*Technical Report*

UCAM-CL-TR-567
ISSN 1476-2986

Number 567

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

# Using inequalities as term ordering constraints

Joe Hurd

June 2003

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

*http://www.cl.cam.ac.uk/*

Series editor: Markus Kuhn

ISSN 1476-2986

# Using inequalities as term ordering constraints

Joe Hurd*
Computer Laboratory
University of Cambridge,
`joe.hurd@cl.cam.ac.uk`

**Abstract**

In this paper we show how linear inequalities can be used to approximate Knuth-Bendix term ordering constraints, and how term operations such as substitution can be carried out on systems of inequalities. Using this representation allows an off-the-shelf linear arithmetic decision procedure to check the satisfiability of a set of ordering constraints. We present a formal description of a resolution calculus where systems of inequalities are used to constrain clauses, and implement this using the Omega test as a satisfiability checker. We give the results of an experiment over problems in the TPTP archive, comparing the practical performance of the resolution calculus with and without inherited inequality constraints.

## 1 Introduction

Term orderings have proved to be an effective way of eliminating redundancy in automated reasoning procedures, particularly in the handling of equality using ordered paramodulation [5] and unfailing completion [1]. Still more redundancy can be eliminated by allowing clauses to inherit term ordering constraints from their parents, and blocking inferences that would make the inherited constraint set inconsistent. Therefore, to apply this method we need an algorithm to decide the satisfiability of a set of ordering constraints. Two term orderings commonly implemented in automated reasoning procedures are the lexicographic path ordering and the Knuth-Bendix ordering [3]. For both orderings, deciding the satisfiability of a set of ordering constraints is an NP-complete problem [5, 4].

In this paper, we show how a Knuth-Bendix ordering constraint may be approximated by a linear equality, in such a way that a set of ordering constraints is inconsistent whenever the corresponding system of linear inequalities is inconsistent. Therefore, we can safely block inferences that would result in an inconsistent system of inequalities, without losing completeness. The main advantage of this representation of ordering constraints is that a standard linear arithmetic decision procedure can be deployed to perform the satisfiability check.

The underlying idea is very simple: the Knuth-Bendix weight of a term is a linear function of the weight of its variables, and so a Knuth-Bendix term ordering implies

*Supported by EPSRC project GR/R27105/01

a linear inequality over variable weights. The main contribution of this paper is the algorithm for generating linear inequalities from term orderings, and a suite of supporting procedures for performing substitutions on inequalities, deciding whether the solutions of one system of inequalities are a subset of the solutions of another system, etc.

In the second half of the paper, we give a formal description of a resolution calculus with inherited inequality constraints, and comment on implementation issues. We present the results of an experiment comparing the calculus with inherited constraints to the same calculus without inherited constraints. The test set for this experiment consists of the unsatisfiable problems in the TPTP archive.

The structure of the paper is as follows: in Section 2 we present the main algorithm for generating inequalities from term orderings, and the supporting procedures; in Section 3 we give a formal description of the resolution calculus we implement, paying particular attention to the part played by inequality constraints; in Section 4 we comment on implementation issues, and compare the performance of the resolution calculus with and without inherited constraints; finally in Section 5 we conclude and examine related work.

## 2 Ordering Inequalities

### 2.1 Background

Let $\Sigma$ be the signature of a first-order theory, containing the finite sets

$$R = \{r_i/m_i \mid 1 \leq i \leq N_R\} \qquad F = \{f_i/n_i \mid 1 \leq i \leq N_F\}$$

of $m_i$-place relation symbols $r_i$ and $n_i$-place function symbols $f_i$. We assume that $\Sigma$ is fixed: from now on when we talk about terms we are talking about elements of $T(\Sigma)$ (the term algebra on $\Sigma$).

Let us fix a precedence order $\ll$ defining a total order on $F$, and also a weight function $w : F \to \mathbb{N}$ mapping function symbols to non-negative integers. For all constants $c/0 \in F$, we require that $w(c/0) > 0$. Also, there must be at most one unary function symbol $f/1 \in F$ with $w(f/1) = 0$, and if one exists it must be maximal w.r.t. $\ll$. Then the weight $|t|$ of a ground term $t$ is defined by the equation

$$|f(a_1, \ldots, a_n)| = w(f/n) + |a_1| + \cdots + |a_n|$$

The Knuth-Bendix (KB) ordering $\prec$ is a binary relation on ground terms, defined as follows. If $s = f(s_1, \ldots, s_m)$ and $t = g(t_1, \ldots, t_n)$, then $s \prec t$ holds precisely when any of the following conditions are satisfied:

- $|s| < |t|$
- $|s| = |t|$ and $f/m \ll g/n$
- $|s| = |t|$ and $f/m = g/n$, and there exists $1 \leq i \leq n$ with

$$s_1 = t_1, \quad s_2 = t_2, \quad \ldots, \quad s_{i-1} = t_{i-1}, \quad s_i \prec t_i$$

We write $\preceq$ for the reflexive closure of $\prec$. The KB ordering is called uniform if the weight function satisfies $w(f/n) = 1$ for every function symbol $f/n \in F$.

Finally, we will sometimes want to consider $\preceq$ as a binary relation on both terms and atoms: this can be achieved by extending the domain of the precedence relation $\ll$ and weight function $w$ from $F$ to $F \cup R$.

## 2.2 Generating Ordering Inequalities

We begin with the proposition that term weights are linear functions.

**Proposition 1** *The weight $|t|$ of a term $t$ is a linear function of the weights of the variables in $t$.*
**Proof:** By structural induction on $t$. If $t$ is a variable $x$, then $|t| = |x|$, which is trivially a linear function of $|x|$. If $t$ is a function $f(t_1, \ldots, t_n)$, then

$$|t| = |f(t_1, \ldots, t_n)| = w(f/n) + |t_1| + \cdots + |t_n|$$

and we simply note that the sum of linear functions is a linear function. □

It is a fact that the KB ordering is a total order on ground terms, but only a partial order on terms containing variables. For example, $f(x, y) \preceq f(y, x)$ will be true whenever $x \preceq y$ is true, and of course the variables $x$ and $y$ may represent arbitrary ground terms. Therefore, if we attempt to evaluate $s \preceq t$ for arbitrary terms $s$ and $t$, there are three possible results:

| | |
|---|---|
| YES: | For all grounding substitutions $\sigma$ we have $s\sigma \preceq t\sigma$. |
| NO: | For all grounding substitutions $\sigma$ we have $s\sigma \not\preceq t\sigma$. |
| MAYBE: | There exist grounding substitutions $\sigma, \tau$ such that $s\sigma \preceq t\sigma$ and $s\tau \not\preceq t\tau$. |

The following algorithm evaluates $s \preceq t$ for arbitrary input terms $s$ and $t$, but in the case of result MAYBE, also returns a linear inequality on the weights of variables that is guaranteed to hold whenever $s \preceq t$.

1. Let $A$ denote a stack, initially empty, containing pairs of terms.
2. Push $(s, t)$ onto $A$.
3. If $A$ is empty then return YES, otherwise pop $(p, q)$ off $A$.
4. If $p = q$ then go to Step 3.
5. Calculate the linear weight functions of $p$ and $q$, and let $d := |q| - |p|$.
6. If all the coefficients of $d$ are zero then go to Step 10.
7. If all the coefficients of $d$ are positive then return YES.
8. If all the coefficients of $d$ are negative then return NO.
9. Return MAYBE with the linear inequality $0 \leq d$.

10. If $p$ is a variable then return YES.

11. If $q$ is a variable, return NO.

12. Let $f(p_1, \ldots, p_m) := p$ and $g(q_1, \ldots, q_m) := q$.

13. If $f/m = g/n$ then push $(p_n, q_n), \ldots, (p_1, q_1)$ onto $A$, and go to Step 3.

14. If $f/m \ll g/n$ then return YES, otherwise return NO.

An important feature of this algorithm is that it always returns a non-trivial inequality, not simply $|s| \leq |t|$ (which must always hold if $s \preceq t$).

**Example:** Consider applying the algorithm when $s = f(x, y)$ and $t = f(y, x)$.

We initially push $(s, t)$ onto an empty stack, and then at Step 3 pop off $p := f(x, y)$ and $q := f(y, x)$. We next calculate

$$d := |q| - |p| = (w(f/2) + |y| + |x|) - (w(f/2) + |x| + |y|) = 0$$

and since all the coefficients of $d$ are zero must jump to Step 10. We then find that $f/2 = f/2$, and so push $(y, x)$ and then $(x, y)$ onto the stack and jump back to Step 3. There, we then pop off $p := x$ and $q := y$, and calculate

$$d := |q| - |p| = |y| - |x|$$

The coefficients of $d$ are neither all zero, nor all positive, nor all negative, and so we return MAYBE together with the linear inequality $0 \leq |y| - |x|$, which is equivalent to $|x| \leq |y|$.

Of course, this is the result we expect. As previously mentioned, $f(x, y) \preceq f(y, x)$ whenever $x \preceq y$, and $x \preceq y$ means that the weight of $y$ will be at least as large as the weight of $x$. $\square$

The above example illustrated all the steps of the algorithm, with the exception of Steps 10, 11 and 14. Note that Steps 10 and 11 are necessary to compare terms of the form $f(x)$ and $x$, when $f$ is a unary function symbol with weight 0.

## 2.3 Satisfiability Checking

The KB ordering algorithm of the previous section is used to generate linear inequalities on the weights of variables, now suppose $S$ is a system of such inequalities. It is natural to ask whether there is an assignment of variable weights that will simultaneously satisfy all the variables in $S$. In fact, it is precisely this notion of satisfiability that decides whether we may perform an inference in the resolution calculus of Section 3.

We begin with the observation that the inequality variables represent term weights, and term weights are always integers. Therefore, we are only interested in whether $S$ has any integer solutions, not any real number solutions. Working with integers has several advantages, such as being able to negate inequalities without needing to introduce a strict $<$ operator, since the equivalence

$$\neg(x \leq y) \iff y < x \iff y + 1 \leq x$$

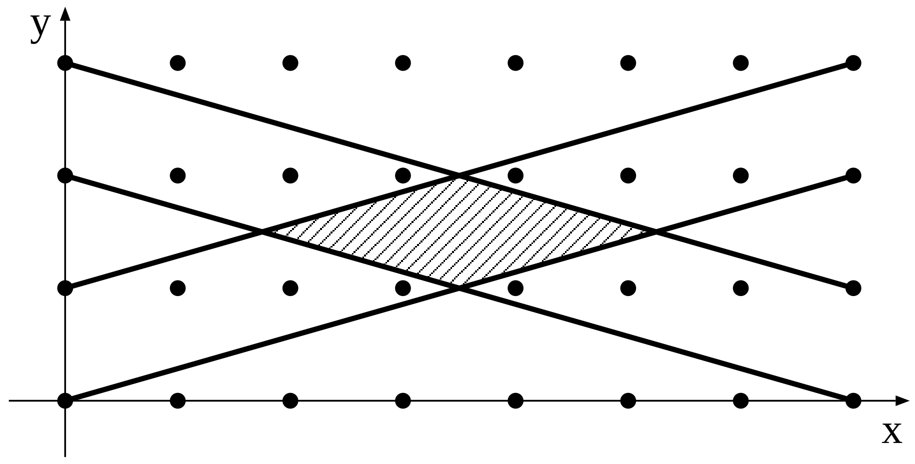

Figure 1: The system of inequalities $S_7$.

holds in the integers. Additionally, there exist systems of inequalities with solutions in reals but none in integers, and so restricting to integer solutions blocks more inferences.[1]

**Example:** Consider the system of inequalities

$$S_n \;=\; \{2x \leq ny,\ ny \leq 2x+n,\ 2n \leq 2x+ny,\ 2x+ny \leq 3n\}$$

where $n$ is an odd natural number parameter. The system $S_7$ is pictured in Figure 1. For all $n$ the area of real number solutions of $S_n$ is $(2n+1)/4$ (which tends to infinity), but for no $n$ are there any integer solutions of $S_n$. □

Although it is more costly to test for integer satisfiability than real satisfiability, there exist algorithms for this which are efficient in practice. In our experiments we use an implementation of the Omega test [8], a fast algorithm originally developed to support the static analysis of programs. We feed as input to the Omega test the current system of inequalities $S$, plus positive inequalities $1 \leq |x|$ for every variable $x$ mentioned in $S$.

## 2.4 Subset Checking

Given a system $S$ of inequalities, let $\mathrm{sol}(S)$ denote the set of integer solutions of $S$. We immediately observe that

$$\mathrm{sol}(S) = \bigcap_{A \in S} \mathrm{sol}(\{A\})$$

since a solution must satisfy all the inequalities in the system. In this section we will demonstrate how to use a satisfiability procedure to perform subset checking: evaluating $\mathrm{sol}(S) \subseteq \mathrm{sol}(T)$ for arbitrary $S$ and $T$. In Section 3, we will use this to implement a subsumption checker for clauses with inequality constraints.

[1]Of course, blocking more inferences does not necessarily speed up the search for a refutation, but at least it does reduce the total search space.

We reduce $\mathrm{sol}(S) \subseteq \mathrm{sol}(T)$ to a satisfiability problem in two stages. In the first stage we decompose $T$ into its constituent inequalities:

$$\begin{aligned} & \mathrm{sol}(S) \subseteq \mathrm{sol}(T) \\ \iff\quad & \mathrm{sol}(S) \subseteq \bigcap_{B \in T} \mathrm{sol}(\{B\}) \\ \iff\quad & \forall B \in T.\ \mathrm{sol}(S) \subseteq \mathrm{sol}(\{B\}) \end{aligned}$$

If an inequality $B$ satisfies $\mathrm{sol}(S) \subseteq \mathrm{sol}(\{B\})$ we say $B$ is redundant for $S$. In the second stage, we reduce redundancy checking to a satisfiability problem:

$$\begin{aligned} & \mathrm{sol}(S) \subseteq \mathrm{sol}(\{B\}) \\ \iff\quad & \mathrm{sol}(S) \cap \overline{\mathrm{sol}(\{B\})} = \emptyset \\ \iff\quad & \mathrm{sol}(S) \cap \mathrm{sol}(\{\neg B\}) = \emptyset \\ \iff\quad & \mathrm{sol}(S \cup \{\neg B\}) = \emptyset \\ \iff\quad & S \cup \{\neg B\} \quad \text{is inconsistent} \end{aligned}$$

Therefore, we can evaluate $\mathrm{sol}(S) \subseteq \mathrm{sol}(T)$ by checking that the system of inequalities $S \cup \{\neg B\}$ is inconsistent for every $B \in T$.

Unfortunately, although this evaluation method is exact, in practice we would like to avoid performing many costly satisfiability checks. We therefore also implement a fast but approximate method to check whether an inequality $B$ is redundant for $S$. If we assume that $S$ contains the set $P$ of positive inequalities $1 \leq |x|$ for every variable $x$ mentioned in $S$ or $B$ (this will always be the case), then we have

$$\begin{aligned} & \mathrm{sol}(S) \subseteq \mathrm{sol}(\{B\}) \\ \Longleftarrow\quad & \exists A \in S.\ \bigwedge P \wedge A \Rightarrow B \\ \iff\quad & \exists A \in S.\ A \sqsubseteq B \end{aligned}$$

We define

$$\begin{aligned} A \sqsubseteq B \quad = \quad & (\forall x.\ \mathrm{coeff}_A(x) \leq \mathrm{coeff}_B(x)) \ \wedge \\ & \mathrm{const}_A + \sum_x \mathrm{coeff}_A(x) \leq \mathrm{const}_B + \sum_x \mathrm{coeff}_B(x) \end{aligned}$$

where $\mathrm{coeff}_A(x)$ is the coefficient of $|x|$ in the inequality $A$, and $\mathrm{const}_A$ is the constant term.

**Example:** If we let $A$ be the inequality $0 \leq 2|x| + 1$, and $B$ be $0 \leq 4|x| + |y| - 1$, then we have $A \sqsubseteq B$, because

$$\begin{aligned} & \mathrm{coeff}_A(x) = 2 \leq 4 = \mathrm{coeff}_B(x) \ \wedge\ \mathrm{coeff}_A(y) = 0 \leq 1 = \mathrm{coeff}_B(y) \ \wedge \\ & \mathrm{const}_A + \sum_x \mathrm{coeff}_A(x) = 1 + 2 = 3 \leq 4 = -1 + 5 = \mathrm{const}_B + \sum_x \mathrm{coeff}_B(x) \end{aligned}$$

This example shows that our definition of $\sqsubseteq$ would be incorrect if the positive equations in $P$ were of the weaker form $0 \leq |x|$: if we set $|x|$ and $|y|$ to zero then $\bigwedge P \wedge A = \top \not\Rightarrow \bot = B$.
□

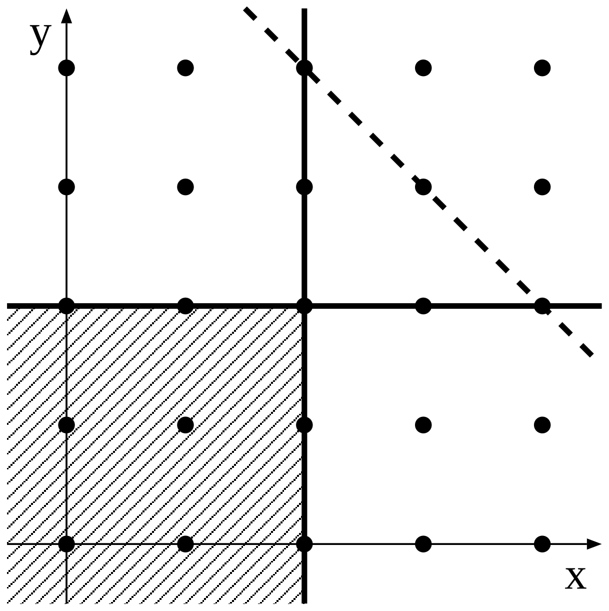

Figure 2: A system of inequalities containing a redundant equation.

We call this test the fast redundancy check, and use it to implement several operations, including subset checking. However, it is important to bear in mind that it is only an approximate test, and there are many trivial cases of redundant equations that it cannot detect.

**Example:** Consider the following system $S$ of inequalities and inequality $B$:

$$S := \{|x| \leq 2,\ |y| \leq 2\} \quad B := |x| + |y| \leq 6$$

This is pictured in Figure 2: the dotted line represents $B$. Plainly, $B$ is redundant for $S$, but there is no $A \in S$ satisfying $A \sqsubseteq B$. $\square$

## 2.5 Substitution

For clauses to be able to inherit constraint sets from their parents, we have to be able to mirror the basic clause operation of substitution on constraint sets. In principle this is easy for a system $S$ of inequalities: if a substitution[2] maps $x \mapsto t$, then replace all instances of $|x|$ in $S$ with $|t|$. Since $|t|$ is a linear function of the weights of variables in $t$, this operation maps linear inequalities to linear inequalities.

Since substitutions are so frequently applied to clauses (once or twice for every attempted inference), we would like the operation on inequality constraints to be as fast as possible. The following algorithm takes as input a substitution $\sigma$ and system $S$ of inequalities, and returns the system $S\sigma$ resulting from applying $\sigma$ to $S$.

1. Let $V$ be the set of variables mentioned in $S$.

[2]For our purposes we regard substitutions as finite maps.

2. Calculate the set $W$ of variables mentioned in $S\sigma$:

$$W = (V - FV(\mathrm{domain}\,\sigma)) \cup FV(\mathrm{range}\,\sigma)$$

3. Construct the table $T$ such that

$$\forall\, v \in V.\ \forall\, w \in W.\ T_{vw} = \mathrm{coeff}_{(0 \le |v\sigma|)}(w)$$

4. For each inequality $A \in S$, construct a new inequality $A\sigma$ where

$$\forall\, w \in W.\ \mathrm{coeff}_{A\sigma}(w) = \sum_{v \in V} \mathrm{coeff}_A(v) T_{vw}$$

5. Return the system $S\sigma$ consisting of all the new inequalities $A\sigma$.

For the puposes of this algorithm, we consider the constant term $c$ in an inequality to be a 'variable' 1 having coefficient $c$. The operation $FV(S)$ returns the variables in the set $S$ of terms.

## 2.6 Merging

The final operation on sets of constraints that we need to support is to merge two sets. This occurs when a clause is derived from more than one parent, inheriting a set of constraints from each of them. Of course, we can simply take the union of the constraint sets, but for systems of inequalities we prefer to eliminate a bit more redundancy than simple duplicates of inequalities. The following algorithm takes as input systems $S$ and $T$ of inequalities, and returns a system $U$ satisfying $\mathrm{sol}(U) = \mathrm{sol}(S \cup T)$.

1. Let $S'$ be the inequalities in $S$ that are not redundant for $T$.

2. Let $T'$ be the inequalities in $T$ that are not redundant for $S'$.

3. Return $S' \cup T'$.

# 3 Resolution Calculus

In this section we present a sound and refutation complete procedure for clausal first-order logic. The soundness follows immediately: even ignoring constraints all derived clauses are logical consequences of the initial clauses. The refutation completeness follows because the calculus is a more conservative (i.e., weaker constraints) version of the paramodulation calculus of Nieuwenhuis and Rubio [6].

We use the notation $C \parallel S$ to represent a clause $C$ constrained with the system $S$ of inequalities. The formal semantics of a constrained clause is a set of ground clauses:

$$\mathrm{Meaning}(C \parallel S) = \{C\sigma \mid \sigma \text{ is a grounding substitution } \wedge\ S\sigma \text{ holds}\}$$

We begin the resolution calculus with a set of constrained clauses $C \parallel \emptyset$, where $C$ is either an initial clause or the reflexivity axiom $x \simeq x$.

## 3.1 Inference Rules

To make the most of the inherited ordering constraints, we will use ordered resolution, factoring and paramodulation inference rules. To begin with, here is the resolution rule:

$$\frac{C \vee A \parallel S \qquad D \vee \neg B \parallel T}{C\sigma \vee D\sigma \parallel S\sigma \cup T\sigma \cup OC}$$

where $\sigma = \mathrm{mgu}(A, B)$. This inference rule is blocked unless $X\sigma \preceq A\sigma$ is satisfiable for every atom $X$ in $C \cup D$, and OC is the system of inequalities generated by these term orderings.

Here is the factoring rule:

$$\frac{C \vee A \vee B \parallel S}{C\sigma \vee A\sigma \parallel S\sigma \cup OC}$$

where $\sigma = \mathrm{mgu}(A, B)$. This inference rule is blocked unless $X\sigma \preceq A\sigma$ is satisfiable for every atom $X$ in $C$, and OC is the system of inequalities generated by these term orderings.

Finally, here is the paramodulation rule:

$$\frac{C \vee s \simeq t \parallel S \qquad D \vee A \parallel T}{C\sigma \vee D\sigma \vee A[t]_p\sigma \parallel S\sigma \cup T\sigma \cup OC}$$

where $\sigma = \mathrm{mgu}(s, A|_p)$. This inference rule is blocked unless: $s\sigma \neq t\sigma$; $t\sigma \preceq s\sigma$ is satisfiable; $X\sigma \preceq s\sigma$ is satisfiable for every atom $X$ in $C$; and $X\sigma \preceq A\sigma$ is satisfiable for every atom $X$ in $D$. As usual, OC is the system of inequalities generated by all these term orderings.

For all three inference rules, we test the resulting constraint set for satisfiability, and block the inference if the set is inconsistent.

## 3.2 Simplification

We use the following simplification rule:

$$\frac{\mathcal{C} \ \cup \ \{s \simeq t \parallel \emptyset\} \ \cup \ \{C \vee A \parallel S\}}{\mathcal{C} \ \cup \ \{s \simeq t \parallel \emptyset\} \ \cup \ \{C \vee A[t\sigma]_p \parallel S\}}$$

where $\sigma$ is the result of matching $s$ to $A|_p$, $s\sigma \neq t\sigma$, and $S \vdash t\sigma \preceq s\sigma$ is valid.

This rule is not just a special case of paramodulation, because we remove a clause from the set of clauses and replace it with the simplified clause. As discussed by Nieuwenhuis and Rubio [5], simplification has a rather delicate relationship with inherited constraints. To ensure refutation completeness, we therefore employ the following conservative strategy: if we ever derive a clause of the form $s \simeq t \parallel S$, we immediately drop all the constraints to give the clause $s \simeq t \parallel \emptyset$.

**Example:** We can use the clause

$$x + (y + z) \simeq y + (x + z) \parallel \emptyset$$

to perform the simplification

$$\begin{array}{rl} & P(w + (v + (w + (v + 0)))) \parallel \{|v| + 1 \leq |w|\} \\ \to & P(v + (v + (w + (w + 0)))) \parallel \{|v| + 1 \leq |w|\} \end{array}$$

The clause constraint tells us that $v \preceq w$ is valid, and so $w + (v + z) \to v + (w + z)$ is a simplifying rewrite for any $z$. □

## 3.3 Subsumption

We say that a clause $C \parallel S$ subsumes a clause $D \parallel T$ if there exists a substitution $\sigma$ satisfying

$$C\sigma \subseteq D \ \wedge \ \#C \leq \#D \ \wedge \ \mathrm{sol}(T) \subseteq \mathrm{sol}(S\sigma)$$

In English: a subsuming clause is more general, has fewer literals, and has weaker constraints.

## 3.4 System

We combine these operations into a standard given-clause loop:

1. Put all the initial clauses into a waiting set $W$, and let the used set $U := \emptyset$.
2. If $W$ is empty then return SATISFIABLE, else pick a clause $C$ from $W$.
3. Simplify $C$ with simplification inferences and demodulation with unit clauses.
4. If $C$ is the empty clause then return INCONSISTENT.
5. If $C$ is subsumed by a clause in $U$ then go to Step 2.
6. Add $C$ to $U$.
7. Add to $W$ all possible inferences between $C$ and clauses in $U$.
8. Remove all clauses from $U$ that can be simplified, and add them to $W$.
9. Go to Step 2.

# 4 Results

In the previous section we presented a resolution calculus in terms of clauses inheriting sets of constraints, but it can also be applied to pure clauses. In this section we describe an experiment to decide which is empirically better on problems in the TPTP archive.

Why could inherited constraints make the search for a refutation *more* efficient? Clause constraints cause some inferences to be blocked, so reducing the total search space. In addition, the clause constraints support greater simplification of clauses, which make the techniques for redundancy elimination more effective.

Why could inherited constraints make the search for a refutation *less* efficient? Clause constraints add an extra condition to the subsumption test, making it less likely that one clause will subsume another. This may have the effect of increasing the search space, since it is possible to imagine there exist many duplicates of a clause, each with different ordering constraints.

## 4.1 Implementation

For this experiment we use the Metis theorem prover,[3] which implements the resolution calculus of Section 3 and supports inherited inequality constraints. It is written in Standard ML, and includes a tactic interface for proving subgoals in the HOL theorem prover [2].[4] Metis also supports the combination of different search strategies using time-slicing, each contributing to a common pool of unit clauses.

Testing for the integer satisfiability of systems of inequalities is dealt with by a Standard ML implementation of (part of) the Omega test, due to Michael Norrish (and ported to MLton by Ken Friis Larsen). Incidentally, this code is also used in the HOL theorem prover as part of the full Omega decision procedure for linear arithmetic. The way we use the test, it can either respond SATISFIABLE, INCONSISTENT or NO CONCLUSION. For our purposes, the (relatively rare) case of NO CONCLUSION is treated as SATISFIABLE, since it the result INCONSISTENT that is used to block inferences.

A system of inequalities is represented as a list of finite maps from variable names to integers, stored with a list of all the variables mentioned. We use a finite map representation because the inequalities tend to be fairly sparse. The operations on systems of inequalities are implemented exactly as described in Section 2.

Finally, for this experiment we use the uniform Knuth-Bendix ordering, where the weight function satisfies $w(f/n) = 1$ for every function symbol $f/n \in F$. Note that in this case the weight of a ground term is simply the number of symbols in it. The precedence relation $\ll$ between function symbols is calculated as follows: select the symbol of greatest arity, and break ties by alphabetical order on the name.

## 4.2 Performance

Our test set consists of the 3297 problems marked 'unsatisfiable' in version 2.5.0 of the TPTP problem archive. Prover **r** (for resolution) implements the resolution calculus without inherited constraints, and prover **o** (for ordering) implements the calculus with inherited constraints. Each prover runs for a maximum of 60 seconds per problem, and for each prover Table 1 displays both the number of problems solved and the number of problems gained (solved by that prover but not the other).

It is natural to ask why prover **o** did so much worse than prover **r**. Let us begin with the 1391 problems that they both solved within the time limit of 60 seconds. We now restrict our attention to the 336 problems that at least one prover took more than one second to solve, to exclude trivial problems from our analysis. On average, prover **o** took

[3]Metis is available at `http://www.cl.cam.ac.uk/~jeh1004/research/metis`.

[4]The HOL version of Metis is interpreted using Moscow ML, but the standalone version used in this experiment is compiled to native code using MLton. The difference in execution speed is about a factor of 10.

| | **r** | **o** |
|---|---|---|
| #solved | 1683 | 1402 |
| #gained | 292 | 11 |

Table 1: The first experiment with inherited constraints.

1.5 seconds longer than prover **r** on these problems. On the other hand, prover **o** needed 15.2 fewer iterations of the given-clause loop.

However, these differences are extremely small, considering the time limit was 60 seconds and the number of iterations is often in the thousands. It is more revealing to look at the problems that were gained. Here are all 11 problems that prover **o** gained, sorted by the number of seconds it took to solve them (this appears in brackets after the problem name):

```
ROB004-1(2.98)  ROB022-1(3.54)  SYN014-1(32.19) LAT019-1(34.10)
COL008-1(40.78) GRP098-1(48.54) SWC378-1(53.00) SYN621-1(54.21)
FLD070-4(57.14) SYN796-1(57.84) FLD055-3(58.92)
```

As can be seen, there were two quick successes, but most of the problems that prover **o** gained were hard-earned.

Now here are the first 12 problems that prover **r** gained, again sorted by the time it took to solve them.

```
LCL033-1(0.02) LCL362-1(0.02) LCL366-1(0.02) LCL081-1(0.02)
LCL035-1(0.02) ROB008-1(0.03) LCL397-1(0.04) LCL363-1(0.04)
LCL104-1(0.07) LCL010-1(0.07) GEO059-2(0.07) SYN037-1(0.10)
```

The difference is striking: how could prover **r** gain so many trivial problems? An investigation reveals that for many problems all the short refutations violate term ordering constraints, and so on these prover **o** must fail.

We therefore implement two strategies to reduce the effect of this weakness. Firstly, we exploit our freedom to drop ordering constraints that we do not think will pay for themselves in the search for a refutation. Consequently, we only keep inequalities for which:

**EITHER** the sum of the coefficients is negative (e.g., $0 \leq |x| - 2|y|$);

**OR** there are more coefficients with negative sign (e.g., $0 \leq 3|x| - |y| - |z|$).

This heuristic is called prover **q** (for quick). Secondly, we run the provers in parallel with a model elimination prover, which is not permitted to take more than 7 seconds of execution time out of the available 60. We do this in an attempt to prove the trivial problems that may happen to evade prover **q**. Note that we must also run prover **r** in parallel with model elimination, so that the difference between the provers is solely whether constraints are inherited or not.

Table 2 compares the performance of the two provers **rm** and **qm** on the same set of 3297 unsatisfiable problems. As can be seen, our two remedies have made a large dent on the difference between the two provers, but it is still a clear win for resolution without inherited constraints. Using model elimination was successful in reducing the number of

| | **rm** | **qm** |
|---|---|---|
| #solved | 1981 | 1899 |
| #gained | 98 | 16 |

Table 2: The second experiment with inherited constraints.

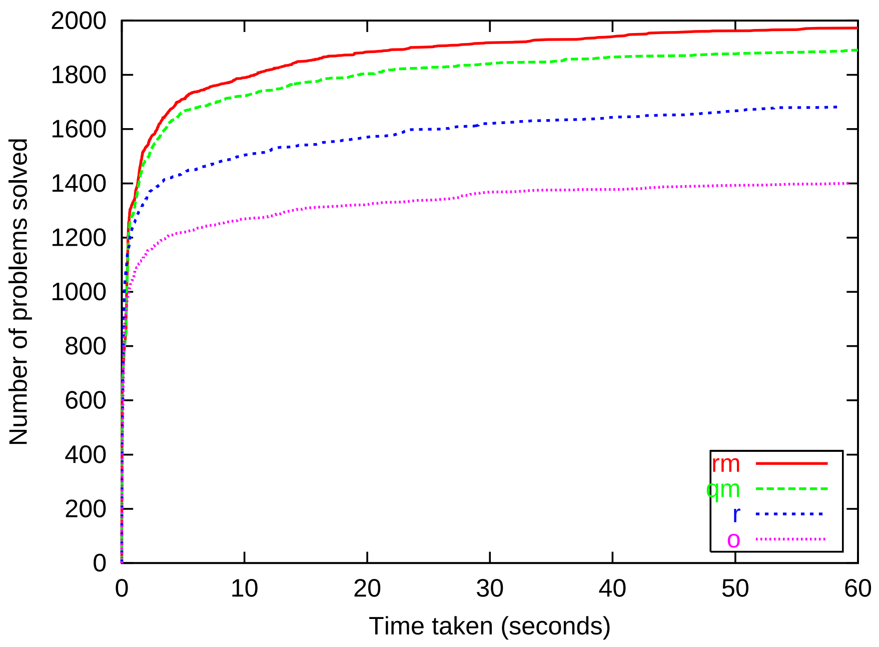

Figure 3: The success rate of the provers.

trivial problems lost due to term ordering violations in short refutations, but there were still 11 problems that prover **rm** proved in less than one second that prover **qm** couldn't solve within 60 seconds.

From the time taken on the list of problems gained by prover **o** (six out of the eleven problems took longer than 50 seconds), it may be thought that prover **o** was just getting in its stride when the time limit cut it off. Although we are currently investigating with longer time limits, at the moment we have little experimental evidence relating to performance at higher time limits. In Figure 3, for each prover we graph the number $n(x)$ of problems solved within $x$ seconds of beginning the search. All of the provers appear to have reached some kind of stability by 60 seconds.

## 5 Conclusions and Related Work

In this paper we have shown how linear inequalities can be used to approximate Knuth-Bendix ordering constraints, and how term operations such as substitution can be carried out on systems of inequalities. The main advantage of this representation is that checking the satisfiability of term ordering constraints can be carried out by an off-the-shelf linear arithmetic decision procedure.

We then implemented a resolution calculus where systems of linear inequalities were used to constrain clauses, and compared its practical performance to the same calculus without inherited constraints. Surprisingly, we found that there are many short proofs that violate term ordering constraints, and this degrades the performance of the calculus with inherited constraints. Although we were able to improve matters by relaxing the ordering constraints and introducing a helper prover, by the end the version without inherited constraints was still in front.

Of course, the ultimate goal is to develop a prover that inherits only the useful ordering constraints, and it is for this reason that we are currently investigating problems that appear to have no short refutation respecting the term ordering. One problem that was an early casualty of inherited constraints was the Steam Roller. In this case it is obvious that inheriting constraints cannot help: there are no function symbols at all in the problem, in which case the weight of every ground term will be 1 (recall that our prover uses the uniform Knuth-Bendix ordering). Further investigation will hopefully shed more light on exactly when inheriting constraints can help.

There has been a great deal of theoretical work on resolution calculi with clauses that inherit term ordering constraints: to start in this area it is hard to do better than Nieuwenhuis and Rubio's excellent survey chapter in the Handbook of Automated Reasoning [5]. This technology was implemented by Ganzinger, Nieuwenhuis and Nivela in the *Saturate* system [7], but this work appears to be more focussed on completing the saturation of a theory (where any reduction of the search space makes the procedure more efficient) than speeding up the search for a refutation (where it's not clear what the effect will be of reducing the search space, as we demonstrated).

## Acknowledgements

I would like to thank Michael Norrish for implementing the Standard ML version of the Omega test, and providing bug-fixes in a timely fashion. Ken Friis Larsen ported the program to the MLton compiler, and provided much useful advice on efficiency. Mike Gordon and the HOL developers provided valuable feedback about the HOL version of Metis, and Judita Preiss greatly improved this paper by proof-reading an early draft.

## References

[1] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[2] Joe Hurd. An LCF-style interface between HOL and first-order logic. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 134–138, Copenhagen, Denmark, July 2002. Springer.

[3] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, Elmsford, N.Y., 1970.

[4] Konstantin Korovin and Andrei Voronkov. Knuth-Bendix constraint solving is NP-complete. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 979–992, Crete, Greece, July 2001. Springer.

[5] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.

[6] Robert Nieuwenhuis and Albert Rubio. Theorem proving with ordering and equality constrained clauses. *Journal of Symbolic Computation*, 19(4):321–351, April 1995.

[7] Pilar Nivela and Robert Nieuwenhuis. Saturation of first-order (constrained) clauses with the *Saturate* system. In Claude Kirchner, editor, *Proceedings of the 5th International Conference on Rewriting Techniques and Applications (RTA-93)*, volume 690 of *Lecture Notes in Computer Science*, pages 436–440. Springer, June 1993.

[8] William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.