SPLASH: U: Termination of Recursive Functions by Lexicographic Orders of Linear Combinations

raphaeldouglasgiles@gmail.com
UNSW Sydney
Australia

1 Problem and Motivation

The widely used proof assistant Isabelle/HOL has a simple yet effective algorithm [3] for generating proofs that certain recursive functions terminate. Isabelle/HOL's termination checker works by finding a lexicographic ordering on the measures generated by each argument of the function using the structural ordering [2] which decreases at every recursive call. This project seeks to extend this algorithm to check for linear combinations of these measures with coefficients in the natural numbers to prove that a broader class of functions terminate.

2 Background and Related Work

2.1 Preliminaries

In functional languages, non-termination comes from one source: recursion. A function which does not recurse, and which only calls functions that do not recurse, will assuredly terminate. Thus, to determine when a function terminates, we must inspect the recursive structure of that function.

Recursive functions may be defined by a collection of pattern matching rules, similar to Haskell. Each line in a function definition defines a pattern for the input data to match, and an output term for the function to return in that case. We call these the defining equations of that function. For example, the binomial coefficient can be written in this form (Figure 1).

$$\begin{aligned} &\mathit{choose}\ (0, k) && = 1 \\ &\mathit{choose}\ (n, 0) && = 1 \\ &\mathit{choose}\ (\mathrm{S}\, n, \mathrm{S}\, k) && = \mathit{choose}\ (n, \mathrm{S}\, k) + \mathit{choose}\ (n, k) \end{aligned}$$

Figure 1. The binomial coefficient as a functional program.

A proof that *choose* terminates is going to rely on the fact that the first projection of the argument, labeled n, strictly decreases every time we make a recursive call. Since n is a natural number, it is eventually going to hit 0 at which point we can't decrease it anymore, and so our function must terminate.

Formalising this idea, we say a recursive function terminates if there is a *reduction order* that, for every defining equation of that function, orders the left term as strictly greater than the right term of that equation. While we use the theory of reduction orders in this paper, this paper is not a tutorial on them. We recommend Baader and Nipkow [8] for a more thorough introduction to the theory of termination.

As, in this paper, we are concerned with the termination of single functions, we make two (common) simplifications. Firstly, instead of defining a reduction order directly on terms, we will define *measure functions* which take terms to some well ordered structure, mimicking the association of natural numbers with the arguments of our function in the above informal proof. This structure is commonly the natural numbers (we will use the definition with zero), but we will also make use of lexicographically ordered lists of natural numbers.

Secondly, we will take the measure of the *argument* to the function, not the whole term. This step is justified by taking the full-term measure of a function to just be the measure of its argument, and the measure of any compound term containing the function (for example, the right hand side of the third line of *choose*) to be the maximum of the argument measures of every function occurrence in that term. Note that this step further constrains the measures we may use. The ordered structure produced by the measure must now also have a unique bottom element. (Due to the use of maximum.) Note further that natural numbers and lexicographic lists of natural numbers meet this constraint.

Applying this to *choose*, we can encode the above informal argument as showing that the measure

$$\lambda t.\ \mathbf{case}\ t\ \mathbf{of}\ \langle n, k\rangle \Rightarrow |n| \mid \_ \Rightarrow 0,$$

decreases at every recursive call. Now, this is a rather unwieldy function definition because it makes no assumption about the type of the given term. For the sake of clarity and ease of communication, we will henceforth say that $\lambda n.\ |n|$, or better yet *size*, the structural size function on our given type, decreases at every recursive call. This represents a slight diversion from the approach taken in our Haskell implementation and paper in progress, where measures are considered in a context which works regardless of the presence of types.

Of course, termination arguments are not always so simple as showing one element of a tuple decreases every time we make a recursive call (which is good news for those of us who research termination checking). For example, consider the following function:

$$g(\mathrm{S}\,x, y) = g(x, \mathrm{S}\,y)$$
$$g(x, \mathrm{S}\,y) = g(x, y)$$

We assume that g, and any function written in this paper, outputs 0 if none of the specified rules apply (i.e. $g(0,0) = 0$).

Here, there is no one part of the argument which decreases and will never increase again, so we don't immediately get termination just from generating the measures on the arguments like we did before. However, again we have an informal argument for termination here which we can formalise and abstract to prove termination for more problems. Here, we notice that while there is no part of the argument which decreases at every recursive call, the first projection of the tuple, labeled x, can never increase again once it has decreased. This is because in every recursive call x either decreases or stays the same size. This means that we can only apply the first recursive rule where x decreases a finite number of times, because the number of calls we can make to this rule is bounded by the value of x we started with. Hence, calls to this rule cannot be the source of non-termination. So, ignoring the first rule because we know it can only be called finitely many times, the second rule can only be called a finite number of times as well since the second projection of the argument, y, is always decreasing.

Another way to phrase this is that the lexicographic combination of the size measure for the first argument and the size measure for the second argument decreases at every recursive call, a measure we will express symbolically as:

$$size \circ \pi_0 * lex * size \circ \pi_1$$

where π_0 and π_1 are the first and second tuple projection functions. Isabelle/HOL's lexicographic termination checker [3] provides an algorithm for finding such a lexicographic combination of measures generated by the arguments which decreases at every recursive call if it exists, and determining when it doesn't exist. It does this in polynomial time.

2.2 Related Work

Isabelle/HOL's lexicographic [3] and size-change [6] termination checkers represent the current state of the art in termination checking for theorem provers, being strictly more powerful than the termination checkers found in Agda [1], [2] and Coq [5]. Our extension of Isabelle/HOL's lexicographic algorithm [3] both maintains the complexity class of and solves a superset of the termination problems solved by this algorithm.

Our approach is similar to Abel and Altenkirch [2] and Bulwahn et al. [3], but improves on their algorithms in that our approach allows for linear combinations measures.

2.3 Lexicographic Algorithm

We now give a brief overview of Isabelle/HOL's lexicographic algorithm [3] [2]. Firstly, this algorithm compares the measures of each argument on every recursive call between the input value and the value passed to the recursive call. For example, let's again consider our function g from before, which we recall was defined by:

$$g(\mathrm{S}\,x, y) = g(x, \mathrm{S}\,y)$$
$$g(x, \mathrm{S}\,y) = g(x, y)$$

To represent the size-change information symbolically, let $<$ mean the measure decreased, ? mean we either don't know or the measure increased and $=$ mean the measure didn't change. We can then summarise the changes in size of measures of g in each recursive call with the following matrix [3],

$$\begin{array}{cc} size \circ \pi_0 & size \circ \pi_1 \\ \left[\begin{array}{cc} < & ? \\ = & < \end{array}\right] & \end{array}$$

where each row corresponds to a recursive call and each column corresponds to the measure of each argument. We will from this point forward omit the labels of the columns of this matrix, since it will often be clear and we wish to orient the reader slowly to thinking about these as matrices in the sense of linear algebra. We use the convention that if all the measures are of the form $size \circ \pi_n$, then the ith column represents the measure $size \circ \pi_i$. The algorithm works by mimicking the informal argument given above; we look for arguments which are always either decrease or stay the same size (which will have $<$ and $=$ entries) and remove these recursive calls from consideration. Phrased purely in terms of operations on the matrix, for every column c of our matrix with only $<$ and $=$ entries, we remove every row of the matrix with a $<$ entry in c. If we eventually remove all the rows of the matrix from repeating this process, then the function must terminate. To find the corresponding measure, we just have to keep track of these columns (measures) c. If c_1 is the first measure we find where we can remove elements from the corresponding column, c_2 the second and so on, then the measure which decreases at every recursive call will be:

$$c_1 * lex * c_2 * lex * \ldots * lex * c_n.$$

For our example, if we represent one iteration of the algorithm with $\rightsquigarrow$ and the empty matrix as $\varnothing$, this algorithm would look like this:

$$\begin{bmatrix} < & ? \\ = & < \end{bmatrix} \rightsquigarrow \begin{bmatrix} = & < \end{bmatrix} \rightsquigarrow \varnothing.$$

The corresponding measure which decreases at every recursive call would be:

$$fst \circ size * lex * snd \circ size.$$

as we expect.

2.4 The Problem

This procedure works for many functions, including the merge function and the Ackermann function; functions which do not obviously terminate. However, consider the following function f:

$$f(\mathrm{S}\,\mathrm{S}\,x, y) = f(x, \mathrm{S}\,y)$$
$$f(x, \mathrm{S}\,\mathrm{S}\,y) = f(\mathrm{S}\,x, y)$$

We know that this function must terminate because the sum of the arguments is clearly always decreasing. In other words, we know that the measure $\pi_0 \circ size + \pi_1 \circ size$ decreases at every recursive call, where we define addition of these measures pointwise using addition on the natural numbers. However, we can't show this using Isabelle/HOL's algorithm. We can see this directly by generating the matrix,

$$\begin{bmatrix} < & ? \\ ? & < \end{bmatrix}$$

which can not be reduced at all using the above procedure since there is no column which only has < and = entries.

3 Uniqueness of the Approach

3.1 Approach

Our algorithm proves the termination of a function in three steps. Firstly, we generate several *primitive measures*, that measure particular parts of the term structure; for example, the left or right hand sides of a tuple, the size of a list, or more generally the structural size of types where this is defined. These primitive measures are generated by analysis of the arguments that occur on the left and right hand side of each defining equation. For a more detailed account of how these primitive measures are generated, we refer the reader to Bulwahn et al. [3].

Then, for every recursive call, we measure the difference between each of these measures applied at the recursive call to the function and the original call to the function. Of course, it's not always possible to get a precise value for this size change information, so it's sometimes necessary to take upper bounds or not have any information about the size change of a particular measure at a particular recursive call.

We tabulate this generated information into a *matrix* of *size changes*. We then apply operations to this matrix to find a *combination* of primitive measures which shows that the overall function terminates. The combination is a lexicographic combination of linear combinations of the primitive measures.

3.2 Extending the Algorithm

Our solution to the problem the Isabelle/HOL algorithm faces is to first change how we represent size-change information. Instead of just saying a particular measure decreased or stayed the same and so on, we now use specific, numeric values by which the sizes changed (when we can find such values) as our matrix entries. When we can't say, we put a ? entry. In instances where the size of a particular argument could increase in size by some integer n or could increase in size by some integer m where $m > n$, we put an m in the corresponding position in the matrix. In this way, the size-change matrix represents worst-case size-change information. For instance, our example f would now give the matrix:

$$\begin{bmatrix} -2 & 1 \\ 1 & -2 \end{bmatrix}.$$

This allows us to add the columns together yielding the matrix $\begin{bmatrix} -1 \\ -1 \end{bmatrix}$ which clearly terminates. We provide a Haskell implementation which demonstrates how to generate such a matrix for functions expressed in a simple lambda calculus where functions must be defined by pattern matching (making the calculation of differences in structural size significantly easier).

This operation of adding together columns of numeric values always yields a new measure. For the same reason, taking any linear combination of numeric columns with non-negative natural coefficients is a valid matrix reduction operation/will yield a valid measure (where we again define this operation pointwise). Finding such a linear combination with natural coefficients is equivalent to doing so with positive rational coefficients, since we can multiply out by a common denominator to get a corresponding natural linear combination. Analogous observations have been made in the context of logic programming [7], [4], though the methods developed there are not directly applicable in our context.

This is the main idea underlying our extension to the algorithm. Namely, if we can find some positive rational linear combination of the columns of our matrix with only numeric entries (i.e. no ?s) such that the resulting vector is less than or equal to 0 in every component, then we can remove some more rows with the lexicographic algorithm and repeat this process to hopefully prove termination. To see what we mean, consider the following (admittedly contrived) example:

$$h(\mathrm{S}\,x, y, z, v, w) = h(x, rand(), z, v, \mathrm{S}\,w)$$
$$h(x, \mathrm{S}\,y, \mathrm{S}\,z, v, w) = h(rand(), y, z, \mathrm{S}\,v, \mathrm{S}\,w)$$
$$h(x, y, z, \mathrm{S}\,\mathrm{S}\,\mathrm{S}\,v, w) = h(x, rand(), \mathrm{S}\,z, v, \mathrm{S}\,w)$$

These $rand()$ functions appearing in the definition are just meant to be random numbers. How these are implemented is unimportant, the point of their inclusion is that they represent a function whose output is unpredictable, and so must irrefutably be given ? entries in any matrix, regardless of the procedure being used to generate such a matrix. This

particular example has been chosen because it has a complicated enough matrix to illustrate the behaviour we'd like, that matrix being:

$$\begin{bmatrix} -1 & ? & 0 & 0 & 1 \\ ? & -1 & -1 & 1 & 1 \\ 0 & ? & 1 & -3 & 1 \end{bmatrix}$$

Here, we're again in a situation where we cannot directly apply the lexicographic algorithm. However, we also cannot in this instance just find a positive rational linear combination of all the columns giving us a vector which is strictly less than 0 in all its components. Using our idea, we'd like to find some linear combination of the last three columns here so that we can get rid of some of these ? entries. Happily, it's not too hard to see that, labelling the columns of the matrix $x_0, x_1 ... x_4$, we can take $2x_2 + x_3 + 0x_4 = \begin{bmatrix} 0 \\ -1 \\ -1 \end{bmatrix}$. This then yields the following matrix:

$$\begin{bmatrix} -1 & ? & 0 \\ ? & -1 & -1 \\ 0 & ? & -1 \end{bmatrix}.$$

We now use the lexicographic algorithm to remove the last two rows, resulting in the matrix:

$$\begin{bmatrix} -1 & ? & 0 \end{bmatrix},$$

which we can then reduce to $\varnothing$ using the lexicographic algorithm and hence we have a termination proof. The corresponding measure which decreases at every recursive call is:

$$(2 \times (size \circ \pi_2) + size \circ \pi_3 + 0 \times (size \circ \pi_4)) * lex * (\pi_0 \circ size).$$

Of course, this begs the question, how do we find such linear combinations? Furthermore, we note that we instead could have chosen the linear combination $x_2 + x_3 + 0x_4 = \begin{bmatrix} 0 \\ 0 \\ -2 \end{bmatrix}$, which would yield the matrix:

$$\begin{bmatrix} -1 & ? & 0 \\ ? & -1 & 0 \\ 0 & ? & -1 \end{bmatrix},$$

which is now no longer solvable. Hence, we need to be careful about our choice of linear combination here; we need it to be optimal in some sense. To find what this optimality condition might be, we begin by noting that the sum of any two of these positive rational linear combinations is itself a positive rational linear combination. This implies that to find such an optimal number linear combination, we just need to maximize the number of non-zero components of the linear combination. We call this problem of trying to find a linear combination with the maximal number of non-zero components the Maximal Negative Entries (MNE) problem.

Now, of course, this leads us to the natural question, how do we solve the MNE problem? Well, the maximal negative entries problem tells us to maximise the number of strictly negative entries of Ax such that $x \geq 0$ and $Ax \leq 0$. We can also frame this as wanting to find some $c \geq 0$ such that $Ax + c = 0$. The thing to notice here is that solutions to this problem are linear, in the sense that if (x, c) is a solution, i.e. that $Ax + c = 0$, then $A(ax) + ac = a(Ax + c) = 0$, and so (ax, ac) is also a solution for any positive a. The crucial thing to note here is that we can pump up the size of any such solution vector here to an arbitrary degree. This means that if we break up c into $b + z$, where b is bounded (say between 0 and 1), and z is unbounded, then we can always guarantee that each component of b is either 0 if there is a 0 in the corresponding component of c, or 1 otherwise. Hence, if we require that solutions maximise the sum of the components of b, this ought to give an encoding of our problem as a linear program.

Based on this reasoning, we have proven the MNE problem for a rational $n \times m$ matrix A equivalent to the following linear programming problem:

Maximise:

$$\sum_{i=0}^{n-1} b_i$$

subject to:

$$\begin{aligned} Ax + b + z &= 0 \\ b_i, z_i, x_i &\in \mathbb{Q} \\ 0 &\leq x_i \\ 0 &\leq b_i \leq 1 \\ 0 &\leq z_i \end{aligned}$$

This formulation allows us to use a linear solver to solve the MNE problem in polynomial time. Since the lexicographic algorithm also runs in polynomial time, this means our extended algorithm runs in polynomial time. We also provide a Haskell implementation of this procedure.

4 Results and Contributions

We have provided a way of efficiently solving the MNE problem, which gives us a termination algorithm. This algorithm takes in a function and first converts it into a size-change matrix. Then it finds a linear combination of all the numeric columns of this matrix which is a solution to the MNE problem, reduces using the lexicographic algorithm and repeats. We have proven that a function must terminate if we can reduce to the empty matrix using this procedure. As noted earlier, linearity of solutions to this linear program also implies that these solutions are unique up to scaling by positive rational numbers. This further implies that this termination

algorithm is deterministic. We've also proven that the algorithm will say a function with matrix M terminates if and only if every function with corresponding matrix M terminates. This further implies that given a function f with associated matrix M, if there exists some lexicographic order of linear combinations of the measures in M that decreases at every recursive call, then the algorithm will give a "terminates" result for f.

5 Conclusion

By extending the way size-change information is represented and the operations used to combine measures, we have improved the capability of the termination checking algorithm underlying Isabelle/HOL's termination checker. We have provided a Haskell implementation of this algorithm in the context of checking the termination of functional programs.

Acknowledgements

This work was done in collaboration with Vincent Jackson and Christine Rizkallah.

References

[1] Andreas Abel. 1998. foetus–termination checker for simple functional programs. *Programming Lab Report* 474 (1998).

[2] Andreas Abel and Thorsten Altenkirch. 2002. A predicative analysis of structural recursion. *Journal of functional programming* 12, 1 (2002), 1–41.

[3] Lukas Bulwahn, Alexander Krauss, and Tobias Nipkow. 2007. Finding lexicographic orders for termination proofs in Isabelle/HOL. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 38–53.

[4] Samir Genaim, Michael Codish, John Gallagher, and Vitaly Lagoon. 2002. Combining norms to prove termination. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 126–138.

[5] Eduarde Giménez. 1994. Codifying guarded definitions with recursive schemes. In *International Workshop on Types for Proofs and Programs*. Springer, 39–59.

[6] Alexander Krauss. 2007. Certified size-change termination. In *International Conference on Automated Deduction*. Springer, 460–475.

[7] Kirack Sohn and Allen Van Gelder. 1991. Termination detection in logic programs using argument sizes. In *Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 216–226.

[8] Franz Baader Tobias Nipkow. 2001. Term Rewriting and All That by Franz Baader and Tobias Nipkow, Cambridge University Press, 1998, ISBN 0-521-45520-0 (hardback), 301pp. *Journal of Functional Programming* 11, 2 (2001), 253–262.