Finding Lexicographic Orders for Termination Proofs in Isabelle/HOL

Lukas Bulwahn, Alexander Krauss, and Tobias Nipkow

Technische Universität München, Institut für Informatik
http://www.in.tum.de/~{bulwahn,krauss,nipkow}

Abstract. We present a simple method to formally prove termination of recursive functions by searching for lexicographic combinations of size measures. Despite its simplicity, the method turns out to be powerful enough to solve a large majority of termination problems encountered in daily theorem proving practice.

1 Introduction

To justify recursive function definitions in a logic of total functions, a termination proof is usually required. Termination proofs are mainly a technical necessity imposed by the system, rather than in the primary interest of the user. It is therefore much desirable to automate them wherever possible, so that they "get in the way" less frequently. Such automation increases the overall user-friendliness of the system, especially for novice users.

Despite the general hardness of the termination problem, a large class of recursive functions occurring in practice can already be proved terminating using a lexicographic combination of size measures. One can see this class of functions as a generalization of the primitive recursive functions.

In this paper, we describe a simple method to generate termination orderings for this class of functions and construct a termination proof from these orderings. Unlike the naive enumeration of all possible lexicographic combinations, which is currently implemented in some systems, we use an algorithm by Abel and Altenkirch [3] to find the right order in polynomial time.

We subsequently show how, by a simple extension, our analysis can deal with mutual recursion, including cases where a descent is not present in every step. When analyzing the complexity of the underlying problem, it turns out that while there is a polynomial algorithm for the case of single functions, the presence of mutual recursion makes the problem NP-complete.

We implemented our analysis is Isabelle/HOL, where it can prove termination of 87% of the function definitions present in the Isabelle Distribution and the Archive of Formal Proofs [1].

K. Schneider and J. Brandt (Eds.): TPHOLs 2007, LNCS 4732, pp. 38–53, 2007.

1.1 Overview of the Analysis

The analysis consists of four basic steps:

1. Assemble a set of size measures to be used for the analysis, based on the type of the function argument.
2. For each recursive call and for each measure, try to prove *local descent*, i.e. that the measure gets smaller at the call. Collect the results of the proof attempts in a matrix.
3. Operating only on the matrix from step 2, search for a combination of measures, which form a global termination ordering. This combination, if it exists, can be found in polynomial time.
4. Construct the global termination ordering and, using the proofs of local descent, show that all recursive calls decrease wrt. the global ordering.

1.2 Related Work

The field of automated termination analysis is vast, and continuously attracts researchers. Many analyses (e.g. [4,13,22]) have been proposed in the literature, and some of them are very powerful. However, these methods are often hard to integrate, as they apply to different formal frameworks (such as term rewriting), and their proofs cannot be easily checked independently.

Consequently, the state of the art in the implementations of interactive theorem provers is much less developed:

In PVS [16] and Isabelle [15], and Coq [5], no automation exists, and users must supply termination orderings manually.

HOL4 [7][1] and HOL Light [8] provide some automation by enumerating all possible lexicographic orderings. For functions with more than five or six arguments, this quickly becomes infeasible.

ACL2 [10] uses heuristics to pick a size measure of a single parameter. Lexicographic combinations must be given manually, and are expressed in terms of ordinal arithmetic.

Recently, a more powerful termination criterion has been proposed for ACL2 [14], based on a combination of the size-change principle [13] and other analyses. However, the analysis is nontrivial and only available as an axiomatic extension that must be trusted, as its soundness cannot be justified within ACL2's first-order logic.

Inspired by this approach, the second author of the present paper developed a formalization of the size-change principle in Isabelle [12], which can be used to show termination for a larger class of functions. While that approach is more powerful than the one presented here, it is also more complicated and computationally expensive.

Only HOL4 tries to guess termination orderings for mutually recursive definitions. But the algorithm is a little ad-hoc and fails on many simple examples.

[1] The guessing of termination orderings in HOL4 is unpublished work by Slind, extending his work on function definitions [20,21].

The algorithm we use in §3.3 to synthesize lexicographic orderings has been discovered independently but earlier by Abel and Altenkirch [2,3]. Compared to their work, we do not just check termination but construct object-level proofs in a formal framework.

2 Preliminaries

We work in the framework of classical higher-order logic (HOL). Many examples are expressed in the Isabelle's meta-logic, with universal quantification ($\bigwedge$) and implication ($\Longrightarrow$). However, the method is not specific to HOL and could easily be adapted to other frameworks, such as type theory.

2.1 Termination Proof Obligations

General recursion is provided by a function definition package [11], which transforms a definition into a non-recursive form definable by other means. Then the original recursive specification is derived from the primitive definition in an automated process.

A termination proof is needed in order to derive the unconstrained recursive equations and an induction rule. Proving termination essentially requires to show that the call relation (constructed automatically from the definition) is wellfounded. A common way of doing this is to embed the call relation in another relation already known to be wellfounded.

As an example, consider the following function implementing the merge operation in mergesort:

$$\begin{array}{lcl} \mathit{merge}\ \mathit{xs}\ [] & = & \mathit{xs} \\ \mathit{merge}\ []\ \mathit{ys} & = & \mathit{ys} \\ \mathit{merge}\ (x{\cdot}\mathit{xs})\ (y{\cdot}\mathit{ys}) & = & \mathit{if}\ x \le y\ \mathit{then}\ x{\cdot}\mathit{merge}\ \mathit{xs}\ (y{\cdot}\mathit{ys})\ \mathit{else}\ y{\cdot}\mathit{merge}\ (x{\cdot}\mathit{xs})\ \mathit{ys} \end{array}$$

Here $\cdot$ denotes the Cons constructor for lists and [] is the empty list. In order to show termination of merge, we must prove the following subgoals:

$$\begin{array}{l} 1.\ \mathit{wf}\ ?R \\ 2.\ \bigwedge x\ \mathit{xs}\ y\ \mathit{ys}.\ x \le y \Longrightarrow ((\mathit{xs},\ y{\cdot}\mathit{ys}),\ (x{\cdot}\mathit{xs},\ y{\cdot}\mathit{ys})) \in ?R \\ 3.\ \bigwedge x\ \mathit{xs}\ y\ \mathit{ys}.\ \neg\ x \le y \Longrightarrow ((x{\cdot}\mathit{xs},\ \mathit{ys}),\ (x{\cdot}\mathit{xs},\ y{\cdot}\mathit{ys})) \in ?R \end{array}$$

Here, $?R$ is a schematic variable which may be instantiated during the proof. Hence, we must come up with a wellfounded relation, for which the remaining subgoals can be proved. The two curried arguments of *merge* have been combined to a pair, hence we can assume that there is only one argument.

In general, if the function has n recursive calls, we get the subgoals

$$\begin{array}{l} 0.\ \mathit{wf}\ ?R \\ 1.\ \bigwedge v_1 \ldots v_{m_1}.\ \Gamma_1 \Longrightarrow (r_1,\ \mathit{lhs}_1) \in ?R \\ \vdots \\ n.\ \bigwedge v_1 \ldots v_{m_n}.\ \Gamma_n \Longrightarrow (r_n,\ \mathit{lhs}_n) \in ?R \end{array}$$

Here, r_i is the argument of the call, lhs_i is the argument on the left hand side of the equation, and Γ_i is the condition under which the call occurs. These terms contain the pattern variables $v_1 ... v_{m_i}$, which are bound in the goal.

In practice, proving the termination conditions is often straightforward, once the right relation has been found. Our approach focuses on relations of a certain form, suitable for a large class of function definitions.

2.2 A Combinator for Building Relations

Isabelle already contains various combinators for building wellfounded relations. We are going to add one more to this, which particularly suits our needs.

The *measures* combinator constructs a wellfounded relation from a list of measure functions, which map function arguments into the natural numbers. This is a straightforward generalization of the well-known *measure* combinator which takes only a single function:

$$measures \;::\; (\alpha \Rightarrow nat)\ list \Rightarrow (\alpha \times \alpha)\ set$$
$$measures\ fs = inv\text{-}image\ (lex\ less\text{-}than)\ (\lambda a.\ map\ (\lambda f.\ f\ a)\ fs)$$

While the definition with predefined combinators is a little cryptic, *measures* can be characterized by the following rules:

$$\frac{f\ x < f\ y}{(x,\ y) \in measures\ (f{\cdot}fs)} \qquad \text{(Measures\_Less)}$$

$$\frac{f\ x \leq f\ y \qquad (x,\ y) \in measures\ fs}{(x,\ y) \in measures\ (f{\cdot}fs)} \qquad \text{(Measures\_Leq)}$$

And, by construction, the resulting relation is always wellfounded:

$$wf\ (measures\ fs) \qquad \text{(Measures\_WF)}$$

Our analysis will produce relations of the form *measures fs* for a suitable function list *fs*.

3 Finding Lexicographic Orderings

The overall algorithm consists of the four steps mentioned before, addressing largely orthogonal issues: Generating measure functions (by a heuristic) , proving local descents (by theorem proving), finding a lexicographic combination (by combinatorics), and reconstructing the global proof (by engineering).

We will use the previously defined *merge* function as a running example.

3.1 Step 1: Generating Measure Functions

From the type τ of the function argument, we generate a set $\mathcal{M}(\tau)$ of measure functions by a simple scheme:

$$\begin{array}{ll} \mathcal{M}(T) & = \{\lambda x.\ |x|_T\} \quad \text{if } T \text{ is an inductive data type} \\ \mathcal{M}(\tau_1 \times \tau_2) & = \{m \circ \mathit{fst} \mid m \in \mathcal{M}(\tau_1)\} \cup \{m \circ \mathit{snd} \mid m \in \mathcal{M}(\tau_2)\} \\ \mathcal{M}(\tau) & = \{\} \quad \text{if } \tau \text{ is a type variable or a function type.} \end{array}$$

For inductive data types, we return the size function $|\,.\,|_T$ associated to that type. Size functions are provided automatically by the definition package for inductive data types [6]. Product types are special and treated differently, as we are mainly interested in the measures of the different components. For products, the measures for the component types are computed recursively and composed with the corresponding projections.

This scheme basically decomposes (possibly nested) tuples and creates projection functions measuring the sizes of the components. Components with a polymorphic or a function type are ignored.

For *merge*, which operates on a pair of lists, we get the two measure functions:

$$m_1 = (\lambda x.\ |\mathit{fst}\ x|) \qquad\qquad m_2 = (\lambda x.\ |\mathit{snd}\ x|)$$

In order to handle types which are not inductive data types, the system can be extended by manually configuring one or more measure functions for them. For example, a useful measure for the integers is the function returning the absolute value.

The emphasis on size measures in this step is mainly motivated by the empirical observation that they tend to be very useful for termination proofs. In fact, the other steps do not depend on the exact nature of the measures and would work for other measures just as well.

3.2 Step 2: Proving Local Descents

A local descent is given if a given measure provably decreases at a given recursive call. For the i-th call and measure m_j, this corresponds to the following property:

$$\bigwedge v_1 \ldots v_m.\ \Gamma_i \Longrightarrow m_j\ r_i < m_j\ \mathit{lhs}_i$$

For each call and each measure, we try to prove this conjecture by a suitable automated method. If this fails, we try to prove the non-strict version instead:

$$\bigwedge v_1 \ldots v_m.\ \Gamma_i \Longrightarrow m_j\ r_i \leq m_j\ \mathit{lhs}_i$$

For the proof attempts, we use Isabelle's `auto` method, which combines rewriting with classical reasoning and some arithmetic. We collect the results of the proof attempts in a matrix M with $M_{ij} \in \{<, \leq, ?\}$, where $<$ and $\leq$ stand for a successful proof of strict or non-strict descent, and ? denotes a failure of both proofs. The theorems resulting from successful proofs are stored for later use.

For the *merge* example, the resulting matrix is

$$M_{merge} = \begin{pmatrix} < & \leq \\ \leq & < \end{pmatrix} .$$

Using the `auto` method to solve these goals is a somewhat arbitrary decision, but it turned out that our proof obligations are routinely solved by this method, usually by unfolding the size function on constructors and using arithmetic. Again, it does not matter for the rest of the development, how these goals are proved, and other methods could be plugged in here, if needed.

If the recursive arguments contain calls to "destructor functions" (like *tail* or *delete*), lemmas about these functions are usually needed in order to prove local descent. Our method makes use of such lemmas when they are available in the theory, but discovering and proving them automatically is an orthogonal issue which we do not address. A method to do this is described by Walther [22].

3.3 Step 3: Finding Lexicographic Combinations

Recall that the rows in the matrix represent the different recursive calls and the columns represent the measures. Our problem of finding a suitable lexicographic order can now be rephrased as follows:

> *Reorder the columns in the matrix in such a way that each row starts with a (possibly empty) sequence of $\leq$-entries, followed by a $<$-entry.*

Such a permutation of the columns, denoted by a list of column indices, is called a *solution*. Usually only a prefix of this list is relevant. Formally, a solution for a matrix M is a list of indices $[c_1, \ldots, c_m]$, where for all row indices i, there is a k, such that $M_{ic_k} = <$ and for all $j < k$, $M_{ic_j} = \leq$.

The following algorithm can be used to find solutions:

1. Find a column with at least one $<$ entry and no ? entry and select it as the new front column. The search fails if no such column exists.
2. Remove all rows where the selected column contains a $<$-entry, as they can be considered "solved". Then also remove the column itself and continue the search on the resulting matrix.
3. The search is successful when the matrix is empty.

Lemma 1. *If the algorithm succeeds, the selected columns are a solution for M.*

Proof. By straightforward induction.

Note that the algorithm has a choice if more than one column could be selected in step 1. However, this does not influence the overall success, since a "wrong" choice cannot lead to a dead-end. This confluence property is illustrated by the following lemma, whose proof is obvious:

Lemma 2. *Let $\sigma = [c_1, \ldots, c_m]$ be a solution, and c a column which could also be chosen in step i of the algorithm. Then $\sigma' = [c_1, \ldots, c_{i-1}, c, c_i, \ldots, c_m]$ is also a solution.*

This confluence property eliminates the need for backtracking on failure, which is the key to making the algorithm polynomial:

Lemma 3. *For an $n \times m$-Matrix, the algorithm uses $O(n^2 m)$ comparisons.*

Although choosing an arbitrary column never makes the algorithm go wrong, it may lead to suboptimal solutions. Consider the following 10×5-matrix, where the $\leq$-entries were left out for readability:

$$M = \begin{pmatrix} & < & ? \\ < & & < \\ < & & < \\ < & & < \\ < & & < \\ < & & < \\ < & & < \\ < & & < \\ & & < < \\ & & < < \end{pmatrix}$$

Naively proceeding from left to right will produce the solution $[1, 2, 3, 4]$, although the shorter solution $[3, 4]$ exists. In systems where proof objects are stored explicitly, their size can influence efficiency, and it might be worthwhile to invest some effort here, to keep proofs small.

As a heuristic, it might seem advantageous to always choose the column with the most $<$-entries, since this would eliminate the corresponding rows once and for all. But this greedy strategy would produce the solution $[2, 1, 5, 3, 4]$ on the above matrix, which is even bigger than with the naive strategy.

In fact, finding the shortest solution is much harder than finding just one solution:

Lemma 4. *Given a Matrix M, the optimization problem of finding a minimal solution for M is NP-hard.*

Proof. The optimization version of the NP-hard Set Cover problem [18] can easily be expressed in a matrix: For a universe $U = \{x_1, \ldots, x_n\}$ and a collection $S_1, \ldots, S_m$ of subsets of U, construct the $n \times m$-matrix M with $M_{ij} = <$ if $x_i \in S_j$ and $\leq$ otherwise. Obviously, every solution for M solves the Set Cover problem and vice-versa.

Since searching for the smallest solution can be hard, we are happy with choosing just the first suitable column in our implementation.

For our example M_{merge}, both $[1, 2]$ and $[2, 1]$ are solutions.

3.4 Step 4: Proof Reconstruction

It remains to assemble the pieces and construct a global relation and a global proof from the solution of the previous step.

From a solution $\sigma = [c_1, \ldots, c_n]$ we construct the relation

$$R = \mathit{measures}\ [m_{c_1}, \ldots, m_{c_n}]\ .$$

It is now straightforward to prove the termination goals with the help of the matrix M and the solution σ:

1. Instantiate the schematic variable *?R* by the relation R.
2. Wellfoundedness is trivial by the rule MEASURES_WF.
3. For each call, inspect the matrix in M_{ic_j} for increasing j. If $M_{ic_j} = \leq$, apply rule MEASURES_LEQ and use the stored theorem from §3.2 to solve its first premise. When $M_{ic_j} = <$, apply rule MEASURES_LESS, and the stored theorem, which solves the goal.

By construction, the theorems resulting from the proofs in §3.2 exactly match the premises of MEASURES_LEQ and MEASURES_LESS, respectively.

As an example, consider the *merge* function again. We get the relation

$R = \mathit{measures}\ [\lambda x.\ |\mathit{fst}\ x|,\ \lambda x.\ |\mathit{snd}\ x|]$

After instantiating the schematic variable and solving the wellfoundedness, two subgoals remain:

1. $\bigwedge x\ xs\ y\ ys.$
$x \leq y \Longrightarrow ((xs,\ y{\cdot}ys),\ (x{\cdot}xs,\ y{\cdot}ys)) \in \mathit{measures}\ [\lambda x.\ |\mathit{fst}\ x|,\ \lambda x.\ |\mathit{snd}\ x|]$
2. $\bigwedge x\ xs\ y\ ys.$
$\neg\ x \leq y \Longrightarrow ((x{\cdot}xs,\ ys),\ (x{\cdot}xs,\ y{\cdot}ys)) \in \mathit{measures}\ [\lambda x.\ |\mathit{fst}\ x|,\ \lambda x.\ |\mathit{snd}\ x|]$

Since $M_{11} = <$, we can directly apply MEASURES_LESS to the first subgoal:

1. $\bigwedge x\ xs\ y\ ys.\ x \leq y \Longrightarrow |\mathit{fst}\ (xs,\ y{\cdot}ys)| < |\mathit{fst}\ (x{\cdot}xs,\ y{\cdot}ys)|$
2. $\bigwedge x\ xs\ y\ ys.$
$\neg\ x \leq y \Longrightarrow ((x{\cdot}xs,\ ys),\ (x{\cdot}xs,\ y{\cdot}ys)) \in \mathit{measures}\ [\lambda x.\ |\mathit{fst}\ x|,\ \lambda x.\ |\mathit{snd}\ x|]$

Now the first subgoal is exactly what we have proved in §3.2, so we can solve it using the stored theorem.

For the remaining subgoal, we must first apply MEASURES_LEQ, since $M_{21} = \leq$:

1. $\bigwedge x\ xs\ y\ ys.\ \neg\ x \leq y \Longrightarrow |\mathit{fst}\ (x{\cdot}xs,\ ys)| \leq |\mathit{fst}\ (x{\cdot}xs,\ y{\cdot}ys)|$
2. $\bigwedge x\ xs\ y\ ys.\ \neg\ x \leq y \Longrightarrow ((x{\cdot}xs,\ ys),\ (x{\cdot}xs,\ y{\cdot}ys)) \in \mathit{measures}\ [\lambda x.\ |\mathit{snd}\ x|]$

Again, we solve the first subgoal using the pre-proved theorem. In one additional step, we can solve the remaining goal using MEASURES_LESS, which finishes the termination proof.

4 Mutual Recursion

For mutually recursive functions, termination generally has to be proved simultaneously. To allow for a uniform treatment, Isabelle internally converts such

definitions into a definition of a single function operating on the sum type, which is a standard transformation technique (see e.g. [21]).

Consider the functions *even* and *odd*, defined by mutual recursion:

$$\begin{array}{lcl} even\ 0 & = & True \\ even\ (Suc\ n) & = & odd\ n \\ odd\ 0 & = & False \\ odd\ (Suc\ n) & = & even\ n \end{array}$$

For the termination proof, the function is seen as a single function over the sum type $nat + nat$. This leads to to the following proof obligations[2]:

1. $wf\ ?R$
2. $\bigwedge n.\ (Inr\ n,\ Inl\ (Suc\ n)) \in ?R$
3. $\bigwedge n.\ (Inl\ n,\ Inr\ (Suc\ n)) \in ?R$

It is not hard to generalize our approach to prove termination for mutual recursions. Unfortunately, we will see that this necessarily destroys the polynomial time complexity.

4.1 Measures for Sum Types

In order to use our analysis on these problems, we must be able to generate measure functions on sum types. Thus we extend the measure generation step (§3.1) as follows:

$$\mathcal{M}(\tau_1 + \tau_2) = \{case_+\ m_1\ m_2 \mid m_1 \in \mathcal{M}(\tau_1),\ m_2 \in \mathcal{M}(\tau_2)\}$$

This means that sum measures are built by taking all combinations of the measures for the component types and combining them with the case combinator for sum types:

$$case_+\ ::\ (\alpha \Rightarrow \gamma) \Rightarrow (\beta \Rightarrow \gamma) \Rightarrow (\alpha + \beta \Rightarrow \gamma)$$

Intuitively, each of the mutually recursive functions gets its own measure, which is applied to its arguments.

Since *even* and *odd* both get only one measure function, $case_+\ (\lambda x.\ |x|)\ (\lambda x.\ |x|)$ is the only combination, and with this measure the proof is indeed successful. The other parts of the analysis require no change.

4.2 Ordering the Functions

With the measures on sum types, many mutually recursive definitions can be handled without problems. But the approach fails for definition like the following:

[2] We denote the injection functions for sum types with *Inl* and *Inr*. If more than two functions are defined simultaneously, nested sums will occur.

$$\begin{array}{lcl} f\ 0 & = & 0 \\ f\ (Suc\ n) & = & g\ n \\ g\ n & = & Suc\ (f\ n) \end{array}$$

In the call from g to f, there is no descent, since the only argument is just passed along. The reason why this definition terminates is not that the argument decreases, but that in this call, the *function* decreases, with respect to a suitable ordering of the functions (here: $f < g$).

With our sum encoding, it is simple to capture this argument by suitable measure functions: It suffices to add measure functions which distinguish between the functions, but are otherwise constant. For f and g, we add the measure $case_+\ (\lambda x.\ 0)\ (\lambda x.\ 1)$. This results in the matrix

$$M = \begin{pmatrix} < & ? \\ \leq & < \end{pmatrix}$$

In general, it is sufficient to add measures which evaluate to 1 for one of the functions and to 0 for all others. $\mathcal{M}_{\mathsf{fn}}(\tau)$ describes these measures formally, again depending only on the argument type τ:

$$\begin{array}{lll} \mathcal{M}_{\mathsf{fn}}(\tau_1 + \tau_2) & = & \{case_+\ m_1\ (\lambda x.0) \mid m_1 \in \mathcal{M}_{\mathsf{fn}}(\tau_1)\} \cup \\ & & \{case_+\ (\lambda x.0)\ m_2 \mid m_2 \in \mathcal{M}_{\mathsf{fn}}(\tau_2)\} \\ \mathcal{M}_{\mathsf{fn}}(\tau) & = & \{(\lambda x.1)\} \quad \text{if } \tau \text{ is not a sum type} \end{array}$$

It is easy to see that with these additional measures (and their lexicographic combinations, which are formed by the subsequent steps), any partial ordering between the functions can be captured.

4.3 Complexity of Mutual Recursion

Taking all combinations of measures for sum types as in §4.1 is a brute-force strategy and it destroys the polynomial runtime behaviour of the algorithm, since the number of sum measures is obviously exponential in the number of functions involved. The question arises, whether we can be a little more intelligent here.

Note in particular that we have introduced some redundancy in the proofs that are performed in step 2: In a call from f to g, it is completely irrelevant what measure we assign to h, and sum measures differing only in the components for functions other than f and g will lead to identical proofs.

It is a better strategy to look at the different calls separately: Consider a call c from f to g (written $c : f \to g$), which corresponds to a goal of the form

$$\bigwedge v_1 \ldots v_m.\ \Gamma_i \Longrightarrow (In_g\ r_i,\ In_f\ lhs_i) \in ?R$$

Here we write In_f and In_g to abbreviate the compositions of injections corresponding to the functions f and g, respectively.

We can now generate separate sets $\mathcal{M}_f$ and $\mathcal{M}_g$ of measure functions for f and g separately and try for which choice of $m_f \in \mathcal{M}_f$ and $m_g \in \mathcal{M}_g$ the following becomes provable:

$\bigwedge v_1 \ldots v_m.\ \Gamma_i \Longrightarrow m_g\ r_i < m_f\ lhs_i$

As before, we try the $\leq$ version if the proof fails and collect the results in a matrix. This time we get one matrix M^c for each recursive call c, and the rows and columns of the matrix stand for the Elements of $\mathcal{M}_f$ and $\mathcal{M}_g$. Note that this information is still polynomial in size.

We must now construct measures on the sum type which are useful to prove termination. At least one of these sum measures must be such that it increases in none of the calls. Such a measure would be used first in the construction of the lexicographic combination.

Abstracting from functions and measures and the like, we have the following problem, which we call the COMBINATION problem:

> Let A and B be finite sets and $M \subseteq A \times A \times B \times B$. Find a mapping $f : A \to B$ such that $\forall x, x' \in A.\ (x, x', f(x), f(x')) \notin M$.

The abstraction is as follows: A is the set of functions, and B is the set of basic measures, where we assume without loss of generality that the number of basic measures is the same for all functions (take the maximum!). The set M encodes the result of the proofs: $(f, g, i, j) \in M$ iff for some call $c : f \to g$, $M^c_{ij} =?$

Is is a simple exercise to construct a set of functions from an arbitrary COMBINATION instance to see that finding sum measures is indeed required to solve this problem.

Lemma 5. *The* COMBINATION *problem is NP-complete.*

Proof. We reduce the 3-COLORING problem for graphs (see e.g. [18]) to COMBINATION. For a graph $G = (V, E)$ to be coloured, we define the problem instance

$$A = V \qquad B = \{\text{Red}, \text{Green}, \text{Blue}\} \qquad M = \{(v, v', c, c) \mid (v, v') \in E,\ c \in B\}.$$

Now every solution f for COMBINATION is a valid colouring: if $(v, v') \in E$ and $f(v) = c$ we have $(v, v', c, c) \in M$ and thus $f(v') \neq c$. Conversely, every colouring is clearly a valid COMBINATION. Since 3-COLORING is NP-hard, so is our problem, and since checking a given solution is trivial, it is also NP-complete.

What conclusions can be drawn from this result? First, it shows that the introduction of mutual recursion really complicates matters, leading to a search space exponential in the number of functions involved. Second, this complexity only concerns the *search* for suitable measures. The number of *proof attempts* to be performed by the theorem prover is still polynomial.

While this is interesting from a theoretic view, in practice the number of functions in a single mutually recursive definition is often quite small, which makes this approach feasible.

5 Examples

5.1 Ackermann Function

The ackermann function, defined by

$$\begin{array}{lcl} ack\ 0\ m & = & Suc\ m \\ ack\ (Suc\ n)\ 0 & = & ack\ n\ 1 \\ ack\ (Suc\ n)\ (Suc\ m) & = & ack\ n\ (ack\ (Suc\ n)\ m) \end{array}$$

is easily proved total by our tool. The generated relation is *measures* $[\lambda x.\ |fst\ x|, \lambda x.\ |snd\ x|]$.

5.2 Many Parameters

If the function has many parameters, enumerating lexicographic orders becomes infeasible, as the following example demonstrates. Both HOL4 and HOL Light fail to prove termination of the following function in reasonable time, while our method succeeds within a second.

$$\begin{array}{lcl} blowup\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & = & 0 \\ blowup\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ (Suc\ i) & = & Suc\ (blowup\ i\ i\ i\ i\ i\ i\ i\ i\ i) \\ blowup\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ (Suc\ h)\ i & = & Suc\ (blowup\ h\ h\ h\ h\ h\ h\ h\ h\ i) \\ blowup\ 0\ 0\ 0\ 0\ 0\ 0\ (Suc\ g)\ h\ i & = & Suc\ (blowup\ g\ g\ g\ g\ g\ g\ g\ h\ i) \\ blowup\ 0\ 0\ 0\ 0\ 0\ (Suc\ f)\ g\ h\ i & = & Suc\ (blowup\ f\ f\ f\ f\ f\ f\ g\ h\ i) \\ blowup\ 0\ 0\ 0\ 0\ (Suc\ e)\ f\ g\ h\ i & = & Suc\ (blowup\ e\ e\ e\ e\ e\ f\ g\ h\ i) \\ blowup\ 0\ 0\ 0\ (Suc\ d)\ e\ f\ g\ h\ i & = & Suc\ (blowup\ d\ d\ d\ d\ e\ f\ g\ h\ i) \\ blowup\ 0\ 0\ (Suc\ c)\ d\ e\ f\ g\ h\ i & = & Suc\ (blowup\ c\ c\ c\ d\ e\ f\ g\ h\ i) \\ blowup\ 0\ (Suc\ b)\ c\ d\ e\ f\ g\ h\ i & = & Suc\ (blowup\ b\ b\ c\ d\ e\ f\ g\ h\ i) \\ blowup\ (Suc\ a)\ b\ c\ d\ e\ f\ g\ h\ i & = & Suc\ (blowup\ a\ b\ c\ d\ e\ f\ g\ h\ i) \end{array}$$

5.3 Multiplication by Shifting and Addition

Pandya and Joseph [17] introduced a new proof rule for total correctness of mutually recursive procedures. The contribution of this proof rule is a refined method for proving termination by analysing the procedure call graph. They motivate their approach with an imperative version of the following example:

$$\begin{array}{lcl} prod\ x\ y\ z & = & \textsf{if}\ y\ mod\ 2 = 0\ \textsf{then}\ eprod\ x\ y\ z\ \textsf{else}\ oprod\ x\ y\ z \\ oprod\ x\ y\ z & = & eprod\ x\ (y - 1)\ (z + x) \\ eprod\ x\ y\ z & = & \textsf{if}\ y = 0\ \textsf{then}\ z\ \textsf{else}\ prod\ (2 * x)\ (y\ div\ 2)\ z \end{array}$$

In the calls from *oprod* and *eprod* the second argument decreases but in the calls from *prod* all arguments are unchanged. Termination is proved automatically because one can order the functions such that *oprod* and *eprod* are less than *prod*.

5.4 Pedal and Coast

Homeier and Martin [9] describe an intricate call graph analysis for which Homeier holds a US patent. Their one example is an imperative version of what they call the *bicycling* program:

$$
\begin{array}{lcl}
\mathit{pedal} & :: & \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \\
\mathit{coast} & :: & \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \\
 & & \\
\mathit{pedal}\ 0\ m\ c & = & c \\
\mathit{pedal}\ n\ 0\ c & = & c \\
\mathit{pedal}\ (\mathit{Suc}\ n)\ (\mathit{Suc}\ m)\ c & = & \mathsf{if}\ n < m \\
 & & \mathsf{then}\ \mathit{coast}\ n\ m\ (c + \mathit{Suc}\ m) \\
 & & \mathsf{else}\ \mathit{pedal}\ n\ (\mathit{Suc}\ m)\ (c + \mathit{Suc}\ m) \\
\mathit{coast}\ n\ m\ c & = & \mathsf{if}\ n < m \\
 & & \mathsf{then}\ \mathit{coast}\ n\ (m - 1)\ (c + n) \\
 & & \mathsf{else}\ \mathit{pedal}\ n\ m\ (c + n)
\end{array}
$$

They claim that termination would be difficult to prove using the rule by Pandya and Joseph. With our algorithm the proof is automatic: In the call from *coast* to *pedal*, no argument decreases. But by ordering the functions such that *pedal* $<$ *coast*, the proof succeeds. In both examples, ordering the functions (see 4.2) is essential. A precise analysis of the relationship between all three termination proof methods is beyond the scope of this paper.

By the way: $\mathit{prod}\ x\ y\ z = x * y + z$ and $\mathit{pedal}\ n\ m\ c = n * m + c$ can be proved (the latter automatically) via the customized induction principles generated from the function definitions [11].

6 Practical Considerations and Possible Extensions

6.1 Empirical Evaluation

From the existing non-primitive recursive function definitions in the Isabelle Distribution and the Archive of Formal Proofs [1], our method can find suitable orderings for 87% of them, usually in less than a second.

The examples where it fails mainly fall in one of the following categories:

1. Definitions which use a customized size function, where some constructors are weighted more than others. This method is essentially polynomial interpretation, done manually.
2. Functions over naturals or integers, where the argument is increasing but bounded from above.
3. Examples for "difficult" functions from the documentation, where a domain specific semantic argument is used for termination.
4. Functions over more powerful set theoretic constructions like ordinals.
5. Functions that aren't total and thus require special treatment anyway.

6.2 Feedback from Failure

If our analysis fails to find a termination proof, this can have several reasons:

1. The function is indeed non-terminating.
2. The function is terminating, but the termination argument is more complicated than just lexicographic combinations of size measures.

3. Lexicographic orders are sufficient, but the automation employed to prove local descents is not sufficient to derive the required facts, although they are true.

Of course it is impossible for the system to distinguish between these three classes of errors. However, instead of just failing, we provide valuable information to the user by showing the matrix with the proof results. For example, the matrix $\begin{pmatrix} < & \leq \\ ? & \leq \end{pmatrix}$ will tell us that there must be something wrong with the second recursive call. This can be a useful debugging aid, at least if the user has a basic understanding of how the method works.

A few other enhancements help to improve feedback given to the user:

- When proving local descent, many unprovable goals get simplified to *False*. For these cases, giving the matrix the entry F instead of ? makes it clear that the $\leq$-inequality is clearly false. Apart from output, F and ? are not treated differently.
- For unsuccessful proof attempts, printing the unfinished proof states can give the user feedback on lemmas that might be necessary in order for the proof to work.
- If a row in the matrix does not have a strict descent ($<$) at all, one could highlight the corresponding recursive call because it (or its proof) needs special user attention.

6.3 Using Other Measure Functions

Our measure generation has an emphasis on size measures of data types, When working on different data, other size functions will be appropriate. We already mentioned the absolute value of an integer. User-defined types will have their own notions of structure and size, and our analysis is able to accommodate this, if users can declare their own measure functions.

Moreover, while the type based choice of measures is simple, this is not the only possible solution.

In the realm of linear arithmetic, Podelski and Rybalchenko [19] describe how to synthesize measures (which they call *ranking functions*) for simple non-nested while loops. It would be interesting to try to use such a method to generate measures for our approach, which can then be combined with others (say, from data types) to more complex termination proofs.

6.4 Incremental Operation

At the moment, the four steps of our algorithm are executed strictly sequentially. In particular, all proof attempts have to be finished and the matrix has to be complete, before the search for the lexicographic combination begins. This separation simplifies presentation and implementation, but is otherwise unnecessary.

The structure of the system could be modified in a way that the columns of the matrix are produced incrementally. Once a column "arrives", the search algorithm checks immediately, whether it can be used to solve some of the calls. Otherwise it must be stored because it could be useful in a later step.

Such an architecture allows for several immediate optimizations:

- Once an ordering is found, no more proofs about other measures have to be tried.
- If certain rows (=calls) have already been solved by previous measures, they can be ignored in subsequent proof attempts. Instead, the corresponding entries can be set immediately to ?, since they will not be used anyway.

These changes do not increase the power of the analysis, but only its efficiency. However, such a lazy strategy would allow us to increase the number of measure functions to try, without compromising the efficiency of the whole system for examples where they are not needed.

7 Conclusion

We showed how to automate termination proofs for a large class of practically relevant function definitions. Our implementation, which is already part of the Isabelle developer version, is invoked by default for all new function definitions. This makes the termination proof invisible in many cases, which helps users concentrate on their actual theorem proving tasks.

References

1. Archive of Formal Proofs, `http://afp.sourceforge.net/`
2. Abel, A.: foetus – termination checker for simple functional programs. Programming Lab Report (1998)
3. Abel, A., Altenkirch, T.: A predicative analysis of structural recursion. J. Functional Programming 12(1), 1–41 (2002)
4. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theor. Comput. Sci. 236(1-2), 133–178 (2000)
5. Barthe, G., Forest, J., Pichardie, D., Rusu, V.: Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, Springer, Heidelberg (2006)
6. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL - lessons learned in formal-logic engineering. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 19–36. Springer, Heidelberg (1999)
7. Gordon, M., Melham, T. (eds.): Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press, Cambridge (1993)
8. Harrison, J.: The HOL Light theorem prover, `http://www.cl.cam.ac.uk/users/jrh/hol-light`
9. Homeier, P.V., Martin, D.F.: Mechanical verification of total correctness through diversion verification conditions. In: Grundy, J., Newey, M. (eds.) Theorem Proving in Higher Order Logics. LNCS, vol. 1479, pp. 189–206. Springer, Heidelberg (1998)

10. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach, June 2000. Kluwer Academic Publishers, Dordrecht (2000)
11. Krauss, A.: Partial recursive functions in higher-order logic. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 589–603. Springer, Heidelberg (2006)
12. Krauss, A.: Certified size-change termination. In: Pfenning, F. (ed.) CADE-21. LNCS, vol. 4603, pp. 460–476. Springer, Heidelberg (to appear, 2007)
13. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 81–92. ACM Press, New York (2001)
14. Manolios, P., Vroon, D.: Termination analysis with calling context graphs. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 401–414. Springer, Heidelberg (2006)
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
16. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) Automated Deduction - CADE-11. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
17. Pandya, P., Joseph, M.: A Structure-directed Total Correctness Proof Rule for Recursive Procedure Calls. The Computer Journal 29(6), 531–537 (1986)
18. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley, New York (1994)
19. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
20. Slind, K.: Function definition in Higher-Order Logic. In: von Wright, J., Harrison, J., Grundy, J. (eds.) TPHOLs 1996. LNCS, vol. 1125, pp. 381–397. Springer, Heidelberg (1996)
21. Slind, K.: Reasoning About Terminating Functional Programs. PhD thesis, Institut für Informatik, TU München (1999)
22. Walther, C.: On proving the termination of algorithms by machine. Artif. Intell. 71(1), 101–157 (1994)