Toward Automating the Discovery of Decreasing Measures

Robert S. Boyer[*], Wilfred J. Legato[†] and Victor W. Marek[‡]

Abstract

An often neglected part of proof automation is simply admitting recursive function definitions into a constructive logic. Since function termination in general is undecidable, current generation theorem provers are quick to involve the human. There is, however, a substantial subset of the class of recursive functions for which termination arguments can be provided automatically. In particular, when the ordinal measure used to justify termination is less than ω^ω, we provide algorithms and proofs that guarantee optimum results, given the capability of existing proof libraries on the theorem proving system.

1 Introduction and motivation

There are at least two applications within the context of automated theorem proving for an algorithm that constructs well-founded decreasing measures. The first arises when admitting recursive functions into constructive logic provers such as NQTHM [RB79, RB98] or ACL2 [KMM00a, KMM00b]. The second arises when justifying an induction heuristic within an experimental theorem prover [Leg05] currently under development. These measures are often difficult to find, especially if the recursive function or inductive instances are mechanically generated. We

[*]Department of Computer Sciences, University of Texas, Austin, TX 78712
[†]National Security Agency, 9800 Savage Rd., Ft. Meade, MD, 20755
[‡]Department of Computer Science, University of Kentucky, Lexington, KY 40506

present here an algorithmic approach to constructing suitable ordinal measures. The method is not general, in that it only makes explicit use of ordinals less than ω^ω. Such ordinals are expressible as lists of natural numbers that are compared lexicographically. We shall see in the concluding section that our methods can be applied more broadly to in effect employ ordinals up to ϵ_0 (i.e. $\omega + \omega^\omega + \omega^{\omega^\omega} \ldots$).

Below we first introduce a statement of a formal problem (Section 1.1) and then look at the motivation of this problem in terms of automated theorem proving (Section 1.2). We then formulate (Section 1.3) our problem as a combinatorial problem. Then, in Section 2 we introduce an algorithm that finds a solution to our problem if one exists. The correctness of our solution is shown by means of a series of statements, culminating in Corollary 2.4. A sufficient condition for existence of a solution is given in Section 3. In Section 4 we provide yet another algorithm also solving the original problem but one that uses data structures more practical for programming. In the Appendix a Lisp implementation for both algorithms is provided. Section 5 contains conclusions and further research directions.

1.1 Formal Statement of the Problem

Given a set of variables $\mathcal{X} = \{x_1, x_2, ...x_n\}$ and elements of the power set of $\mathcal{X}$, $S(j, i)$ for $i = 1, 2, \ldots, n$, and $j = 1, 2, \ldots, m$, determine whether an ordering of $x_1, x_2, ...x_n$ exists such that for all i, j if $S(j, i)$ is nonempty then there exists x_k in $S(j, i)$ such that x_k precedes x_i in that ordering.

1.2 Background of the Problem

The constructive logic theorem provers NQTHM and ACL2 allow users to define a broad class of functions that include the primitive recursive functions. Both theorem provers guarantee that all functions admitted under their respective definitional principles terminate, and thus are well defined. Although many termination proofs are automatically derived, there are simple definitions such as the following where the user must identify a decreasing measure function.

$f(x, y) =$

if $x = 0$ **then**
  if $y = 0$ **then** 0 **else** $f(y-1, y-1)$
else $1 + f(x-1, y)$

When x or y is not a natural number, we define $f(x, y) = 0$. Although this function is fabricated to illustrate a point, one can easily imagine it arising as the recursive function [Mc63] for a doubly nested program loop. Such functions are commonplace when formally modeling low level programs [Leg02], and may in fact include large numbers of variables representing state components.

The correspondence between this function and the formal problem statement is

$\mathcal{X} = \{x, y\}$
$S(1,1) = \{y\}$
$S(1,2) = \emptyset$
$S(2,1) = \emptyset$
$S(2,2) = \emptyset$

The algorithms in this paper will identify the following ordinal measure function

$$\mu(x, y) = \nu(y) \cdot \omega + \nu(x)$$

where ν is a well-founded measure on the naturals.

More generally, the recursive definition of $f(x)$ within NQTHM and ACL2 generates the following proof obligations (among others).

$$h_j(x) \implies \mu(r_j(x)) < \mu(x)$$

where x represents the tuple of arguments $(x_1, x_2, \ldots x_n)$, and $h_j(x)$ is the predicate *governing* whether the recursive call $f(r_j(x))$ is made. $r_j(x)$ may in fact involve the function being defined, as is the case with the Ackermann function. The algorithms in this paper will, if successful, determine a measure function μ of the following form

$$\mu(x) = \sum_i \nu_i(x) \cdot \omega^{\pi(i)-1}$$

where ν_i is an ordinal valued function of x, and π is a permutation reflecting the ordering of $x_1, x_2, \ldots x_n$ in a solution to the formal problem statement. In practice, $\nu_i(x)$ typically will depend only on x_i and will likely be a function (e.g. COUNT within NQTHM or ACL2-COUNT within ACL2) that simply counts the number of constructors applied in generating an inductively defined object.

Given the capabilities of the theorem prover and its proof libraries

$$h_j(x) \implies \nu_i(r_j(x)) < \nu_i(x)$$

can be proved automatically for some values of i. Let G_j be the set of variables x_i for which this is true. Other ν_i can be shown to remain unchanged under the transformation r_j, and the remaining are treated as if they increased. For each "increasing" ν_i set $S(j, i) = G_j$. Set the remaining $S(j, k) = \emptyset$. If $G_j = \emptyset$, there is no solution. If a solution to the formal problem statement

$$x_{\pi(n)}, x_{\pi(n-1)}, \ldots x_{\pi(1)}$$

exists, it defines π and indirectly the measure μ. We observe that the formal problem statement guarantees that μ decreases on each recursive call, because any non decreasing ν_i is preceded in the order by a decreasing one. In this case f terminates on all inputs.

The definitional principles for both NQTHM and ACL2 will on occasion use h_j that are in fact weaker than the actual predicate governing the jth recursive call. So there will be situations where an otherwise admissible function fails to be admitted, regardless of the strength of the theorem prover, its proof libraries, or whether the techniques proposed in this paper are used.

We describe now the second application of our techniques. Rather than pattern an induction after recursive function definitions as is done in NQTHM and ACL2, the experimental theorem prover [Leg05] takes a novel approach toward induction. When presented with a clause

$$L_1(x) \vee L_2(x) \vee \ldots \vee L_t(x)$$

to be proved, it augments the set of rewrite rules with t additional *measured* rules constructed as follows. The universally quantified variables x are replaced by

pattern variables z. Then for each literal $L_i(z)$ a rewrite rule is created whose hypothesis is the conjunction of the negations of the remaining literals in the clause. $L_i(z)$ is converted to an equality (if not already so) and this equality is oriented into a replacement rule using a term ordering function compatible with the existing set of rewrite rules. When a measured rule is applied, in addition to relieving its hypothesis there is the further obligation of showing that the substitution σ_j generated by the pattern match for the jth application of a measured rule derived from this clause satisfies

$$h_j(x) \implies \mu(\sigma_j(z)) < \mu(x)$$

where h_j represents the context in which the measured rule is applied and μ is defined as before. Depending on the capabilities of the theorem prover and its proof libraries

$$h_j(x) \implies \nu_i(\sigma_j(z)) < \nu_i(x)$$

can automatically be proved for some i. We collect all variables x_i for which this is true into the set G_j. If $G_j = \emptyset$, the rule application fails. For those variables x_i that possibly increase, we set $S(j, i) = G_j$. We set the remaining $S(j, i) = \emptyset$. If the resulting measure problem can be solved, then the rule is applied. We do not need to identify the ordering. Its existence is sufficient to justify the induction.

We observe that the measure function μ evolves with each application of a measured rule derived from the same clause, since the measure must be consistent over all applications. Thus the measure problem is solved repeatedly as new rows are added to the array $S(j, i)$. It is this application that motivates the need for efficient solutions to the combinatorial problem, since the algorithm will be applied many times on measured rules derived from clauses possibly containing large numbers of state variables.

1.3 The combinatorial problem

By an assignment of sets to elements (or simply *set-assignment*) we mean a function $\mathcal{S}$ from a set X to its power set $\mathcal{P}(X)$. When X is finite, $X = \{x_1, \ldots, x_n\}$

we write such $\mathcal{S}$ as

$$\begin{pmatrix} x_1 & x_2 & \dots & x_n \\ S_1 & S_2 & \dots & S_n \end{pmatrix}$$

A set-multiassignment is a natural generalization of an assignment. Namely, we have m set-assignments $\mathcal{S}_k$, $1 \leq k \leq m$. We will store all these assignments in a single matrix:

$$\begin{pmatrix} x_1 & x_2 & \dots & x_n \\ S_{1,1} & S_{1,2} & \dots & S_{1,n} \\ \dots & \dots & \dots & \dots \\ S_{k,1} & S_{k,2} & \dots & S_{k,n} \\ \dots & \dots & \dots & \dots \\ S_{m,1} & S_{m,2} & \dots & S_{m,n} \end{pmatrix}$$

and call that matrix a *multiassignment*.

Let $\mathcal{S}$ be such a set-multiassignment on a finite set $X = \{x_1, \dots, x_n\}$. The question we want to decide is:

> ($\star$) *Is there an ordering $\prec$ of the set X such that for every k, $1 \leq k \leq m$ and for every i, $1 \leq i \leq n$, whenever $S_{k,i}$ is not empty then there is $x_j \in S_{k,i}$ such that $x_j \prec x_i$?*

Thus $\mathcal{S}$ is an instance, and $\prec$ (if it exists) is a *solution* of the problem ($\star$) for the instance $\mathcal{S}$.

The problem ($\star$), for a given instance $\mathcal{S}$, can of course, be solved by exhaustion on all $n!$ orderings of $\{x_1, \dots, x_n\}$. Our algorithm solves this problem for set-multiassignments in order $O(n^3 \cdot m)$. It either returns an ordering $\prec$, or a string $impossible$ if no such ordering exists.

Before we write the algorithm, first in English and then in pseudocode, we need some definitions.

In the text below, $\{x_1, \dots, x_n\}$ is a fixed finite set. A (total) ordering of the set $\{x_1, \dots, x_n\}$ is an irreflexive, connected, transitive relation, on the set $\{x_1, \dots, x_n\}$.

Formally, we say that an ordering $\prec$ of $\{x_1, \dots, x_n\}$ is a *solution* for the set-multiassignment $\mathcal{S}$ if $\prec$ satisfies ($\star$).

We use the term *prefix* to denote an ordering of a subset of $\{x_1, \ldots, x_n\}$. We will use a symbol $\sqsubset$ to denote prefixes. The reason for this terminology is that we can think about an ordering of $\{x_1, \ldots, x_n\}$ as a string over the alphabet $\{x_1, \ldots, x_n\}$ with no repeated symbols. An *initial segment* of an ordering $\prec$ is a subset D of $\{x_1, \ldots, x_n\}$ such that whenever $y \in D$, and $z \prec y$ then $z \in D$. The initial segments of the ordering can be identified with prefixes - those are just listings of the elements of the initial segment D according to the ordering $\prec|_D$.

When $\sqsubset$ is a prefix, we denote by $D_\sqsubset$ its domain. That is, $D_\sqsubset$ is the set of elements occurring in $\sqsubset$. Next, when $\sqsubset$ is a prefix, we say that $\sqsubset$ is *extensible* to a solution if there is an ordering $\prec$ of the entire $\{x_1, \ldots, x_n\}$ such that $\sqsubset$ is a prefix of $\prec$ and $\prec$ is a solution.

Finally, let $\sqsubset$ be a prefix. We say that an element x_i of $\{x_1, \ldots, x_n\} \setminus D_\sqsubset$ is *ready* for $\sqsubset$ if for all k, $1 \leq k \leq m$, either $S_{k,i} = \emptyset$ or $D_\sqsubset \cap S_{k,i} \neq \emptyset$.

2 Algorithm 1 and its pseudocode

Algorithm **1** belongs to the family of greedy algorithms. After initialization it selects as the shortest prefix any x_i such that for all k, $1 \leq k \leq m$, $S_{k,i} = \emptyset$. If there is no such x_i it returns the string $impossible$.

Next, at each step, it attempts to find a new element, not in the current prefix, that is ready for this prefix. If it cannot find any but there are elements not in the prefix it returns the string $impossible$. If there are elements that are not in the prefix and which are ready for that prefix, it selects one and appends it at the end of the current prefix. Finally, when it exhausts the entire set $\{x_1, \ldots, x_n\}$ without producing the string $impossible$ it returns the prefix (which is then, of course, a solution).

Here is the pseudocode for this algorithm. The symbol $\frown$ is interpreted as *concatenation* of strings, and $\langle a \rangle$ is a string consisting of a single symbol a.

Algorithm 1.
Input: A set-multiassignment on a finite set $\{x_1, \ldots, x_n\}$
Output: An ordering of $\{x_1, \ldots, x_n\}$ satisfying $(\star)$, or a string *impossible*
/∗ Initialization ∗/
(1) $\sqsubset := \emptyset$,
/∗ Basic loop ∗/
(2) **while** $(\{x_1, \ldots, x_n\} \setminus D_{\sqsubset} \neq \emptyset)$
(3) {
(4) **if** (there is no x_i ready for $\sqsubset$)
(5) {return($impossible$)};
(6) **else**
(7) {
(8) select x_i such that x_i is ready for $\sqsubset$;
(9) $\sqsubset := \sqsubset^\frown \langle x_i \rangle$;
(10) }
(11) };
(12) return ($\sqsubset$);

Lemma 2.1 *Let $\mathcal{S}$ be a set-multiassignment on $\{x_1, \ldots, x_n\}$ and $\prec$ be an ordering of $\{x_1, \ldots, x_n\}$ which solves the problem $(\star)$ for the set-multiassignment $\mathcal{S}$. Let us write the ordering $\prec$ as*

$$y_1 \prec y_2 \prec \ldots \prec y_n.$$

Then for every $j, 0 < j \leq n$, y_j is ready for the prefix $\sqsubset_j$ defined as $\prec |_{\{y_1, \ldots, y_{j-1}\}}$.

The proof follows directly from the definition of solution. □

Theorem 2.2 *Assume that $\mathcal{S}$ is a set-multiassignment and $\prec$ is a solution for the problem $(\star)$ for $\mathcal{S}$. Next, assume that $\sqsubset$ is an initial segment of $\prec$ and that $y \notin D_{\sqsubset}$, and y is ready for $\sqsubset$. Then there exists a solution of the problem $(\star)$ for $\mathcal{S}$, $\prec'$, such that $\sqsubset^\frown \langle y \rangle$ is a prefix of $\prec'$.*

Proof: Let us define the relation $\prec'$ as follows:

1. If $x, x' \in D_{\sqsubset}$ then $x \prec' x' \quad \text{iff} \quad x \prec x'$

2. If $x \in D_\sqsubset$, $x' \notin D_\sqsubset$ then $x \prec' x'$

3. If $x \notin D_\sqsubset$, $x \neq y$, then $y \prec' x$

4. If $x, x' \notin D_\sqsubset$, $x \neq y \neq x'$ then $x \prec' x' \quad \text{iff} \quad x \prec x'$.

It is easy to check that $\prec'$ is, in fact, a total ordering of $\{x_1, \ldots, x_n\}$. Now consider the ordering $\prec$. If y is the immediate successor (in $\prec$) of the last element of $D_\sqsubset$, we do nothing. Otherwise, we take y from its current position in $\prec$ and "slide" it back to the position immediately following the last element of $\sqsubset$. Calling the resulting ordering $\prec'$, we see that the order within $D_\sqsubset$ and within $(\{x_1, \ldots, x_n\} \setminus D_\sqsubset) \setminus \{y\}$ is maintained as we pass from $\prec$ to $\prec'$. The only change is that y has been moved to follow $\sqsubset$. We observe that, by construction, $\sqsubset^\frown \langle y \rangle$ is a prefix of $\prec'$.

All we need to show is that $\prec'$ is a solution for the problem $(\star)$ for the set-multi-assignment (i.e. an instance) $\mathcal{S}$. To this end, let z belong to $\{x_1, \ldots, x_n\}$. Several cases need to be considered.

1. $z \in D_\sqsubset$. Let $z = x_i$. Then, since $\prec$ is a solution, for all k, $1 \leq k \leq m$ it is the case that either $S_{k,i}$ is empty, or some element of $S_{k,i}$ $\prec$-precedes z. Say this element is s. Then, by the definition of $\prec'$, case (1), $s \prec' z$. Thus the condition for a solution is satisfied in this case.

2. $z = y$. Let $y = x_i$. Then because y is ready for $D_\sqsubset$, for each k, $1 \leq k \leq m$, $S_{k,i}$ is empty, or $S_{k,i} \cap D_\sqsubset \neq \emptyset$. By clause (2) of the definition of $\prec'$, all elements of $D_\sqsubset$ $\prec'$-precede y. Thus the condition for the solution is satisfied in this case too.

3. $z \notin D_\sqsubset$, $z \neq y$. We now have two subcases:

   (a) $z \prec y$. In this case the set of elements $\prec'$-preceding z is actually bigger than the set of the predecessors of z in $\prec$. Namely, in addition to all the $\prec$-predecessors of z it now also contains y. Now, let $z = x_i$. Then, because $\prec$ was a solution, whenever $1 \leq k \leq m$, either $S_{k,i}$ is empty, or the set of $\prec$-predecessors of z has a nonempty intersection with $S_{k,i}$. But then, since the set of $\prec'$-predecessors of z is bigger, whenever $S_{k,i} \neq \emptyset$, then the intersection of $S_{k,i}$ with the set of $\prec'$-predecessors of z is nonempty. Thus the condition for the solution is satisfied in this case.

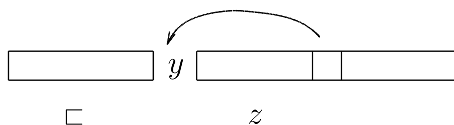

Figure 1: Moving y immediately after $\sqsubset$, Case 3(a)

(b) $y \prec z$. In this case the set of $\prec'$-predecessors of z and the set of $\prec$-predecessors coincide (the orderings $\prec'$ and $\prec$ do not coincide on that set, but the set is the same!). Thus, since $\prec$ is a solution, the condition for the solution is satisfied in this case.

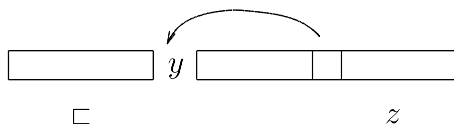

Figure 2: Moving y immediately after $\sqsubset$, Case 3(b)

This completes the argument. □

Theorem 2.2 entails the correctness of Algorithm **1**. Specifically we have:

Theorem 2.3 *Algorithm* **1** *finds a solution if there is one.*

Proof. Assume that $\mathcal{S}$ is a set-multiassignment, and $\prec$ is a solution for the problem $(\star)$ for $\mathcal{S}$. By induction on $j \leq n$ we show that after j iterations of the basic loop (2), the content of the variable $\sqsubset$ is a prefix of a solution.

This is certainly true at the initialization; the content of $\sqsubset$ is the empty sequence which is a prefix of $\prec$. Now, assume that after j iterations $\sqsubset$ holds a prefix of a solution. Assuming that $j \neq n$, since there is a solution $\prec'$ such that $\sqsubset$ is a prefix of $\prec'$, the first element of $\prec'$ that follows $\sqsubset$, say y is by Lemma 2.1 ready for $\sqsubset$. Therefore, the set of elements in $\{x_1, \ldots, x_n\}$ that are ready for $\sqsubset$ and do not belong to $D_{\sqsubset}$ is nonempty. Therefore line (4) will select *some* element y. But by Theorem 2.2 there is a solution $\prec''$ such that $\sqsubset \frown \langle y \rangle$ is a prefix of $\prec''$. Thus the

inductive step is proved. But now, when $j = n$, the prefix $\sqsubset$ must coincide with a solution, which is returned by line (12). $\Box$

Corollary 2.4 *Algorithm* **1** *finds the solution to the problem* $(\star)$ *for a set-multi-assignment* $\mathcal{S}$ *if and only if a solution exists.*

Proof: We proved that if a solution exists, one will be found by the algorithm **1**. But it is easy to see from the definition of ready elements that if a sequence $\vec{x}$ of length n is returned by the algorithm **1** then it is a solution. $\Box$

Let us look at the complexity of algorithm **1**. The test at line (4) is run at most n times and within each run for at most n sets each of size n we test m-times either the emptiness or nonemptiness of its intersection with a set consisting of at most n elements. This can be done in time $O(n^2)$, and hence algorithm **1** runs in time at most $O(n^3 \cdot m)$.

A further inspection shows, however, that we can do better, if we maintain the sets $S_{j,k}$ sorted and, additionally, we maintain a separate variable containing all elements of the prefix in sorted order. Namely, by a version of the familiar **merge-sort** algorithm, we can test if two such sets have non-empty intersection in linear time. This implies that the algorithm runs in at most $O(n^2 \cdot m)$.

Theorem 2.2 used the fact that given a solution $\prec$ and an element $y \in \{x_1, \ldots, x_n\}$, we could slide a given element y closer to the front, to *any position* as long as y is ready for the prefix of $\prec$ determining this position. This implies that given a solution $\prec$ we can define the *indicator function* – a function which assigns, to a given element y, the first place where y can be moved. Then we can move y to any position between this indicator and its current position in the solution $\prec$ and the resulting ordering will still be a solution. Formally, given an instance $\mathcal{S}$ and a solution $\prec = \langle y_1, \ldots, y_n\rangle$ and an element $y = y_k$, the *indicator* of y in $\prec$ is the least i such that $\langle y_1, \ldots, y_{i-1}, y, y_i, \ldots, y_{k-1}, y_{k+1}, \ldots, y_n\rangle$ is also a solution. We denote this value by $ind_{\mathcal{S},\prec}(y)$. It is easy to see that because $\prec$ is a solution, the function $\lambda(y) ind_{\mathcal{S},\prec}(y)$ is well defined.

If $\mathcal{S}$ is a multiassignment and $\mathcal{T}$ arises from $\mathcal{S}$ by elimination of some of its rows, then any solution $\prec$ to $\mathcal{S}$ is a solution to $\mathcal{T}$. It turns out that given such a situation there is a relationship between the corresponding indicator functions for $\mathcal{S}$ and $\mathcal{T}$.

Specifically, we have the following property.

Proposition 2.5 *If $\mathcal{S}$, $\mathcal{T}$ are two set-multiassignments and $\mathcal{T}$ arises from $\mathcal{S}$ by elimination of some rows, and $\prec$ is a solution of $(\star)$ for the instance $\mathcal{S}$, then $\prec$ is a solution of $(\star)$ for the instance $\mathcal{T}$ and the indicator function for $\mathcal{T}$ is pointwise smaller or equal than that for $\mathcal{S}$. That is, for all y, $ind_{\mathcal{T},\prec}(y) \leq ind_{\mathcal{S},\prec}(y)$.*

Proposition 2.5 tells us that if the requirements for the instance $\mathcal{T}$ are less stringent than those for $\mathcal{S}$ then we can slide elements further back in $\mathcal{T}$ and maintain the property of being a solution.

3 A sufficient condition for the case $m = 1$

Let $\mathcal{S} = \left\langle \begin{matrix} x_1 & x_2 & \dots & x_n \\ S_1 & S_2 & \dots & S_n \end{matrix} \right\rangle$ be a set assignment on the set $X = \{x_1, \dots, x_n\}$. We say that the set-assignment $\mathcal{S}$ satisfies condition $(\star\star)$

if

whenever $x_j \in S_i$ then $S_j \subset S_i$.

Let us note that the requirement is the strict inclusion.

If $n > 0$ then the condition $(\star\star)$ implies that there must be S_i such that $S_i = \emptyset$. For if $S_i \neq \emptyset$ then there is $x_j \in S_i$ such that $S_j \subset S_i$. Then either $S_j = \emptyset$ or there is $x_k \in S_j$ such that $S_k \subset S_j$. If $S_k \neq \emptyset$ we could continue. Since all sets $S_i, i \leq n$ are finite, we eventually have to reach the empty set.

Proposition 3.1 *If $\mathcal{S}$ is an assignment of sets to elements of $X = \{x_1, \dots, x_n\}$ satisfying $(\star\star)$ then there is a positive solution to the problem $(\star)$ for $\mathcal{S}$.*

Proof: Assume $\mathcal{S}$ satisfies the condition $(\star\star)$. Define a graph $G_{\mathcal{S}} = \langle X, E \rangle$ on the set X by defining the edges as follows:

$$(x_i, x_j) \in E \quad \text{if and only if} \quad S_i \subset S_j.$$

We claim that the graph $\langle X, E \rangle$ is acyclic. Indeed, if $H = \langle x_{i_1}, \dots, x_{i_t} \rangle$ is a simple cycle in $\langle X, E \rangle$, then we have

$$S_{i_1} \subset S_{i_2} \subset \dots \subset S_{i_t} \subset S_{i_1}$$

which is an obvious contradiction.

Now, knowing that $G = \langle X, E \rangle$ is an acyclic graph we can topologically sort $G_{\mathcal{S}}$. We claim that every topological sort $\prec$ of $G = \langle X, E \rangle$ has the property $(\star\star)$.

To this end, let x_j be an element of X. If $S_j = \emptyset$, there is nothing to prove.

Thus assume $S_j \neq \emptyset$. Select $x_i \in S_j$. Then, by condition $(\star\star)$, $S_i \subset S_j$, and by the definition $(x_i, x_j) \in E$. Since $\prec$ is the topological sort of $\langle \{x_1, \ldots, x_n\}, E \rangle$, $x_i \prec x_j$ as desired. $\square$

Proposition 3.2 *The condition $(\star\star)$ can be tested in time polynomial in the size of $\mathcal{S}$. Once the condition is met, the ordering $\prec$ can be found in the time linear in the size of $\mathcal{S}$.*

Proof: Assuming that the family $\mathcal{S}$ is implemented as a double linked list, it is easy to test that $\mathcal{S}$ assigns an empty set to at least one element of X. This testing can be done in time linear in the size of $\mathcal{S}$. Then, for each pair $\langle i, j \rangle$, the inclusion of $S_j \subset S_i$ can be tested in time linear in the size of X. There are at most $|X|^2$ such tests to be performed.

Finally, once the condition $(\star\star)$ has been checked, the graph $G_{\mathcal{S}}$ can be computed in time linear in the size of $\mathcal{S}$, and since the topological sort can be performed in time linear in the size of $G_{\mathcal{S}}$, we are done. $\square$

4 Algorithm 2 and its pseudocode

In this section we study an alternative algorithm in which the information is stored in two binary matrices D and U. The matrix D describes where the substitutions "go down". That is $D_{j,i} = 1$ if substitution j provably goes down on variable i and $D_{j,i} = 0$, otherwise. Likewise, $U_{j,i} = 1$ if substitution j possibly goes up on variable i, and $U_{j,i} = 0$, otherwise. We also adopt the convention that the variables are indexed starting with 1. We set $D_{j,0} = U_{j,0} = 1$ for all j. If we represent the jth rows of the arrays D and U by D_j and U_j then these bit vectors may be represented as integers using the formulas

$$D_j = \sum_i D_{j,i} \cdot 2^i$$

and

$$U_j = \sum_i U_{j,i} \cdot 2^i$$

The following algorithm decides whether there exists a measure that provably decreases over all j. If so, a measure can be constructed by successively appending the variables indexed by non-zero bits of M (see the algorithm below) to a growing prefix. Failure to find a decreasing measure, does not rule out the possibility of later finding such a measure after adding more non-zero values to D and more zero values to U. Here is the alternative algorithm.

Algorithm 2.
Input: Binary matrices D and U
Output: *true* if an ordering of $\{x_1, \ldots, x_n\}$ exists that satisfies the requirements of a decreasing ordinal measure, otherwise *false*.
/* Main Loop */
(1) $\mathcal{D} := \bigvee_j D_j, \mathcal{U} := \bigvee_j U_j$, $M := \mathcal{D} \wedge \neg\mathcal{U}$;
(2) **if** $M = 0$
(3) **then** { **if** $(\mathcal{D} \vee \mathcal{U}) = 0$
(4) **then** return (true)
(5) **else** return (false)};
(6) **else**
/* Inner loop */
(7) {**for all** j
(8) **if** $(M \wedge D_j) \neq 0$
(9) **then**
(10) $\{D_j := 0$;
(11) $U_j := 0\}$;
(12) }
(13) **go to** 1.

4.1 Analysis of Algorithm 2

A variable ordering $\langle x_1, x_2, \ldots, x_n \rangle$ gives rise to a decreasing measure provided for each j if $U(j,i) = 1$ then there exists x_k preceding x_i such that $D(j,k) = 1$. We make use of the following lemma.

Lemma 4.1 *Suppose the variable ordering specified by listing $y_1, y_2, \ldots, y_n$ gives rise to a decreasing measure. Suppose x_i is a variable for which the corresponding column in the matrix U is all 0's and the corresponding column in D has at least one 1. Suppose that $x_i = y_k$ for some $k > 1$. Then the variable ordering $x_i, y_1, \ldots, y_{k-1}, y_{k+1}, \ldots, y_n$ also gives rise to a decreasing measure.*

This follows directly from the criteria defining a decreasing measure and is quite similar to the previous argument. The case for x_i is trivial, since no $U(j, i)$ is 1. The cases for $y_1, y_2, \ldots, y_{k-1}$ follow because we have increased the number of variables preceding them. The cases for $y_{k+1}, \ldots, y_n$ remain unchanged, because they have the same sets of variables preceding them.

It is clear that the above algorithm will give rise to a decreasing measure if it terminates with no non-zero rows, since at each stage it zeroizes rows known to decrease under the partial measure constructed. It remains to be shown that if the above algorithm halts without eliminating all rows, then no decreasing measure exists. Equivalently, if a solution exists then the above algorithm will eliminate all rows. We use strong induction on the number of non-zero rows. Assume that for fewer than m non-zero rows the above algorithm finds a solution when one exists. Suppose the system D, U with m non-zero rows has a solution. Then the mask M computed at step 1 is non-zero, otherwise every variable that goes down in some substitution goes up in another and consequently no variable can be placed at the beginning of a substitution. Construct a prefix to a variable ordering by placing all variables corresponding to a 1 within M at the beginning (in some arbitrary order). From Lemma 4.1, we know that moving the variables identified by M one at a time to the beginning gives rise to a solution if one exists. Since each row containing a 1 in the prefix is guaranteed to decrease under any measure beginning with the prefix, we need no longer consider them. Zeroizing their rows simultaneously, eliminates the columns within the prefix. By the induction hypothesis, since the reduced system has a solution, continuation of the algorithm will result in all rows being zeroized.

The run time of this algorithm is proportional to the product of the number of variables times the number of substitutions. If counting each bit vector operation as separate operations on each bit, the run time is linear in the number of substitutions and quadratic in the number of variables.

5 Conclusions and further research

In this paper we have shown how to automatically derive ordinal measures to prove function termination or justify an induction heuristic. It should be noticed that our methods may in fact be applied in a more general setting. Rather than associating each ν_i with a variable, ν_i could in fact be an arbitrary ordinal valued function of x. In an extreme case, there need only be one ν_i representing the complete measure μ. In less extreme cases, variables could be grouped in very natural ways (for example components of a multiple precision integer) into a single ν_i. By supplying such functions it is possible to derive ordinal measures up to ϵ_0.

It should also be noticed that several recursive functions used in classical work, for instance those appearing in [Ge36, Go44], require the use of ordinals beyond ω^ω.

Our algorithm is fast (works in polynomial time), but perhaps a less efficient but more general algorithm would better suit the application. For example, it is possible to strengthen the hypotheses under which we prove that a variable decreases by assuming that those variables in the current prefix neither increase nor decrease. Such strengthening obviously entails a computational cost, either up front or as part of the measure determining algorithm. We hope that stronger results (i.e. more general, but still easily checkable conditions) will be found, and even more importantly, introduced into automated theorem proving systems.

References

[RB79] R. S. Boyer and J. S. Moore. *A Computational Logic*, Academic Press, 1979.

[RB98] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*, Academic Press, 1998.

[Ge36] Gentzen, G. Die Widerspruchsfreiheit der reinen Zahlentheorie *Mathematisches Annalen* 112:493–565, 1936.

[Go44] Goodstein, R.L. On the Restricted Ordinal Theorem, *Journal of Symbolic Logic* 9:33–41, 1944.

[KMM00a] M. Kaufmann, P. Manolis and J. S. Moore. *Computer-Aided Reasoning, An Approach*, Kluwer, 2000.

[KMM00b] M. Kaufmann, P. Manolis and J. S. Moore, (editors). *Computer-Aided Reasoning, ACL2 Case Studies*, Kluwer, 2000.

[Leg02] W. J. Legato, A Weakest Precondition Model for Assembly Language Programs, unpublished, February 2002. Available at `http://www.cs.uky.edu/~marek`.

[Leg05] W. J. Legato, Experimental Theorem Prover, software available from the author, 2005.

[Mc63] J. McCarthy, Towards a Mathematical Science of Computation, in *Information Processing 1962: Proceedings of IFIP Congress 1962* (C. M. Popplewell, ed.), (Amsterdam), pages 21-28, North Holland, 1963, available at `http://www-formal.stanford.edu/jmc`.

Appendix

We include here Lisp implementations for Algorithms 1 and 2, together with some testing code, available at `http://www.cs.uky.edu/~marek`.

```
;;; We represent each of the sets S(j,i) as an integer bit vector
;;;                          k
;;; s(j,i) =   sum       2
;;;          x in S(j,i)
;;;           k
;;; Let s = ((s(1,1) s(1,2) ... s(1.n))
;;;          (s(2,1) s(2,2) ... s(2,n))
;;;                 ...
;;;          (s(m,1) s(m,2) ... s(m,n)))

;;; Algorithm 1

(defun measure-ok (s)
  (do       ; loop until the prefix p is stable
   ((p 1)   ; initially p is empty
    (pp 0)) ; the old value of p
   ((equal p pp) ; when p is stable, return true iff all xi
    (equal (1+ (length (car s))) (logcount p))) ; are assigned
   (setq pp p)
   (do     ; intersect the "ready" sets over all j
    ((sj s (cdr sj))
     (pmask -1)) ; prepare to intersect ready vars over all j
    ((null sj) (setq p (logior p pmask))) ; extend prefix
    (do    ; compute the "ready" set for the jth substitution
     ((i 2 (+ i i)) ; advance i over powers of 2
      (ready 0)
      (si (car sj) (cdr si))) ; si = (S(j,1) S(j,2), ... S(j,n))
     ((null si)
      (setq pmask (logand pmask ready))) ; must be ready for j
     (and (zerop (logand p i)) ; if xi is not assigned and S(j,i)
          (or (zerop (logand p (car si))) ; holds an assigned var
              (setq ready (logior i ready)))))))) ; xi is ready
```

```
;;; Generate the s(j,i), where the kth element of the list sub
;;; represents whether variable xk went up, down or remained the
;;; same under the jth substitution.

(defun predecessor (sub)
  (let ((down
         (do ; generate the set Gj of the paper
          ((i 2 (+ i i))
           (pmask 0)
           (sub sub (cdr sub)))
          ((null sub) pmask)
          (and (equal (car sub) 'down)
               (setq pmask (logior pmask i))))))
    (do
     ((sub sub (cdr sub))
      (s nil)) ; generate sj
     ((null sub) (reverse s))
     (push (case (car sub)
                 ('equal (if (zerop down) 0 1)) ; unconstrained
                 ('down 1) ; unconstrained
                 ('up down); must follow a decreasing variable
                 (t t))  ; this should never happen
           s))))

;;; Create the list of sj's.

(defun s-gen (subs)
  (mapcar #'predecessor subs))

;;; Generate the mask of positions within sub with value key.

(defun gen-mask (sub key)
  (do ((sub sub (cdr sub))
       (i 2 (+ i i))
       (d 1))
      ((null sub) d)
      (and (equal (car sub) key)
           (setq d (+ d i)))))
```

```
;;; Algorithm 2

;;; D(j,i) = 1 if substitution j provably goes down on variable i
;;; U(j,i) = 1 if substitution j possibly goes up on variable i
;;; Variables are indexed starting with 1.  D(j,0) = U(j,0) = 1
;;;
;;; D(j) = sum D(j,i)*2^i, ds = (D(1) D(2) ... D(m))
;;;         i
;;;
;;; U(j) = sum U(j,i)*2^i, us = (U(1) U(2) ... U(m))
;;;         i
;;;

;;; due is destructive on the lists ds and us.
;;; Use (due (copy-list ds) (copy-list us))
;;; to preserve original values.

(defun due (ds us)
  (let (d u m)
    (loop
     (setq d (reduce #'logior ds)
           u (reduce #'logior us)
           m (logand d (lognot u)))
     (or (not (zerop m))
         (return (if (zerop (logior d u))
                     t
                   nil)))
     (do ((ds ds (cdr ds))
          (us us (cdr us)))
         ((null ds))
         (or (zerop (logand m (car ds)))
             (setf (car ds) 0
                   (car us) 0))))))
```

```
;;; Compare algorithms 1 and 2 on ntries randomly generated
;;; tests of nvars variables and nsubs substitutions.

(defun tess (nvars nsubs ntries)
  (let ((flg t))
    (dotimes (k ntries flg)
      (let* ((test
              (do ((j 0 (1+ j))
                   (subs nil (cons
                              (do ((i 0 (1+ i))
                                   (sub nil (cons
                                             (case (random 3)
                                                   (0 'up)
                                                   (1 'down)
                                                   (2 'equal))
                                             sub)))
                                  ((<= nvars i) sub))
                              subs)))
                  ((<= nsubs j) subs)))
             (masks (s-gen test))
             (ds (mapcar #'(lambda (x) (gen-mask x 'down)) test))
             (us (mapcar #'(lambda (x) (gen-mask x 'up)) test))
             (t1 (measure-ok masks))
             (t2 (due ds us)))
        (format t "test=~A due=~A~%" test t2)
        (format t "Algorithms agree? = ~A~%" (equal t1 t2))
        (setq flg (and flg (equal t1 t2)))))))
```