{M} \models :- u_1 \doteq_{s_1} v_1, \dots, u_n \doteq_{s_n} v_n.$$

Hence, $\mathbf{M}$ is a model of every clause in H. This concludes the proof. □

It is interesting to note that the soundness part of Theorem 2.4 follows from the fact that $\overset{*}{\leftrightarrow}_E$ is the *least* congruence on $GT(H)$ containing E, and that the completeness part follows from the fact that $\overset{*}{\leftrightarrow}_E$ is a graph congruence. It only remains to prove that $\overset{*}{\leftrightarrow}_E$ exists and to give an algorithm for computing it.

3. Existence of the Congruence Closure

We now prove that the congruence closure of a relation R on the graph $GT(H)$ exists. This can be done by interleaving steps in which a purely equational congruence closure is computed, and steps in which a purely implicational kind of closure is computed. The advantage of this method (even though it is not the most direct) is that it justifies the correctness of the algorithm presented in the next section, and that it can also be used for showing the completeness of an extension of SLD-resolution. However, this application will be presented elsewhere (see Gallier and Raatz, 1986).

First, we define the concept of equational congruence closure.

† One might worry that the case where $f_{\mathbf{H}}(y_1, \dots, y_k)$ is set to some arbitrary term t_0 might cause some ground equation $u \doteq v$ to be valid in $\mathbf{M}$, even though it is not true that $u \overset{*}{\leftrightarrow}_E v$. This is indeed possible, but not harmful, because $\overset{*}{\leftrightarrow}_E$ is a congruence on $\mathbf{H}$, and so, $(*)$ holds.

3.1. EQUATIONAL CONGRUENCE CLOSURE

The notion of equational congruence closure was first introduced (under a different name) by Kozen (1976, 1977*a*). In fact, Dexter Kozen (1976) appears to have given an $O(n^2)$-time algorithm solving the word problem for finitely presented algebras before everyone else. Independently, the concept of congruence closure was defined in Nelson and Oppen (1980). We have added the qualifier *equational* in order to distinguish it from the more general notion defined in section 2.3 that applies to Horn clauses.

For our purpose, we only need to consider the concept of equational closure on the graph $GT(H)$ induced by some (fixed) set H of ground Horn clauses. In the rest of this section, it is assumed that a fixed set H of ground Horn clauses is given.

DEFINITION 3.1. An $S(H)$-indexed family R of relations R_s over $TERM(H)_s$ is an *equational congruence on* $GT(H)$ iff:

(1) Each R_s is an equivalence relation.
(2) For every pair $(u, v) \in TERM(H)^2$, if $\Lambda(u) = \Lambda(v)$, $\rho(\Lambda(u)) = (s_1 \ldots s_n, s)$, and for every i, $1 \leq i \leq n$, $u[i] R_{s_i} v[i]$, then $u R_s v$.

The following lemma was first shown by Kozen (1976, 1977*a*). For the sake of completeness we present the proof given in Gallier (1986).

LEMMA 3.2. *Given any $S(H)$-indexed family R of relations on $TERM(H)$, there is a smallest equational congruence $\stackrel{*}{\cong}_R$ on the graph $GT(H)$ containing R.*

PROOF. We define the sequence R^i of $S(H)$-indexed families of relations inductively as follows: For every sort $s \in S(H)$, for every $i \geq 0$,

$$
\begin{aligned}
R_s^0 &= R_s \cup \{(u, u) \mid u \in TERM(H)_s\},\\
R_s^{i+1} &= R_s^i \cup \{(v, u) \in TERM(H)^2 \mid (u, v) \in R_s^i\}\\
&\quad \cup \{(u, w) \in TERM(H)^2 \mid \exists\, v \in TERM(H), (u, v) \in R_s^i \text{ and } (v, w) \in R_s^i\}\\
&\quad \cup \{(u, v) \in TERM(H)^2 \mid \Lambda(u) = \Lambda(v), \rho(\Lambda(u)) = (s_1 \ldots s_n, s),\\
&\qquad \text{and } u[j] R_{s_j}^i v[j],\ 1 \leq j \leq n\}.
\end{aligned}
$$

Let $(\stackrel{*}{\cong}_R)_s = \bigcup_{i \geq 0} R_s^i$.

It is easily shown by induction that every equational congruence on $GT(H)$ containing R contains every R^i, and that $\stackrel{*}{\cong}_R$ is an equational congruence on $GT(H)$. Hence, $\stackrel{*}{\cong}_R$ is the least equational congruence on $GT(H)$ containing R. □

Since the graph $GT(H)$ is finite, there must exist some integer i such that $R^i = R^{i+1}$. Hence, the equational congruence closure $\stackrel{*}{\cong}_R$ of R is computable.

We now define the concept of implicational closure.

3.2. IMPLICATIONAL CLOSURE

Let H be a set of equational ground Horn clauses.

DEFINITION 3.3. An $S(H)$-indexed family R of relations R_s over $TERM(H)_s$ is an *implicational relation on* $GT(H)$ iff:

For every pair (u, v) of nodes in $TERM(H)^2$ corresponding to a node $u \doteq_s v$ in the graph $GC(H)$:

(1) If $u \doteq_s v \in H$, then $u R_s v$.

(2) If $u \doteq_s v$ is the head of a clause $u \doteq_s v : -u_1 \doteq_{s_1} v_1, \ldots, u_n \doteq_{s_n} v_n$ in H, and for every i, $1 \leq i \leq n$, $u_i R_{s_i} v_i$, then $u R_s v$.

The following result is well known (e.g. Van Emden and Kowalski, 1976; Apt and Van Emden, 1982, p. 845), but a simple proof is worth mentioning.

LEMMA 3.4. *Given a set H of equational ground Horn clauses, given any $S(H)$-indexed family R of relations on $TERM(H)$, there is a smallest implicational relation $\stackrel{*}{\supset}_R$ on the graph $GT(H)$ containing R. The relation $\stackrel{*}{\supset}_R$ is called the* implicational closure *of R on $GT(H)$.*

PROOF. We define the sequence R^i of $S(H)$-indexed families of relations inductively as follows: For every sort $s \in S(H)$, for every $i \geq 0$,

$$R_s^0 = R_s \cup \{(u, v) \in TERM(H)^2 \mid u \doteq_s v \in H\},$$
$$R_s^{i+1} = R_s^i \cup \{(u, v) \in TERM(H)^2 \mid u \doteq_s v \text{ is a node in } GC(H), \text{ and there is some clause } u \doteq_s v : -u_1 \doteq_{s_1} v_1, \ldots, u_n \doteq_{s_n} v_n \text{ in } H, \text{ such that, } u_j R_{s_j}^i v_j,\ 1 \leq j \leq n\}.$$

Let $(\stackrel{*}{\supset}_R)_s = \bigcup_{i \geq 0} R_s^i$.

As in the previous proof, it is easily shown that $\stackrel{*}{\supset}_R$ is the implicational closure of R. □

Since $GT(H)$ is finite, there is a least integer i such that $R^i = R^{i+1}$. Hence, the implicational closure $\stackrel{*}{\supset}_R$ of R is computable.

Note that $\stackrel{*}{\supset}_R$ is not necessarily an equivalence relation, but this does not matter because we are going to interleave implicational closure steps, and equational congruence closure steps.

3.3. CONGRUENCE CLOSURE FOR HORN CLAUSES

The idea is to interleave steps in which the implicational closure is computed, and steps in which the equational congruence closure is computed.

THEOREM 3.5. *Given a set H of equational ground Horn clauses, given any $S(H)$-indexed family R of relations on $TERM(H)$, there is a smallest congruence closure $\stackrel{*}{\leftrightarrow}_R$ on the graph $GT(H)$ containing R.*

PROOF. We define the sequence R^i of $S(H)$-indexed families of relations inductively as follows: For every sort $s \in S(H)$, for every $j \geq 0$,

$$R_s^0 = R_s,$$
$$R_s^{2j+1} = \stackrel{*}{\supset}_{R_s^{2j}},$$
$$R_s^{2j+2} = \stackrel{*}{\doteqdot}_{R_s^{2j+1}}.$$

Let $(\stackrel{*}{\leftrightarrow}_R)_s = \bigcup_{i \geq 0} R_s^i$.

Since the graph $GT(H)$ is finite, there is some integer $i \geq 2$ such that $R^i = R^{i+1}$. If $i = 2j$, since $R_s^{2j+1} = \stackrel{*}{\supset}_{R_s^{2j}}$ and $j \geq 1$, then R_s^{2j} is an equational congruence, and R_s^{2j+1} is a congruence on $GT(H)$. If $i = 2j+1$, since $R_s^{2j+2} = \stackrel{*}{\doteqdot}_{R_s^{2j+1}}$ and $j \geq 1$, then R_s^{2j+1} is an implicational relation, and R_s^{2j+2} is a congruence on $GT(H)$. It can also easily be shown by induction that any congruence on $GT(H)$ containing R contains every R^i. Hence, $\stackrel{*}{\leftrightarrow}_R$ is the congruence closure of R on $GT(H)$. □

The above theorem gives a method for computing $\stackrel{*}{\leftrightarrow}_R$. However, this method is not efficient. We shall give a faster algorithm based on the equational congruence closure algorithm for ground equations and Dowling and Gallier's (1984) algorithm for computing an implicational closure.

4. Algorithm for Testing Unsatisfiability, Version 1

First, we present an algorithm using Nelson and Oppen's (1980) congruence closure algorithm, and a variation of Dowling and Gallier's (1984) bottom-up algorithm for testing the unsatisfiability of propositional Horn clauses. It is possible to do better using Downey *et al.*'s (1980) congruence closure algorithm, but this algorithm is more difficult to follow. It is given in the next section.

The basic idea is to compute the least congruence $\overset{*}{\leftrightarrow}_E$ on $GT(H)$ containing E by interleaving implicational closure steps and equational congruence closure steps, as in the proof of theorem 3.5. Roughly speaking, the algorithm works by *propagation*. The graph $GT(H)$ is used to propagate equational information as follows. For any two nodes u and v labelled with the same symbol f, if $u[1], \ldots, u[n]$ are the successors of u, and $v[1], \ldots, v[n]$ are the successors of v, if for every i, $1 \leq i \leq n$, we know that $u[i] \overset{*}{\leftrightarrow}_E v[i]$, then we must also have $u \overset{*}{\leftrightarrow}_E v$. Congruence in $GT(H)$ is also propagated by reflexivity, symmetry, and transitivity.

The graph $GC(H)$ is used to propagate implicational information as follows. For any node $u \doteq v$, if $u_1 \doteq v_1, \ldots, u_n \doteq v_n$ are the targets of all edges with source $u \doteq v$ labelled C (where C is the clause $u \doteq v; -u_1 \doteq v_1, \ldots, u_n \doteq v_n$), if for every i, $1 \leq i \leq n$, we know that $u_i \overset{*}{\leftrightarrow}_E v_i$, then we must also have $u \overset{*}{\leftrightarrow}_E v$. This type of propagation can be achieved by attaching a truth field to every node of the graph $GC(H)$, as in Dowling and Gallier (1984).

Observe that congruence propagation in $GT(H)$ may trigger implicational propagation in $GC(H)$, and conversely. Indeed, whenever two nodes u and v in $GT(H)$ such that $u \doteq v$ is a node of the graph $GC(H)$ become congruent, we can set the truth field of node $u \doteq v$ to **true**. Conversely, whenever the truth field of a node $u \doteq v$ in $GC(H)$ becomes **true**, the two terms u and v are congruent in $GT(H)$.

The algorithm is designed in such a way that two procedures cooperate to the propagation process, in an alternating fashion. Procedure *satisfiable* propagates implicational information. Procedure *closure* propagates equational information. The two procedures cooperate via two queues. Procedure *satisfiable* starts with a queue *queue* containing every node $u \doteq v$ such that $u \doteq v \in H$, and every node $u \doteq u$ such that $u \doteq u$ appears as a literal on the right-hand side of some clause in H. The truth field of each node in *queue* is also set to **true**. Procedure *closure* starts with a queue *combine* containing all pairs (u, v) such that $u \doteq v \in H$.

Procedure *satisfiable* is called first, and propagates implicational information as much as possible, in a bottom-up fashion, until *queue* becomes empty. During this phase, each new pair (u, v) such that the truth field of node $u \doteq v$ becomes **true** is added to the queue *combine*. At this point, *closure* is called, and equational information is propagated as much as possible, until *combine* becomes empty. During this phase, for every pair (u, v) corresponding to the node $u \doteq v$ in $GC(H)$ such that u and v become congruent (either due to congruence, symmetry, or transitivity), if the truth field of $u \doteq v$ is not already **true**, then $u \doteq v$ is added to *queue* and the truth field of node $u \doteq v$ is set to **true**. When *closure* terminates, if *queue* has been refilled, then we proceed with another round in *satisfiable*. Otherwise, the algorithm stops.

During a call to *satisfiable*, we may detect that the truth field of node $\perp$ becomes **true**. This means that H is unsatisfiable, and the algorithm stops. The algorithm must terminate, because every step either marks a new truth field, or makes two new nodes congruent.

The main difficulty is to make the algorithm fast. Both in *satisfiable* and *closure*, it is crucial to propagate information as soon as possible to ancestors. In *satisfiable*, this can be achieved as in Dowling and Gallier (1984) by attaching counters to the nodes. In *closure*, this can be achieved as in Downey *et al.* (1980) by using a signature table and by representing an equivalence relation by its corresponding partition. Then, the two fast procedures *UNION* and *FIND* for operating on partitions are available (see Tarjan, 1975). $UNION(R, u, v)$ combines the equivalence classes of nodes u and v into a single class of the relation R. $FIND(R, u)$ returns a unique name associated with the equivalence class of node u.

Before getting more involved with the details of these algorithms, we give an example illustrating the method. To simplify the notation, an equation of the form $Pt_1 \ldots t_n \doteq \top$ is denoted as $Pt_1 \ldots t_n$, and we will also omit sorts in equality symbols.

EXAMPLE 4.1. Consider the following set H of ground Horn clauses:

$$f^3a \doteq a :- fa \doteq fb \quad (1)$$

$$a \doteq b \quad (2)$$

$$Pa \quad (3)$$

$$f^5a \doteq a :- Qa \quad (4)$$

$$Qa :- f^3a \doteq a \quad (5)$$

$$Ra :- fa \doteq a, Pfa \quad (6)$$

$$:- Rfa \quad (7)$$

The graphs $GC(H)$ and $GT(H)$ are shown in Fig. 1.

Initially, *queue* contains $a \doteq b$ and Pa, and *combine* contains the pairs (a, b) and $(Pa, \top)$. During the first call to *satisfiable*, no truth propagation takes place, nothing is added to the queue *combine*, and *closure* is called. During this call to *closure*, fa and fb are made

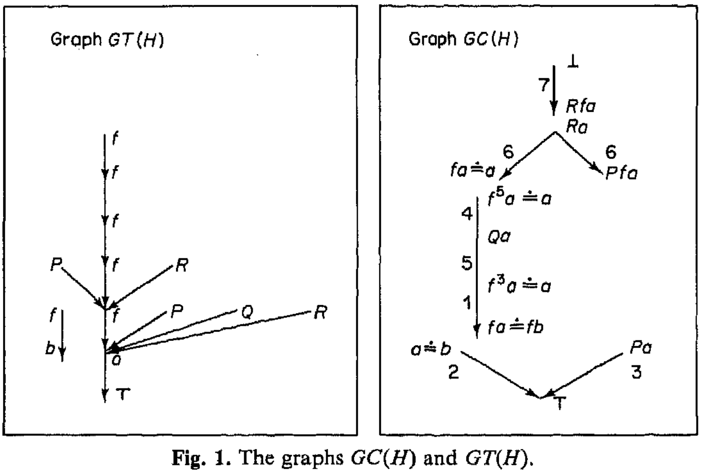

Fig. 1. The graphs $GC(H)$ and $GT(H)$.

congruent, the node $fa \doteq fb$ is placed into *queue*, and the truth field of $fa \doteq fb$ is set to **true**. Then, we proceed with another round in *satisfiable*. During this round, the truth fields of $f^3a \doteq a$, Qa, and $f^5a \doteq a$ are set to **true**, the pairs (f^3a, a), $(Qa, \top)$, and (f^5a, a) are entered into *combine*, and *closure* is called. During this call, fa, f^2a, f^3a, f^4a, and f^5a are all made congruent to a, Pfa is made congruent to $\top$, Qa is made congruent to $\top$, $fa \doteq a$ and Pfa are placed into *queue*, and the truth fields of $fa \doteq a$ and Pfa are set to **true**. In the next round in *satisfiable*, the truth field of Ra is set to **true**, the pair $(Ra, \top)$ is added to *combine*, and *closure* is called. During this call, Rfa is made congruent to $\top$, Rfa is entered into *queue*, and the truth field of Rfa is set to **true**. During the last round in *satisfiable*, the truth field of $\perp$ is set to **true**, and the algorithm stops since unsatisfiability has been detected.

For simplicity of presentation, we will explain in two separate steps how efficiency can be achieved. First, we explain how efficiency can be achieved in *satisfiable*, but for *closure*, we use the simple version of the congruence closure algorithm due to Nelson and Oppen (1980) as presented in Gallier (1984).

As in Dowling and Gallier (1984), we assign a counter *numargs*$[C]$ to every clause $C \in H$, we precompute the list *clauselist*$[n]$ for every node n of the graph $GC(H)$, and we also precompute the array *poslitlist*. For every clause $C \in H$, *numargs*$[C]$ is the number of literals on the right-hand side of :− in clause C that have the value **false**, and *poslitlist*$[C]$ is the head (left-hand side) of the clause C, if non-empty. If clause C is a negative clause, then *poslitlist*$[C]$ = **false**. For every node n in $GC(H)$, *clauselist*$[n]$ is the list of all Horn clauses in H in which n occurs as a premise (on the right-hand side of :− in C).

In order to propagate truth as quickly as possible, whenever a node n becomes **true**, for every clause C in *clauselist*$[n]$, the counter *numargs*$[C]$ is decremented by one. The crucial fact is that the node $v =$ *poslitlist*$[C]$ is ready to become **true** iff *numargs*$[C] = 0$. Hence, by propagating information to predecessor nodes using *clauselist*, we make the test for propagation very cheap.

The queue *queue* is used to traverse the graph $GC(H)$ in a bottom-up, breadth first fashion. When for some clause C, the counter *numargs*$[C]$ reaches the value 0, the node $v =$ *poslitlist*$[C]$ is entered into the queue *queue* if its truth field is currently **false**, and the truth field of v is set to **true**. When a node n is popped off *queue*, for every clause C in *clauselist*$[n]$, the counter *numargs*$[C]$ is decremented by one. Since every node is marked **true** as soon as it is entered into *queue* and only **false** nodes can be entered, each node is entered into *queue* at most once. As shown in Dowling and Gallier (1984), this algorithm runs in linear time in the number of occurrences of literals in clauses in H.

Whenever a node n in $GC(H)$ becomes **true**, since n corresponds to an equation $u \doteq v$ (where $u = left(n)$ and $v = right(n)$ for two obvious functions *left* and *right*), we check whether u and v are not congruent, and if not, the pair (u, v) is entered into the queue *combine*. Correspondingly, in *closure*, when two nodes x and y in $GT(H)$ become congruent, we need to check whether $x \doteq y$ (or $y \doteq x$) is a node L in $GC(H)$, and if this is the case and the truth field of L is **false**, we need to enter node L into *queue* and set the truth field of L to **true**. Now, we detect that x and y are congruent when a call to *UNION* is made. Indeed, two nodes become congruent either due to congruence of their respective children, symmetry, or transitivity. (In a previous incorrect version of the algorithm, we made the mistake to queue node L only when x and y became congruent due to congruence of their respective children!) The obvious solution which consists in considering each pair (u, v), where $u \in [x]$, and $v \in [y]$, and check whether either $u \doteq v$ or $v \doteq u$ is a node in $GC(H)$ is not satisfactory, because it contributes a quadratic number of steps. However, there is a way of doing this checking without increasing the time complexity of the algorithm.

When the graph $GC(H)$ is built, for every node $L = u \doteq v$, we associate two class fields, $lclass(L)$ and $rclass(L)$, such that the field $lclass(L)$ contains the name $FIND(R, u)$ of the equivalence class of u, and the field $rclass(L)$ contains the name $FIND(R, v)$, where R is the equivalence relation on the nodes of $GT(H)$. Also, when we create the graph $GT(H)$, for every node $u \in GT(H)$, we create a list (possibly empty) $classlist(u)$ of pointers, such that each pointer either points to the class field $lclass(L)$ of each node $L = u \doteq v$ in the graph $GC(H)$, or to the class field $rclass(L)$ of each node $L = v \doteq u$ in the graph $GC(H)$. Then, when the classes of x and y are merged in a $UNION(R, x, y)$ operation, we compare the sizes of the classes $[x]$ and $[y]$, and the name α of the largest of the two classes is assigned to the union of the classes. If equivalence classes are represented as trees, and the unique name associated with a class is the root element of the tree representing this class, as in the fast *UNION* and *FIND* algorithms due to Tarjan (1975), this strategy corresponds to the *weighting rule*. Then, for every node u in the smallest of the two classes, we set each class field $lclass(L)$ or $rclass(L)$ pointed to by some pointer on the list $classlist(u)$ to α. Whenever some class field of a node L is modified, we check whether $lclass(L) = rclass(L)$, that is, whether $FIND(R, u) = FIND(R, v)$. If $lclass(L) = rclass(L)$ and the truth field of L is **false**, we set the truth field of L to **true**, and place L onto *queue*.

Procedure *satisfiable* is essentially algorithm 2 of Dowling and Gallier (1984), except that atomic formulae instead of clauses are queued, and a redundant **for** loop has been eliminated. Procedure *closure* is taken from Gallier (1986).

ALGORITHM TESTING THE UNSATISFIABILITY OF A SET OF GROUND HORN CLAUSES, POSSIBLY WITH EQUATIONAL ATOMS

```
program testHorn1(infile, outfile);
  {k = number of distinct positive literals in H
  m = number of basic Horn clauses in H}
  constant nodefalse = 0;
  type clause = record
    clauseno: 1 .. maxclause;
    next: ↑clause
  end;
  type literal = record
    val: boolean;
    atom: termpair;
    clauselist: ↑clause;
    lclass, rclass: class
  end;
  type Hornclause = array[1 .. maxliteral] of literal;
  type Graph = graph of subterms as described in the text;
  type count = array[1 .. maxclause] of nodefalse .. maxliteral;
  var H: Hornclause;
      GT(H): Graph;
      R: partition;
      numargs, poslitlist: count;
```

```
    combine, queue : queuetype;
    numpos : 0 .. maxclause; {number of positive unit clauses}
    consistent : boolean;
  begin
    input Horn clause H;
    build(GT(H));
    let R = the partition corresponding to the identity relation on TERM(H);
    let combine = list of pairs of terms (v, w) such that
                either v ≐ w or w ≐ v is a positive unit clause in H;
    let queue = list of literals occurring in positive unit Horn clauses,
              and literals t ≐ t occurring on the right-hand side of some clause
              in H, and numpos their number;
    Set H[node].val := true for every node in queue;
    consistent := true;
    satisfiable(H, queue, combine, GT(H), consistent);
    if consistent then
      print("Satisfiable Horn Clause");
      printassignment
    else
      print("Unsatisfiable Horn Clause")
    endif
  end
```

Procedure *satisfiable*

```
procedure satisfiable(var H : Hornclause; var queue, combine : queuetype;
                      var GT(H) : Graph; var consistent : boolean);
  var clause1 : 1 .. maxclause;
      node, nextnode, u, v : nodefalse .. maxliteral;
  begin
      {Propagate true as long as new literals become true
      and no inconsistency}
    while queue <> nil and consistent do
      {propagate true for every clause in the clauselist for the
      positive literal node, the head of the queue}
      node := pop(queue);
      {for every clause clause1 on the clauselist for node,
      decrement the number of negative literals and check
      whether the positive literal nextnode in clause1 can be computed}
      for clause1 in H[node].clauselist do
        numargs[clause1] := numargs[clause1] - 1;
        {If all negative literals in clause1 are true and
        the positive literal is not already computed, then compute}
        if numargs[clause1] = 0 then
          nextnode := poslitlist[clause1];
          if not H[nextnode].val then
            {If nextnode is a positive literal, then set to true and enter nextnode
            into the queue. Otherwise, nextnode corresponds to false and
            H is inconsistent}
```

```
          if nextnode <> nodefalse then
            queue := push(nextnode, queue);
            H[nextnode].val := true;
            u := left(H[nextnode].atom); v := right(H[nextnode].atom);
            if FIND(R, u) ≠ FIND(R, v) then
              combine := push((u, v), combine)
            endif
          else
            consistent := false
          endif
        endif
      endif
    endfor;
    if queue = nil and consistent then
      closure(combine, queue, R)
    endif
  endwhile
end
```

The procedure *MERGE* uses the function *CONGRUENT* that determines whether two nodes u, v are congruent, and the procedure *unionupdate* that performs the union of two classes and the updating of *queue*.

Function *CONGRUENT*

```
function CONGRUENT(R: partition; u,v: node): boolean;
  var flag: boolean; i,n: integer;
  begin
    if Λ(u) = Λ(v) then
      let n = |w| where ρ(Λ(u)) = (w, s);
      flag := true;
      for i := 1 to n do
        if FIND(R,u[i]) <> FIND(R,v[i]) then
          flag := false
        endif
      endfor;
      CONGRUENT := flag
    else
      CONGRUENT := false
    endif
  end
```

Procedure *unionupdate*

```
procedure unionupdate(var R: partition; x, y: node; var queue: queuetype);
  var u: node; α,β: class; L: nodefalse..maxliteral;
  begin
    determine which of the two classes [x], [y] is the largest;
    let α be the largest class, and β be the smallest;
    UNION(R,x,y); {The union of the two classes gets the name α}
```

```
for each u ∈ β do
  for each pointer p in classlist(u) do
    if H[L].val = false then
      set the field H[L].lclass or H[L].rclass pointed to by p to α;
      if H[L].lclass = H[L].rclass then
        queue := push(L,queue);
        H[L].val := true
      endif
    endif
  endfor
endfor
end
```

Procedure *MERGE*

```
procedure MERGE(var R : partition; u,v : node; var queue : queuetype);
  var X, Y : set-of-nodes; x, y : node;
  begin
    if FIND(R,u) <> FIND(R,v) then
      X := the union of the sets P_x of predecessors of all
             nodes x in [u], the equivalence class of u;
      Y := the union of the sets P_y of predecessors of all
             nodes y in [v], the equivalence class of v;
      unionupdate(R,u,v,queue);
      for each pair (x, y) such that x ∈ X and y ∈ Y do
        if FIND(R,x) <> FIND(R,y) and CONGRUENT(R,x,y)
        then
          MERGE(R,x,y,queue);
        endif
      endfor
    endif
  end
```

Procedure *closure*

```
procedure closure(var combine, queue : queuetype; var R : partition);
  begin
    while combine ≠ nil do
      (u_i, v_i) := pop(combine);
      MERGE(R, u_i, v_i, queue)
    endwhile
  end
```

Note that each time it is called, the procedure *satisfiable* is applied to disjoint subgraphs. Since it runs in time linear in the number of occurrences of literals in the clauses corresponding to the labels of each graph, the total time complexity of *satisfiable* is linear in the number of occurrences of literals in H, which is bounded by n, the length of H considered as a string obtained by concatenating all its clauses.

Let p be the number of edges and q be the number of nodes in $GT(H)$. In Nelson and Oppen (1980) it is shown that the number of calls to *CONGRUENT* is bounded by $O(pq)$, for any sequence of calls to *MERGE*, and that the number of calls to *FIND* from *CONGRUENT* is bounded by $O(p^2)$, for any sequence of calls to *MERGE*. Now, since there are q nodes, and every call to *MERGE* increases the number of blocks of the partition, there are at most $q-1$ calls to *MERGE* altogether.

Let us find an upper bound on the total number of steps contributed by *unionupdate*. Given any call *unionupdate*$(R, x, y, queue)$, if β is the name of the smallest of the two classes $[x]$ and $[y]$, the number of steps contributed by this call is the sum of the numbers of pointers on each list *classlist*(u), over the nodes $u \in \beta$. We shall prove that the total number of steps contributed by *unionupdate* is bounded by $(2r+q)(\lfloor \log(q) \rfloor + 1)$, where q is the number of nodes in $GT(H)$, and r is the number of nodes in $GC(H)$.

First, note that since every node $L \in GC(H)$ is of the form $u \doteq v$, where u and v are nodes in $GT(H)$, if the partition R has k blocks $B_1, \ldots, B_k$, and r_i is the number of nodes in $GC(H)$ such that, for some $u \in B_i$, such a node is pointed to by some pointer in *classlist*(u), we have $\sum_{i=1}^{i=k} r_i \leq 2r$. Also, note that for every block B_i, the calls to *unionupdate* that created the block B_i can be arranged into a binary tree. We prove the following claim.

Claim. Given any block B containing Q elements, if the number of nodes in $GC(H)$ pointed to by a pointer on some *classlist* processed during some call to *unionupdate* that resulted in B is R, then the number of steps contributed by these calls is bounded by $(R+Q)(\lfloor \log(Q) \rfloor + 1)$.

Proof. We proceed by induction on Q. No merging takes place unless $Q \geq 2$. For $Q = 2$, there are at most $max\{R, 1\}$ steps, and the claim holds. For $Q > 2$, consider the top call to *unionupdate*. B is obtained by merging a block B' containing J elements and a block B'' containing $Q-J$ elements. Without loss of generality, we can assume that $2J \leq Q$, so that B' is the smallest block. For each element $u_i \in B'$, let R_i be the number of nodes in $GC(H)$ pointed to by some element in *classlist*(u_i). Hence, the number of nodes affected by the calls to *unionupdate* that formed B'' is bounded by $R - \sum_{i=1}^{i=J} R_i$. Now, using the induction hypothesis, the number of steps contributed by all calls to *unionupdate* in forming B is bounded by

$$S = \left(J + \sum_{i=1}^{i=J} R_i\right)(\lfloor \log(J) \rfloor + 1) + \left(Q - J + R - \sum_{i=1}^{i=J} R_i\right)(\lfloor \log(Q-J) \rfloor + 1) + J + \sum_{i=1}^{i=J} R_i.$$

But $\lfloor \log(J) \rfloor + 1 = \lfloor \log(2J) \rfloor$. Hence,

$$S = \left(J + \sum_{i=1}^{i=J} R_i\right)(\lfloor \log(2J) \rfloor + 1) + \left(Q - J + R - \sum_{i=1}^{i=J} R_i\right)(\lfloor \log(Q-J) \rfloor + 1).$$

Since $2J \leq Q$, we have

$$S \leq \left(J + \sum_{i=1}^{i=J} R_i\right)(\lfloor \log(Q) \rfloor + 1) + \left(Q - J + R - \sum_{i=1}^{i=J} R_i\right)(\lfloor \log(Q) \rfloor + 1)$$
$$= (R+Q)(\lfloor \log(Q) \rfloor + 1).$$

This concludes the proof of the claim. □

Let q_i be the number of elements in B_i. Using the above claim, the sum of the contributions of all calls to *unionupdate* for the k blocks is bounded by $\sum_{i=1}^{i=k} (r_i + q_i)(\lfloor \log(q_i) \rfloor + 1)$, which is bounded by

$$(\lfloor \log(q) \rfloor + 1) \sum_{i=1}^{i=k} (r_i + q_i) \leq (2r+q)(\lfloor \log(q) \rfloor + 1).$$

Since both q and r are $O(n)$, the contribution of all calls to *unionupdate* is $O(n\log(n))$.

We can show that *testHorn*1 runs in time $O(n^2)$ by the following argument. We use Nelson and Oppen's argument to show that *closure* can be implemented to run in time $O(p^2)+O(pq)$. Since the graph $GT(H)$ is obtained from all subterms occurring in atomic formulae in H, and for every term, the number of its subterms is linear in the length of the term, both p and q are linear in n, the length of H. Then, provided that constant and function symbols are encoded as integers, $GT(H)$ can be constructed in time linear in n. Even if we need to lexically analyse the constant and function symbols, the graph $GT(H)$ can be constructed in time $O(n\log(n))$ and $O(n)$ storage. As shown in Dowling and Gallier (1984), the graph $GC(H)$ can also be constructed in time linear in n if constant and function symbols are encoded as integers, or in time $O(n\log(n))$ otherwise, and $O(n)$ storage. Hence, *testHorn*1 runs in time $O(n^2)$.

In the next section, it is shown that if the congruence closure algorithm of Downey *et al.* (1980) is substituted for the previous version of *closure*, then an algorithm running in $O(n\log(n))$ is obtained.

5. Algorithm for Testing Unsatisfiability, Version 2

The main new ingredient in the fast congruence closure algorithm of Downey *et al.* (1980) is the notion of a *signature*. In order to decide quickly whether two nodes are congruent, they assign to each node u having $n>0$ successors, the tuple $sig(u)=(f, \alpha(u[1]), \ldots, \alpha(u[n]))$, where $\alpha(v)$ is an integer identifying uniquely the equivalence class of node v. Then, two nodes u, v are congruent iff $sig(u)=sig(v)$. When two nodes u, v become congruent, the signatures of all nodes having some successor in the equivalence class of either u or v need to be updated. The main trick is to precompute for every node u the list $list(u)$ of all nodes that have at least one successor in the equivalence class of u, and to use a "modify the smaller half" strategy to update signatures. When u and v become congruent, the size of $list(FIND(R, u))$ and $list(FIND(R, v))$ are compared. Then, of the two old classes, the name of the one with more predecessors is given to the new class. Thus, the only signatures that change when two classes are combined are those of nodes with a successor in the old class with fewer predecessors.

It is shown by Downey *et al.* (1980) that there is no loss of generality in restricting our attention to graphs with outdegree bounded by 2, without affecting the complexity of the algorithm. Hence, we will assume that the outdegree reduction presented in section 2.2 of their paper has been applied to $GT(H)$.

Their algorithm uses the function *UNION* and *FIND* to operate on partitions, and also the function *list*, such that $list(u)$ is the list of nodes with at least one successor in the equivalence class of u. Initially, the partition corresponds to the classes of the equivalence relation induced by E.

The procedure *congclosure* uses a second data structure, called a *signature table*, to store nodes and their signatures. Each signature is either a pair (f, α), or a triple (f, α_1, α_2), where f is (the code of) a symbol, and $\alpha, \alpha_1, \alpha_2$, are integer codes of equivalence classes. Three operations can be performed on the signature table:

enter(v): Store v with its current signature in the signature table.
delete(v): Delete v and its signature from the signature table, if present.
query(v): If some node w in the signature table has the same signature as v, then return w, otherwise **nil**.

Initially, the signature table is empty. The algorithm maintains the signature table so that any given signature appears at most once.

The algorithm also uses the set *combine*, and the queue *pending*. The queue *pending* contains a list of nodes to be entered (with their signature) in the signature table. The set *combine* contains a set of pairs of congruent nodes whose equivalence classes are to be combined.

When in *satisfiable* the truth field of a node $u \doteq v$ is set to **true**, if u and v are not already congruent, we need to merge the equivalence classes of u and v, and update the queue *pending* in preparation for the next call to *congclosure*. This is achieved by procedure *update*. Similarly, in *congclosure*, when two nodes u and v are made congruent, if $u \doteq v$ (or $v \doteq u$) is a node of $GC(H)$, the truth field of that node is set to **true**, and this node is entered into *queue* (using *unionupdate*). Otherwise, *congclosure* behaves like the algorithm in Downey *et al.* (1980).

A FAST ALGORITHM TESTING THE UNSATISFIABILITY OF A SET OF GROUND HORN CLAUSES, POSSIBLY WITH EQUATIONAL ATOMS

```
program testHorn2(infile, outfile);
  {k = number of distinct positive literals in H
  m = number of basic Horn clauses in H}
  constant nodefalse = 0;
  type clause = record
    clauseno : 1 .. maxclause;
    next : ↑clause
  end;
  type literal = record
    val : boolean;
    atom : termpair;
    clauselist : ↑clause;
    lclass, rclass : class
  end;
  type Hornclause = array[1 .. maxliteral] of literal;
  type Graph = graph of subterms as described in the text;
  type count = array[1 .. maxclause] of nodefalse .. maxliteral;
  var H : Hornclause;
      GT(H) : Graph;
      R : partition;
      numargs, poslitlist : count;
      queue : queuetype;
      numpos : 0 .. maxclause; {number of positive unit clauses}
      consistent : boolean;
   begin
      input Horn clause H;
      build(GT(H));
      let R = the partition such that two terms v, w are in the same class
              iff either v ≐ w or w ≐ v is a positive unit clause in H;
      let queue = list of literals occurring in positive unit Horn clauses,
                  and literals t ≐ t occurring on the right-hand side of some
                  clause in H, and numpos their number;
```

```
    Set H[node].val:= true for every node in queue;
    let pending = list of nodes in GT(H) that are not leaf nodes.
    consistent:= true;
    satisfiable(H,queue,pending,GT(H),consistent);
    if consistent then
      print("Satisfiable Horn Clause");
      printassignment
    else
      print("Unsatisfiable Horn Clause")
    endif
  end
```

Procedure *satisfiable*

```
procedure satisfiable(var H:Hornclause; var queue,pending:queuetype;
                      var GT(H):Graph; var consistent:boolean);
  var clause1:1..maxclause;
      node,nextnode:nodefalse..maxliteral;
  begin
      {Propagate true as long as new literals become true
      and no inconsistency}
    while queue<> nil and consistent do
      {propagate true for every clause in the clauselist for the
      positive literal node, the head of the queue}
      node:= pop(queue);
      {for every clause clause1 on the clauselist for node,
      decrement the number of negative literals and check
      whether the positive literal nextnode in clause1 can be computed}
      for clause1 in H[node].clauselist do
        numargs[clause1]:= numargs[clause1]-1;
        {If all negative literals in clause1 are true and
        the positive literal is not already computed, then compute}
        if numargs[clause1] = 0 then
          nextnode:= poslitlist[clause1];
          if not H[nextnode].val then
            {If nextnode is a positive literal, then set to true and enter nextnode
            into the queue. Otherwise, nextnode corresponds to false and
            H is inconsistent}
            if nextnode<>nodefalse then
              queue:= push(nextnode,queue);
              H[nextnode].val:= true;
              u:= left(H[nextnode].atom); v:= right(H[nextnode].atom);
              if FIND(R,u) ≠ FIND(R,v) then
                update(GT(H), pending,queue,u,v)
              endif
            else
              consistent:= false
            endif
          endif
```

```
        endif
      endfor;
      if queue = nil and consistent then
        congclosure(pending, queue, H)
      endif
    endwhile
  end
```

Procedure *update*

```
procedure update(var GT(H): Graph; var pending, queue: queuetype; v, w: term);
  var u: term;
  begin
    if |list(FIND(R,v))| < |list(FIND(R,w))| then
      for each u ∈ list(FIND(R, v)) do
        delete(u); pending := push(u, pending)
      endfor;
      unionupdate(R, w, v, queue)
    else
      for each u ∈ list(FIND(R, w)) do
        delete(u); pending := push(u, pending)
      endfor;
      unionupdate(R, v, w, queue)
    endif
  end
```

Procedure *congclosure*

```
procedure congclosure(var pending, queue: queuetype; var H: Hornclause);
  var combine: pairlist;
      u, v, w: term;
  begin
    while pending ≠ nil do
      combine := ∅;
      for each v ∈ pending do
        if query(v) = nil then
          enter(v)
        else
          combine := combine ∪ (v, query(v))
        endif
      endfor;
      pending := nil;
      for each (v, w) ∈ combine do
        if FIND(R,v)) ≠ FIND(R, w) then
          if |list(FIND(R, v))| < |list(FIND(R,w))| then
            for each u ∈ list(FIND(R,v)) do
              delete(u); pending := push(u, pending)
            endfor;
            unionupdate(R, w, v, queue)
          else
```

```
                for each u ∈ list(FIND(R,w)) do
                  delete(u); pending:= push(u,pending)
                endfor;
                unionupdate(R,v,w,queue)
              endif
            endif
          endfor
        endwhile
      end
```

The complexity analysis for *satisfiable* and *unionupdate* is unchanged. There can be at most $q-1$ *UNION* operations, since each *UNION* reduces the number of equivalence classes by one, and there are at most q classes initially (where q is the number of nodes in $GC(H)$). The number of *list* operations is bounded by a constant times the number of *UNION* operations, and is thus $O(q)$. The number of *FIND* operations is bounded by a constant times the number of additions to *pending*. Now, the reasoning used in Downey *et al.* (1980) still applies here, and shows that the number of additions to *pending* is bounded by $q+2q\log(2q)$, which is $O(q\log(q))$.

As discussed in Downey *et al.* (1980), if we use the fast *UNION* and *FIND* algorithms of Tarjan (1975), and the fast *list* operation of Downey *et al.*, the total time for *UNION* and *list* operations is $O(q)$, and the total time for *FIND* operations is $O(q\log(q))$. Since the number of additions to *pending* is $O(q\log(q))$, the dominating time factor is the time spent doing operations to the signature table (additions and deletions to *pending*). We consider the four methods given by Downey *et al.* implementing signature table operations.

(1) *Balanced binary tree.* If a balanced binary tree is used, each table operation requires $O(\log(q))$ time but $O(q)$ storage. The worst-case time bound is $O(q\log^2(q))$ and the storage required in $O(q)$. Since q is linearly related to n, the length of H, and the other algorithms require at most $O(n\log(n))$ time and $O(n)$ storage, this version of *testHorn*2 runs in time $O(n\log^2(n))$ and space $O(n)$.

(2) *Trie.* If a trie is used, for any k chosen in advance, we need $O(\log(q)/\log(k))$ time for each table operation using $O(kq)$ storage. Hence, this version of *testHorn*2 runs in time $O(n\log^2(n)/\log(k))$ and space $O(kn)$.

(3) *Array.* Using an array, each table operation requires constant time but $O(pq)$ storage. This version of *testHorn*2 runs in time $O(n\log(n))$ and space $O(n^2)$.

(4) *Hash table.* Using a hash table, each table operation takes constant time and $O(q)$ space on the average. This version of *testHorn*2 runs in time $O(n\log(n))$ and space $O(n)$ on the average.

I wish to thank Igor Faynberg and Jerzy Sliwinski for their incisive comments, and in particular for pointing out errors in a previous version of these algorithms. They have implemented and tested *testHorn*1.

References

Apt, K. R., Van Emden, M. H. (1982). Contributions to the theory of logic programming. *J. Assoc. Comp. Mach.*, **29**, (3), 841–862.

Dowling, W. F., Gallier, J. H. (1984). Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Logic Programming* **3**, 267–284.

Downey, P. J., Sethi, R., Tarjan, E. R. (1980). Variations on the common subexpressions problem. *J. Assoc. Comp. Mach.* **27**, (4), 758–771.

Gallier, J. H. (1986). *Logic for Computer Science: Foundations of Automatic Theorem Proving*. New York: Harper and Row.

Gallier, J. H., Raatz, S. (1986). Extending *SLD*-resolution to equational horn clauses using *E*-unification. *J. Logic Programming* (in press).

Kozen, D. (1976). Complexity of Finitely Presented Algebras. Technical Report TR 76-294, Department of Computer Science, Cornell University, Ithaca, NY.

Kozen, D. (1977*a*). Complexity of Finitely Presented Algebras. *9th STOC Symposium*, Boulder Colorado, 164–177, May 1977.

Kozen, D. (1977*b*). Finitely Presented Algebras and the Polynomial Time Hierarchy. Technical Report TR 77-303, Department of Computer Science, Cornell University, Ithaca, NY.

Kozen, D. (1981). Positive first-order logic is NP-complete. *IBM J. Res. Dev.* **25**, (4), 327–332.

Nelson, G., Oppen, D. C. (1980). Fast decision procedures based on congruence closure. *J. Assoc. Comp. Mach.* **27**, (2), 356–364.

Oppen, D. C. (1980). Reasoning about recursively defined data structures. *J. Assoc. Comp. Mach.* **27**, (3), 403–411.

Tarjan, E. R. (1975). Efficiency of a good but not linear set union algorithm. *J. Assoc. Comp. Mach.* **22**, (2), 215–225.

Van Emden, M. H., Kowalski, R. A. (1976). The semantics of predicate logic as a programming language. *J. Assoc. Comp. Mach.* **23**, (4), 733–742.