T-Rex: Termination of Recursive Functions Using Lexicographic Linear Combinations

Raphael Douglas Giles
The University of Melbourne, Australia

Vincent Jackson
The University of Melbourne, Australia

Christine Rizkallah
The University of Melbourne, Australia

Abstract

We introduce a powerful termination algorithm for structurally recursive functions that improves on the core ideas behind lexicographic termination algorithms for functional programs. The algorithm generates linear-lexicographic combinations of primitive measure functions measuring the recursive structure of terms. We introduce a measure language that enables the simplification and comparison of measures and we prove meta-theoretic properties of our measure language. Moreover, we demonstrate our algorithm, on an untyped first-order functional language and prove its soundness and that it runs in polynomial time. We also provide a Haskell implementation. As part of this work, we also show how to solve the maximisation of negative vector-components as a linear program.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases Termination, Recursive functions

Digital Object Identifier 10.4230/LIPIcs.ICALP.2024.139

Category Track B: Automata, Logic, Semantics, and Theory of Programming

Supplementary Material
Software (Source Code): `https://github.com/vjackson725/term_check` [9]
archived at `swh:1:dir:c7dd8ada9bdd696f86cd11bc6751817ed37ad120`

1 Introduction

To guarantee the *total correctness* of a program, it is essential to prove that the program terminates [10]. While the halting problem is undecidable for general recursive functions [8, 21], there has been a long line of work on creating termination algorithms that automatically determine whether certain classes of functions terminate [14, 22, 11, 7, 2, 1].

In the context of functional programming, a termination algorithm typically takes a recursive function f as input and attempts to find a function from terms in the language to some well-founded order; this function is called a measure function. When the measure is applied to the arguments of f, it strictly decreases at each recursive call site [22, 11]. Such a measure can be as simple as a single argument to the function that always decreases at each recursive call. Such a simple measure is sufficient for demonstrating the termination of primitive recursive functions [18]. For example, consider the following function defining the binomial coefficient:

$$\begin{aligned} \mathit{choose}\ \langle 0, k\rangle &= 1 \\ \mathit{choose}\ \langle n, 0\rangle &= 1 \\ \mathit{choose}\ \langle \mathbf{S}\, n, \mathbf{S}\, k\rangle &= \mathit{choose}\, \langle n, \mathbf{S}\, k\rangle + \mathit{choose}\, \langle n, k\rangle \end{aligned}$$

To prove that *choose* terminates, we observe that the first projection of the argument, labelled n, strictly decreases at each recursive call site. Since n is a natural number and the natural numbers are well-ordered, we can conclude that *choose* must eventually terminate. Thus

© Raphael Douglas Giles, Vincent Jackson, and Christine Rizkallah;
licensed under Creative Commons License CC-BY 4.0

a decreasing measure of *choose* is $\mathbf{size} \circ \pi_1$. The measure of the argument to the *choose* function is the function $\mathbf{size} \circ \pi_1$, where the function π_1 projects the first argument out of the tuple and **size** returns the interpretation of object language number n in the underlying language or logic.

Of course, termination arguments are not always as simple as finding a single argument that decreases at every recursive call. Consider the following example:

$$\begin{array}{lcl} ex_1\,\langle 0, 0\rangle & = & 0 \\ ex_1\,\langle \mathbf{S}\,x, y\rangle & = & ex_1\ \langle x, \mathbf{S}\,y\rangle \\ ex_1\,\langle x, \mathbf{S}\,y\rangle & = & ex_1\ \langle x, y\rangle. \end{array}$$

In this example, no single part of the argument decreases at every recursive call. Hence, we cannot simply find a measure by considering each projection separately. However, this function terminates, and the argument for its termination can be generalised to determine the termination of a larger class of functions.

Note that the first projection of the argument never increases in any recursive call, and in the first recursive call, it strictly decreases. Hence the first recursive call can only be applied a finite number of times, bounded by the size of the first projection. With this fact, we may set the first recursive call aside. Observe that, considered alone, the second recursive call can also only be called a finite number of times, as the second projection of the argument always decreases. Thus, there can only be a finite number of recursive calls to this function overall. This idea is captured by the fact that the lexicographic order of the size of the first followed by that of the second argument decreases at each recursive call.

The Isabelle/HOL theorem prover [17] has a state-of-the-art termination algorithm that uses this idea for automatically detecting termination arguments [7, 2]. This more involved lexicographic order algorithm is quite powerful, covering termination arguments for various functions with non-trivial termination proofs, including the merge function on sorted lists and the Ackermann function [3]. However, it cannot handle the following function:

$$\begin{array}{lcl} ex_2\,\langle 0, 0\rangle & = & 0 \\ ex_2\,\langle \mathbf{S}\,(\mathbf{S}\,x), y\rangle & = & ex_2\ \langle x, \mathbf{S}\,y\rangle \\ ex_2\,\langle x, \mathbf{S}\,(\mathbf{S}\,y)\rangle & = & ex_2\ \langle \mathbf{S}\,x, y\rangle \end{array}$$

Though there is no lexicographic combination of generated measures that decreases at every recursive call, we know that this function must terminate because the sum of the arguments is always decreasing. More precisely, the measure $(\lambda t.\ (\mathbf{size} \circ \pi_1)\ t + (\mathbf{size} \circ \pi_2)\ t)$ decreases at every recursive call.

A termination algorithm that considers sums of the same generated measures as those of Isabelle/HOL would be successful in proving termination for ex_2. If we were to also consider linear combinations with coefficients in $\mathbb{N}$, it would also prove termination for ex_1, since we could take $(\lambda t.\ (2 \cdot ((\mathbf{size} \circ \pi_1)\ t)) + (\mathbf{size} \circ \pi_2)\ t)$.

The Isabelle/HOL algorithm can also not handle the following example, which converts from sparse lists to lists. Sparse lists provide a space-efficient representation of lists that contain many repeated elements. For example the list $[\mathtt{a}, \mathtt{a}, \mathtt{a}, \mathtt{h}, \mathtt{h}]$ would be represented as $[(\mathtt{a}, 3), (\mathtt{h}, 2)]$. We can more formally define the data type $\mathbf{SparseList}\ \alpha$ as inductively generated by elements $\mathbf{SNil}$ and $\mathbf{SCons}\ (x : \alpha)\ (n : \mathbb{N})\ (xs : \mathbf{SparseList}\ \alpha)$. The following function converts sparse lists to regular lists; defined through the usual **Nil** and **Cons** constructors:

$$\begin{array}{lcl} \mathit{toList}\ \mathbf{SNil} & = & \mathbf{Nil} \\ \mathit{toList}\ (\mathbf{SCons}\ x\ 0\ xs) & = & \mathit{toList}\ xs \\ \mathit{toList}\ (\mathbf{SCons}\ x\ (\mathbf{S}\,n)\ xs) & = & \mathbf{Cons}\ x\ (\mathit{toList}\ (\mathbf{SCons}\ x\ n\ xs)). \end{array}$$

$$
\begin{array}{lcl}
ack\,\langle 0, n\rangle & = & \mathbf{S}\,n \\
ack\,\langle \mathbf{S}\,m, 0\rangle & = & ack\,\langle m, \mathbf{S}\,0\rangle \\
ack\,\langle \mathbf{S}\,m, \mathbf{S}\,n\rangle & = & ack\,\langle m, ack\,\langle \mathbf{S}\,m, n\rangle\rangle
\end{array}
\qquad
\begin{array}{l}
\\
\text{call 1} \\
\text{call 2} \\
\text{call 3}
\end{array}
\begin{array}{c}
\mathbf{size} \circ \pi_1 \quad \mathbf{size} \circ \pi_2 \\
\left[\begin{array}{cc}
< & ? \\
< & ? \\
\leq & <
\end{array}\right]
\end{array}
$$

Figure 1 The Ackermann function is presented on the left and its difference matrix on the right.

This function terminates because either the number of **SCons** constructors decreases or it stays the same and the number n decreases. As this function has a lexicographic termination argument, we might expect the Isabelle/HOL termination algorithm to be able to handle it. The reason it cannot is that Isabelle/HOL only generates measures that consider the difference in the number of **SCons** constructors for each recursive function call – missing the fact that the number *inside* the **SCons** decreases.

We provide a novel algorithm, called T-Rex, that proves termination for a large class of functions including all the examples above. It first generates a set of measures based on structural size that is sufficiently detailed to handle examples such as *toList* where the decrease in structural size occurs within another data structure. It then determines whether there exists a lexicographic combination of these measures that decreases at every recursive call, or such an $\mathbb{N}$-linear combination, as well as complex measures combining the two (see ex_4). We demonstrate how to analyse the termination of functional programs by operating directly on and measuring the recursive structure of terms, without relying on the type structure or using higher-order logic and theories of term rewriting. This means that T-Rex can be used to analyse the termination of functional programs without relying on an underlying theorem prover, making it accessible to a wider community. We provide an implementation of our language and our T-Rex algorithm in Haskell.

2 Isabelle/HOL's Termination Algorithm

Isabelle/HOL's lexicographic termination algorithm [7] compares the measures of each argument to the function to that of each recursive call. For example, consider the Ackermann function (Figure 1). The Ackermann function has three recursive calls for which we need to show a decrease between the structural size of the initial and recursive arguments.

The size-change information for these calls is represented in a matrix, where each row corresponds to a recursive call and each column corresponds to a measure of part of the argument. The size-change comparison matrix for *ack* example (Figure 1) consists of the two columns that represent the size-change relation on the first and the second components of the argument tuple, respectively.

The entries record the change between the original and recursive arguments, recorded with the symbols $<$, $\leq$, and ?. The symbol $<$ means that there is a strict decrease in the size of the recursive call argument compared to the initial one, the symbol $\leq$ means that the size decreases or remains the same, and the symbol ? means that the size either increased or the relationship between the sizes is unknown.

From this point forward we omit the labels of the rows and columns of matrices, since these labels clutter the notation and as we wish to orient the reader slowly to thinking about these as matrices in the sense of linear algebra. This will ease introducing the ideas for our extension to the termination algorithm. Isabelle/HOL's lexicographic algorithm works by mimicking the informal argument given in the introduction. It attempts to find

arguments that always either decrease or stay the same size (i.e., that have $<$ and $\leq$ entries) and removes these recursive calls from consideration. Phrased in terms of the matrix, the algorithm repeatedly finds a column of the matrix with only $<$ and $\leq$ entries, and which must contain a $<$ entry, and removes every row of the matrix that has a $<$ entry in that column. If it eventually removes all the rows of the matrix, then the function must terminate.

To find the corresponding measure, it just keeps track of the measure associated with each column. Let m_1 be the measure associated with the first encountered column that only contains $<$ and $\leq$ entries, m_2 the measure associated with the second, and so on, up to the measure associated with the last such column m_n. The *lexicographic* measure, $[m_1, m_2, \ldots, m_n]_{\text{lex}}$, takes the measures m_1 to m_n, and combines their results into a lexicographically ordered tuple. Isabelle/HOL's lexicographic procedure produces a sequence of measures that, when combined lexicographically, decreases at every recursive call.

Returning to the Ackermann example, the sequence of row eliminations computed by the above algorithm is

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

where $\varnothing$ stands for the empty matrix. The corresponding measure computed by this procedure, that decreases at every recursive call is $[\mathbf{size} \circ \pi_1, \mathbf{size} \circ \pi_2]_{\text{lex}}$.

Limitations of the Isabelle/HOL algorithm

As noted earlier the Isabelle/HOL algorithm, while very effective for many problems, fails to prove termination for examples such as ex_2. We can see this directly by generating the corresponding size-change matrix for ex_2:

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

which cannot be reduced using the above procedure, since there is no column with only $<$ and $\leq$ entries. In order to extend this algorithm to prove the termination of such functions, we need more information in these matrices. The information we will use is the numeric increase or decrease of the measured size, when such a value is computable.

In this example, we can compute exactly how much each measure changes at each recursive call and we want to maintain a matrix with entries corresponding to those numeric size-changes, namely:

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

With this information, we can add the columns of this matrix, to produce a single column corresponding to the sum of these size changes, which in this case is negative in every entry, i.e. decreases at every recursive call, hence, proving termination.

This idea requires we solve three problems: how to generate such matrices, how to find such $\mathbb{N}$-linear combinations, and how to integrate this approach with the lexicographic algorithm. We first develop a language as well as a measure language in order to demonstrate how measure generation works. From there, the remainder of the paper is dedicated to solving these three problems.

Function names $f : N$ Variables $x, y, z : V$

$$\begin{array}{lll} \text{Terms } t \in T & ::= & \mathbf{var}\, x \mid \langle\rangle \mid \langle t, t\rangle \mid \mathbf{inl}\ t \mid \mathbf{inr}\ t \mid \mathbf{roll}\,\{t\} \mid \mathbf{app}\, f\, t \\ \text{Patterns } p \in P & ::= & \mathbf{var}\, x \mid \langle\rangle \mid \langle p, p\rangle \mid \mathbf{inl}\ p \mid \mathbf{inr}\ p \mid \mathbf{roll}\,\{p\} \end{array}$$

Function body $B = [(P \times T)]$ Program $\Gamma : [(N \times B)] ::= \overline{(f, b)}$

$0 = \mathbf{roll}\,\{\mathbf{inl}\,\langle\rangle\}$ $\mathbf{S}\, n = \mathbf{roll}\,\{\mathbf{inr}\ n\}$

Figure 2 Language Syntax.

3 Language

As our termination algorithm targets functional languages, we introduce a core calculus that supports common features of these languages, such as recursion, pattern matching, and standard algebraic data types. Standard ADTs can be encoded in our core language through the constructors that the language provides: recursive structures (roll), products, sums and unit. Our language is untyped, but would permit the addition of the standard type system for these constructs. Our language does not support lambda terms. The concrete syntax of our language is similar to Haskell function definitions. We provide examples in Sections 1, 2.

The syntax of our language is presented in Figure 2. A program is a sequence of function declarations, which are themselves a pair of a function name and function body. Each function body is a sequence of defining equations which are pairs of a pattern p and a term t. We will write the elements of a function body in the form $f\ p = t$, where f is the function name.

Each pattern has a corresponding term that it destructs on. Patterns contain binding variables $\mathbf{var}\, x, \mathbf{var}\, y, \mathbf{var}\, z$, written x, y, z for short, patterns for destructing sums $\mathbf{inl}\ p$ and $\mathbf{inr}\ p$, a pattern for destructing tuples $\langle p, p\rangle$, the pattern for destructing units $\langle\rangle$, and the pattern for destructing rolls $\mathbf{roll}\,\{p\}$. Terms include constructors for each of these as well as function application $\mathbf{app}\, f\, t$, written, $f\ t$ for short. When defining a function, terms describe how to build up the structure of the output using the variables bound by the input pattern.

4 Measure Language

We can prove that a function terminates by finding a measure of the size of the function's argument and showing that the argument's size decreases on every recursive call. This is called a *measure function*, and it is of type $T \to \mathcal{O}$ from the set of terms T to some well-ordered set $\mathcal{O}$. In particular, we use the set of natural numbers ($\mathbb{N}$) for primitive measures and linear combinations of primitive measures, and the set of lists of natural numbers ($[\mathbb{N}]$) under the lexicographic order for our overall lexicographic-linear measures.

There are five basic measures, **unroll**, **uninl**, **uninr**, π_1, and π_2. These deconstruct the basic datatypes, **roll**, **inl** , **inr** , and the left and right subterms of $\langle -, -\rangle$ respectively. The **unroll** destructor adds 1 to the return value, and **uninl** and **uninr** immediately terminate when passed the opposite constructors **inr** and **inl** respectively. In addition, there are two constant measures **case01** and **case10**, that attempts to match the input term as a sum, with **case01** returning 0 for **inl** and 1 for **inr** , and **case10** doing the opposite. There are two measure operators: $\triangleleft$ and **fix**. The measure operator $\triangleleft$ functionally composes two measures, and the **fix** measure operator repeatedly evaluates the given measure.

$$
\begin{array}{lcl}
\text{Measure destructor } m_{\mathrm{d}} & ::= & \mathbf{unroll} \mid \mathbf{uninl} \mid \mathbf{uninr} \mid \pi_1 \mid \pi_2 \\
\text{Recursive measure } m_{\mathrm{r}} & ::= & m_{\mathrm{d}} \mid m_{\mathrm{r}} \triangleleft m_{\mathrm{d}} \\
\text{Primitive measure } m & : & M \\
m & ::= & (\mathbf{fix}\, m_{\mathrm{r}}) \mid \mathbf{case01} \mid \mathbf{case10} \mid m \triangleleft m_{\mathrm{d}} \\
\text{Measure State } M_{st} & = & (M \times T \times \mathbb{N}) \uplus \mathbb{N}
\end{array}
$$

Measure Evaluation Semantics $(-)\Downarrow : (M \times T \times \mathbb{N}) \to M_{st}$

$$
\begin{array}{lcll}
((m \triangleleft \mathbf{unroll}), (\mathbf{roll}\, \{t\}), i)\Downarrow & = & (m, t, 1+i)\Downarrow & \\
((m \triangleleft \mathbf{uninl}), (\mathbf{inl}\ t), i)\Downarrow & = & (m, t, i)\Downarrow & \\
((m \triangleleft \mathbf{uninr}), (\mathbf{inr}\ t), i)\Downarrow & = & (m, t, i)\Downarrow & \\
((m \triangleleft \mathbf{uninr}), (\mathbf{inl}\ t), i)\Downarrow & = & i & \\
((m \triangleleft \mathbf{uninl}), (\mathbf{inr}\ t), i)\Downarrow & = & i & \\
((m \triangleleft \pi_1), \langle s, t\rangle, i)\Downarrow & = & (m, s, i)\Downarrow & \\
((m \triangleleft \pi_2), \langle s, t\rangle, i)\Downarrow & = & (m, t, i)\Downarrow & \\
(\mathbf{case01}, (\mathbf{inl}\ t), i)\Downarrow & = & i & \\
(\mathbf{case01}, (\mathbf{inr}\ t), i)\Downarrow & = & 1+i & \\
(\mathbf{case10}, (\mathbf{inl}\ t), i)\Downarrow & = & 1+i & \\
(\mathbf{case10}, (\mathbf{inr}\ t), i)\Downarrow & = & i & \\
((\mathbf{fix}\, m), t, i)\Downarrow & = & (((\mathbf{fix}\, m) \triangleleft m), t, i)\Downarrow & \\
(m, t, i)\Downarrow & = & (m, t, i) & \mathbf{otherwise}
\end{array}
$$

Measure Functional Semantics $[\![-]\!] : M \to (T \rightharpoonup \mathbb{N})$

$$[\![m]\!]\, t \quad = \quad (m, t, 0)\Downarrow$$

Note that this function is partial as the result of $\Downarrow$ is not always a natural number.

Figure 3 Measure language semantics.

Semantics

The big-step evaluation semantics of the measure language $(-)\Downarrow$, defined in Figure 3, takes as input a triple consisting of a term t in T, a measure m and a natural number. If the measure exhaustively destructures the term through the evaluation, the semantics returns a natural number, and if the evaluation is stuck, the semantics returns a triple of the same type as the input triple.

Note that the evaluation semantics is stuck when it encounters a term that is a variable, a function application, or the term and the measure have mismatched term constructors and measure destructors. The functional semantics abstracts from this case, only being defined for closed terms where the measure and term constructors match.

We will prove that our algorithm correctly demonstrates termination (when it succeeds in finding a measure) by showing that, for every initial argument a_{i} and every argument a_{r} to the recursive calls, $m\ a_{\mathrm{r}} < m\ a_{\mathrm{i}}$ [11]. Note that this method requires that a_{r} and a_{i} are *closed*, that is contain no variables or function calls. We will ensure this in our proofs by substituting variables for closed terms and functions for functions from closed terms to closed terms.

Bounded Difference

In general, we will not be able to determine the exact measure of a term statically, as measure execution can get stuck. Thus, we will need to approximate the difference between the partial evaluations of a measure. This approximation must be an upper bound on the difference between the measures (when applied to any substitution which results in a natural number).

Bounded difference $(\dot{-}) : M_{st} \to M_{st} \to \mathbb{Z} \uplus \{\omega\}$

$$
\begin{array}{rcll}
(m, s, i) & \dot{-} & (m, s, j) & = & i - j \\
i & \dot{-} & (m', t, j) & = & i - j \\
(m, s, i) & \dot{-} & j & = & \omega \\
(m, s, i) & \dot{-} & (m', t, j) & = & \omega \quad \textbf{otherwise}.
\end{array}
$$

Figure 4 Bounded difference.

The bounded difference (Figure 4), written $\dot{-}$, is a conservative estimate of the true difference between the size of a measure applied to two arguments. It has four cases, which depend on whether the measure fully executes and returns an integer, or whether the execution gets stuck, resulting in a measure-term-integer triple. In the first case, the value is exactly what the true subtraction would yield, as the s terms cancel. In the second case, we take the pessimistic assumption that t evaluates to a natural number which maximises the overall size of the difference; in this case, this occurs when t would evaluate to 0, so we may safely ignore the term. The third and fourth cases, we again assume the term s maximises the overall size. However, here s could evaluate to any natural number, and there is no finite bound. Thus in these cases, we return the error value ω, to denote that the subtraction is unboundedly large. We extend the order on integers to $\mathbb{Z} \uplus \{\omega\}$ by asserting that for all $n \in \mathbb{Z}$, $n \leq \omega$.

▶ **Lemma 1** (Difference is Bounded). *For all measures $m : M$, terms $t_1, t_2 : T$, and substitutions θ, where $[\![m]\!]\,\theta(t_1)$ and $[\![m]\!]\,\theta(t_2)$ are defined,*

$$[\![m]\!]\,\theta(t_1) - [\![m]\!]\,\theta(t_2) \;\leq\; (m, t_1, 0)\Downarrow \dot{-}\, (m, t_2, 0)\Downarrow.$$

5 Combining Measures

The termination of a function can be proven by finding a well-founded measure of the size of the arguments, such that the size of the arguments always decreases between the initial and recursive calls of the function.

Our termination algorithm builds measures from three components: *primitive measures*, *linear measures*, and *lexicographic measures*. Primitive measures are functions that transform a term into a natural number, representing a certain measure of the size of that term. We use "primitive measure" to refer both to the syntactic construct (M) and the corresponding mathematical functions ($T \to \mathbb{N}$).

A *linear combination of measures* is a weighted sum of measures. Given a vector of measures $\mathbf{m} : (\alpha \to \mathbb{N})^n$ and a vector of weights $\mathbf{w} : \mathbb{N}^n$ of the same length, we define

$$
\begin{array}{lcl}
(\cdot) & : & \mathbb{N}^n \to (\alpha \to \mathbb{N})^n \to (\alpha \to \mathbb{N}) \\
\mathbf{w} \cdot \mathbf{m} & = & (\lambda x.\ \sum_{i=1}^{m} \mathbf{w}_i \cdot \mathbf{m}_i(x)).
\end{array}
$$

In our termination algorithm, we will only take linear combinations of primitive measures, specialising the above to $\mathbb{N}^n \to (T \to \mathbb{N})^n \to (T \to \mathbb{N})$.

Since linear programming methods need to work in an ordered field, the tools we use will produce linear combinations of measures with positive *rational* coefficients. Finding such a linear combination is equivalent to finding a linear combination with natural coefficients, as the rational weights can be transformed to integers by multiplication of the least common multiple of the divisors of the weights. Note that the Haskell implementation of our algorithm uses fixed-point numbers, rather than exact arithmetic, which can lead to precision loss in this step.

6 Termination Algorithm

We want to show that our functions terminate, by showing that there is a measure of the arguments that decreases between every recursive call. We do this by combining primitive measures into linear combinations, and then making lexicographic combinations of these. Our termination algorithm is composed of three main phases:

1. Generating primitive measures of structural size for the sub-terms of the argument.
2. Generating the *primitive measure difference matrix*, that captures the size change between the initial and recursive arguments, as measured by the primitive measure functions.
3. Using the primitive measure difference matrix to find a lexicographic-linear combination of the primitive measure functions that decreases over every recursive call.

Finding lexicographic-linear combinations can be further broken down into two steps

i. Using the integer-only columns of the primitive measure difference matrix to find a linear combination of the primitive measure functions that decreases over the recursive calls that correspond to these columns.
ii. Using the linear combination procedure iteratively to determine a lexicographic combination of the linear measure functions that decreases over all recursive calls including those that do not have a simple integer difference.

To see how this algorithm works in the case of lexicographic termination, consider ex_3.

$$\begin{array}{lcl} ex_3\ \langle x, y, \mathbf{S}\, z\rangle & = & ex_3\ \langle x, y, z\rangle \\ ex_3\ \langle x, \mathbf{S}\, y, 0\rangle & = & ex_3\ \langle x, y, h\, y\rangle \\ ex_3\ \langle \mathbf{S}\, x, 0, 0\rangle & = & ex_3\ \langle x, h\, x, h\, x\rangle \end{array}$$

Note that, in this example h is some arbitrary function, and that we assume that any function called in a function definition (apart from the function being defined) terminates. That is, we show termination conditional on all called functions also terminating. However, when proving termination, we do not use any knowledge about the behaviour of called functions; so we have no bounds on the value that h will return. The primitive measures for ex_3 and the resulting difference matrix are

$$\begin{array}{l} m_1 = (\mathbf{fix}\,(\mathbf{uninr} \triangleleft \mathbf{unroll})) \triangleleft \pi_1 \\ m_2 = (\mathbf{fix}\,(\mathbf{uninr} \triangleleft \mathbf{unroll})) \triangleleft \pi_2 \\ m_3 = (\mathbf{fix}\,(\mathbf{uninr} \triangleleft \mathbf{unroll})) \triangleleft \pi_3 \end{array} \qquad \begin{bmatrix} 0 & 0 & -1 \\ 0 & -1 & \omega \\ -1 & \omega & \omega \end{bmatrix}.$$

The value ω marks the worst-case value difference. The function h is assumed to terminate but may return a term of arbitrary size; thus the maximum bound of the change in value cannot be any integer. Note that ω prevents any column containing it from being used in a linear programming problem.

The ex_3 function has a lexicographic termination argument, as m_1 decreases or stays the same everywhere, m_2 decreases or stays the same when m_1 stays the same, and m_3 decreases when m_1 and m_2 stay the same. By the lexicographic elimination algorithm,

$$\begin{bmatrix} 0 & 0 & -1 \\ 0 & -1 & \omega \\ -1 & \omega & \omega \end{bmatrix} \rightsquigarrow \begin{bmatrix} 0 & -1 & \omega \\ -1 & \omega & \omega \end{bmatrix} \rightsquigarrow \begin{bmatrix} 0 & -1 & \omega \end{bmatrix} \rightsquigarrow \varnothing.$$

Linear and lexicographic termination arguments can be combined together to find a linear-lexicographic termination measure. To illustrate this, consider the function ex_4:

$$\begin{array}{lcl} ex_4\ \langle \mathbf{S}\, x, y, z, v, w\rangle & = & ex_4\ \langle x, h\, y, z, v, \mathbf{S}\, w\rangle \\ ex_4\ \langle x, \mathbf{S}\, y, \mathbf{S}\, z, v, w\rangle & = & ex_4\ \langle h\, x, y, z, \mathbf{S}\, v, \mathbf{S}\, w\rangle \\ ex_4\ \langle x, y, z, \mathbf{S}\,\mathbf{S}\,\mathbf{S}\, v, w\rangle & = & ex_4\ \langle x, h\, y, \mathbf{S}\, z, v, \mathbf{S}\, w\rangle \end{array}$$

where h represents an arbitrary function. The following is the difference matrix for ex_4:

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

Solving it requires both the linear and lexicographic aspects of our approach. This is again a situation where the pure lexicographic algorithm is not applicable. Moreover, in this case, one cannot find a positive rational linear combination of all the columns that results in a vector that is strictly less than 0 in all its entries. Using both ideas, we want to find some linear combination of the last three columns in the matrix, so as to eliminate some of the ω entries. Fortunately, it is not difficult to see that, labelling the columns of the matrix $c_1, c_2, \ldots, c_5$, one can take $2c_3 + c_4 + 0c_5 = (0 \ -1 \ -1)^\mathsf{T}$. (Where $(-)^\mathsf{T}$ is vector/matrix transposition.) This yields the following matrix

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

We now use the lexicographic algorithm to remove the last column and bottom-most two rows, resulting in the matrix $\begin{bmatrix} -1 & \omega \end{bmatrix}$, and then to reduce this matrix to $\varnothing$. Thus we have obtained a termination proof. The corresponding measure that decreases at every recursive call is $[(2 \cdot (\mathbf{size} \triangleleft \pi_3) + (\mathbf{size} \triangleleft \pi_4)), (\mathbf{size} \triangleleft \pi_1)]_{\text{lex}}$.

6.1 Primitive Measure Generation

Each term has several primitive measures, which are generated by observing how that term is destructured in the definition of the recursive function. These primitive measures are made of a composition of two parts: a function which descends through the non-recursive part of a term to extract the recursively constructed term, and a measure of that recursive term.

For example, take the two terms $\langle \mathbf{roll}\,\{\mathbf{inr}\ x\}, \langle\rangle\rangle$ and $\langle x, \langle\rangle\rangle$, and assume the former is the input argument to a function, and the second is the recursive argument. We can clearly see that the structural size of the left part of the tuple decreases from input to recursive call, due to the removal of the **roll**. We capture this intuition with the primitive measure function $(\mathbf{fix}\,(\mathbf{uninr} \triangleleft \mathbf{unroll})) \triangleleft \pi_1$. The purpose of this measure is to extract the recursive part of the argument, i.e. to remove the $\langle -, \langle\rangle\rangle$ structure, and then to count the number of nestings of the structure $\mathbf{roll}\,\{\mathbf{inr}\ -\}$.

Primitive measure generation is composed of two parts: generating the recursive part of the measure and generating the function that extracts the recursive subterm. The function $primM_{\text{R}}$ generates the recursive parts of the primitive measures. It takes a variable name x, a partially constructed recursive measure m_r, and a term t. The measure is initialised with id for notational convenience where $m \triangleright \text{id} = m$; in our implementation we use lists of measure destructors.

The recursive measures are generated by the function

$$
\begin{array}{lcl}
primM_{\mathrm{R}}(x, m_r, \mathbf{var}\, y) & = & \begin{cases} \{\mathbf{fix}\, m_r\} & \mathbf{if}\ x = y \wedge m_r \neq \mathrm{id} \\ \emptyset & \mathbf{if}\ x \neq y \vee m_r = \mathrm{id} \end{cases} \\
primM_{\mathrm{R}}(x, m_r, \langle\rangle) & = & \emptyset \\
primM_{\mathrm{R}}(x, m_r, \langle t_1, t_2\rangle) & = & primM_{\mathrm{R}}(x, (\pi_1 \triangleleft m_r), t_1) \\
 & & \cup\, primM_{\mathrm{R}}(x, (\pi_2 \triangleleft m_r), t_2) \\
primM_{\mathrm{R}}(x, m_r, \mathbf{inl}\ t) & = & primM_{\mathrm{R}}(x, \mathbf{uninl} \triangleleft m_r, t) \\
primM_{\mathrm{R}}(x, m_r, \mathbf{inr}\ t) & = & primM_{\mathrm{R}}(x, \mathbf{uninr} \triangleleft m_r, t) \\
primM_{\mathrm{R}}(x, m_r, \mathbf{roll}\,\{t\}) & = & primM_{\mathrm{R}}(x, \mathbf{unroll} \triangleleft m_r, t) \\
primM_{\mathrm{R}}(x, m_r, \mathbf{app}\, f\, t) & = & \emptyset.
\end{array}
$$

Consider the example above comparing x and $\mathbf{roll}\,\{\mathbf{inl}\ x\}$. As expected, the generator produces the set $\{\mathbf{fix}\,(\mathbf{uninl} \triangleleft \mathbf{unroll})\}$. A more complex example is comparing x and $\mathbf{roll}\,\{\langle x, x\rangle\}$; the measure set generated is $\{\mathbf{fix}\,(\pi_1 \triangleleft \mathbf{unroll}), \mathbf{fix}\,(\pi_2 \triangleleft \mathbf{unroll})\}$ with a generated measure for each occurrence of x.

We can now generate a set of measures for recursive terms. However, we still need to generate the part of the measure that extracts the subterm where the recursion occurs. This part of the measure is appended to the recursive measures generated by calling $primM_{\mathrm{R}}$. It is constructed by recursion on both the initial and recursive arguments.

$$
\begin{array}{lcl}
primM(m, \mathbf{var}\, x, u) & = & \{r \triangleleft m \mid r \in primM_{\mathrm{R}}(x, \mathrm{id}, u)\} \\
primM(m, t, \mathbf{var}\, y) & = & \{r \triangleleft m \mid r \in primM_{\mathrm{R}}(y, \mathrm{id}, t)\} \\
primM(m, \langle\rangle, \langle\rangle) & = & \emptyset \\
primM(m, \langle t_1, t_2\rangle, \langle u_1, u_2\rangle) & = & primM(\pi_1 \triangleleft m, t_1, u_1) \\
 & & \cup\, primM(\pi_2 \triangleleft m, t_2, u_2) \\
primM(m, \mathbf{inl}\ t, \mathbf{inl}\ u) & = & primM(\mathbf{uninl} \triangleleft m, t, u) \\
 & & \cup\, \{\mathbf{case01} \triangleleft m, \mathbf{case10} \triangleleft m\} \\
primM(m, \mathbf{inr}\ t, \mathbf{inr}\ u) & = & primM(\mathbf{uninr} \triangleleft m, t, u) \\
 & & \cup\, \{\mathbf{case01} \triangleleft m, \mathbf{case10} \triangleleft m\} \\
primM(m, \mathbf{roll}\,\{t\}, \mathbf{roll}\,\{u\}) & = & primM(\mathbf{unroll} \triangleleft m, t, u) \\
primM(m, t, u) & = & \emptyset \qquad\qquad \mathbf{otherwise}
\end{array}
$$

Note that sums, in addition to the measures of recursive size-change of the components, also have measures to detect when a sum switches from left to right (or vice-versa). Due to these measures and the fact we are finding linear combinations of primitive measures, we do not need to generate quadratically many measures for the combinations of the left and right submeasures, as done by others [7].

To generate the primitive measures for a function f, we first apply *primM* to every initial and recursive argument pair in the equations defining f, initialising m with id. For each measure m generated through this process, if m is of the form $(\mathbf{fix}\ \ (m' \triangleleft \cdots \triangleleft m')) \triangleleft m''$, with the m' repeated, we simplify it to $(\mathbf{fix}\ \ m') \triangleleft m''$. This step is useful in cases where, for example, a function removes the same pattern multiple times; like ex_2, which removes multiple **S** constructors. This simplification step ensures the simplified measure function only removes one of these patterns at a time. The simplified measure returns the same size as $[\![m]\!]$ when $[\![m]\!]$ is defined, and it is defined for more terms; this property follows from the definition of the evaluation function.

With this method for primitive measure generation in hand, we return to the *toList* example from Section 1. To demonstrate how this measure generation works, we first need to unfold the SparseList and List constructors in *toList* according to Figure 5,

$$\begin{array}{llll} \mathbf{Nil} & = \mathbf{roll}\,\{\mathbf{inl}\,\langle\rangle\} & \mathbf{SNil} & = \mathbf{roll}\,\{\mathbf{inl}\,\langle\rangle\} \\ \mathbf{Cons}\ x\ xs & = \mathbf{roll}\,\{\mathbf{inr}\,\langle x, xs\rangle\} & \mathbf{SCons}\ x\ n\ xs & = \mathbf{roll}\,\{\mathbf{inr}\,\langle x, \langle n, xs\rangle\rangle\} \end{array}$$

Figure 5 Encodings of lists and sparse lists.

$$\begin{array}{lcl} \mathit{toList}\ (\mathbf{roll}\,\{\mathbf{inl}\,\langle\rangle\}) & = & \mathbf{roll}\,\{\mathbf{inl}\,\langle\rangle\} \\ \mathit{toList}\ (\mathbf{roll}\,\{\mathbf{inr}\,\langle x, \langle \mathbf{roll}\,\{\mathbf{inl}\,\langle\rangle\}, xs\rangle\rangle\}) & = & \mathit{toList}\ xs \\ \mathit{toList}\ (\mathbf{roll}\,\{\mathbf{inr}\,\langle x, \langle \mathbf{roll}\,\{\mathbf{inr}\ n\}, xs\rangle\rangle\}) & = & \mathbf{roll}\,\{\mathbf{inr}\,\langle x, \langle \mathit{toList}\ (\mathbf{roll}\,\{\mathbf{inr}\,\langle x, \langle n, xs\rangle\rangle\})\rangle\rangle\}. \end{array}$$

The primitive measure generation algorithm produces measures corresponding to the recursive uses of both n and xs. The two measures, along with their difference matrix, are

$$\begin{array}{ll} m_0 = \mathbf{fix}\,(\pi_2 \triangleleft \pi_2 \triangleleft \mathbf{uninr} \triangleleft \mathbf{unroll}) & \\ m_1 = \mathbf{fix}\,(\mathbf{uninr} \triangleleft \mathbf{unroll}\,) \triangleleft (\pi_1 \triangleleft \pi_2 \triangleleft \mathbf{uninr} \triangleleft \mathbf{unroll}) & \begin{bmatrix} -1 & \omega \\ 0 & -1. \end{bmatrix} \end{array}$$

These can then be combined into an overall decreasing measure as $[m_0, m_1]_{\text{lex}}$.

6.2 Primitive Measure Difference Matrix

The fundamental data structure used in our termination-checking algorithm, T-Rex, is the *primitive measure difference matrix*, $D : (\mathbb{Z} \uplus \{\omega\})^{n \times k}$, or *difference matrix* for short. Rows in D represent argument-pairs and columns represent primitive measures. The elements are the conservative size change between the initial argument and the recursive argument. Formally, for any function definition (with n recursive calls), let the recursive call matrix of that function be $R : (T)^{n \times 2}$ where R_{i1} is the pattern of the i-th recursive call (lifted to a term) and R_{i2} is the recursive argument term of the i-th recursive call. We define the difference matrix D of a vector of k primitive measures, $\mathbf{m} : M^k$, as $D_{ij} = (\mathbf{m}_j, R_{i2}, 0)\Downarrow \dot{-}\, (\mathbf{m}_j, R_{i1}, 0)\Downarrow$ where $1 \le i \le n$ and $1 \le j \le k$. This matrix captures the size-change information of the measures $\mathbf{m}$ on every recursive call in the function. Note that it uses the upper bound difference, $(\dot{-})$, and so the estimated difference can be unbounded, i.e. ω.

Returning to the function ex_2, the entries of the difference matrix evaluate to

$$\begin{array}{llllll} m_{\mathbf{S}}\ x & \dot{-} & (m_{\mathbf{S}}\ x + 2) & = -2 & \text{and} & \\ (m_{\mathbf{S}}\ x + 1) & \dot{-} & m_{\mathbf{S}}\ x & = 1 & & \end{array} \qquad \begin{array}{llll} m_{\mathbf{S}}\ y & \dot{-} & (m_{\mathbf{S}}\ y + 2) & = -2 \\ (m_{\mathbf{S}}\ y + 1) & \dot{-} & m_{\mathbf{S}}\ y & = 1 \end{array}$$

where $m_{\mathbf{S}} = \mathbf{fix}\,(\mathbf{uninr} \triangleleft \mathbf{unroll})$,

This results in the following difference matrix:

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

In example ex_3, the arbitrary functions result in the measures becoming incomparable, resulting in ω. Consider the reduced differences that do not result in 0,

$$\begin{array}{llll} m_{\mathbf{S}}\ y & \dot{-} & (m_{\mathbf{S}}\ y + 1) & = -1 \\ m_{\mathbf{S}}\ (h\ y) & \dot{-} & 1 & = \omega \\ m_{\mathbf{S}}\ (h\ x) & \dot{-} & 1 & = \omega \end{array}$$

where $m_{\mathbf{S}} = \mathbf{fix}\,(\mathbf{uninr} \triangleleft \mathbf{unroll})$.

When the measure of the recursive argument, on the left-hand side, is not a concrete integer (e.g. $m_{\mathbf{S}}\ (h\ y)$), but the right hand side is, we can only pessimistically conclude that the value could be any large natural number. Thus we return ω, to mark that we cannot handle this case using linear arithmetic.

6.3 Finding Linear Measures

We wish to use our primitive measures to create better measures; that is, measures that decrease on more recursive calls. We will first consider *linear* combinations of measures. As we discussed in the last section, the difference matrix captures the size change information of the arguments as they change between recursive calls. This information helps determine weights that can be used to construct a linear combination of the primitive measures. Note that this phase only considers D matrices that contain no ω values. This enables applying standard linear solvers. The ω values are handled at a later phase.

To find suitable weights, first we will show that *any* vector of positive weights $\mathbf{w} : (\mathbb{Z}^+)^n$ that satisfies the equation $D\mathbf{w} \leq 0$ produces a non-increasing measure. Secondly, we will determine how to pick the *best* such vector: that is, the one that produces a measure that *strictly* decreases between the arguments of as many recursive calls as possible.

Decreasing linear combinations

The following lemma shows that any non-positive solution to $D\mathbf{w}$ can be used to construct a non-increasing measure.

▶ **Lemma 2** (Soundness of linear measure construction). *Given a vector of positive weights $\mathbf{w} : (\mathbb{Z}^+)^n$, a vector of measures $\mathbf{m} : M^n$, if there are no ω values in D and $(D\mathbf{w})_i \leq 0$ for row i, then for each initial-recursive argument pair R_i, and all substitutions θ where, for every j, $[\![\mathbf{m}_j]\!]\ \theta(R_{i1})$ and $[\![\mathbf{m}_j]\!]\ \theta(R_{i2})$ are defined, then $(\mathbf{w} \cdot [\![\mathbf{m}]\!])\ \theta(R_{i2}) - (\mathbf{w} \cdot [\![\mathbf{m}]\!])\ \theta(R_{i1}) \leq 0$. Similarly, if $(D\mathbf{w})_i < 0$, then $(\mathbf{w} \cdot [\![\mathbf{m}]\!])\ \theta(R_{k2}) - (\mathbf{w} \cdot [\![\mathbf{m}]\!])\ \theta(R_{k1}) < 0$. (Where $[\![-]\!]$ is lifted pointwise.)*

To give an example, we return to the function ex_2, which has the difference matrix

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

The weight vector $\mathbf{w} = (1\ 1)^\mathsf{T}$ produces the all-negative vector $D\mathbf{w} = (-1\ -1)^\mathsf{T}$. As such, we can conclude that the measure $(1\ 1)^\mathsf{T} \cdot (m_1\ m_2)^\mathsf{T}$ guarantees that ex_2 terminates.

The maximal negative entries problem

This problem is concerned with finding the best weights $\mathbf{w}$, that ensure that the maximum number of recursive calls decrease. To see why not just any weight will do, consider again the example ex_4. We showed that its matrix,

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

was solvable using our proposed algorithm. However, had we chosen instead the linear combination $(1\ 1\ 0)^\mathsf{T}$, this would have resulted in the output vector $(0\ 0\ -2)^\mathsf{T}$, which only removes the final recursive call / row, resulting in the unsolvable subproblem:

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

Thus, we want to pick weights such that not only $D\mathbf{w} \leq 0$, but also that $D\mathbf{w}$ has as many negative entries as possible. We call this the *maximal negative entries* (MNE) problem.

▶ **Definition 3** (The Maximal Negative Entries Problem). *For a matrix A, find a vector $\mathbf{x}$ such that the vector $A\mathbf{x}$ has a maximal number of negative entries.*

Reducing the MNE problem to a linear program

We want to find a programmatic method to solve this problem. This problem can be reduced to solving a linear program. This is fortunate as linear programs are well-studied and they are solvable in polynomial time [5].

The MNE problem tells us to maximise the number of strictly negative entries of $A\mathbf{x}$, such that $\mathbf{x} \geq 0$ and $A\mathbf{x} \leq 0$. We can reframe this as finding some slack vector $\mathbf{c} \geq 0$, such that $A\mathbf{x} + \mathbf{c} = 0$ and $\mathbf{c}$ has a maximal number of *positive* entries (as $A\mathbf{x} = -\mathbf{c}$).

Secondly, note that solutions to this problem are linear, in the sense that if $(\mathbf{x}, \mathbf{c})$ is a solution to $A\mathbf{x} + \mathbf{c} = 0$ and $\mathbf{c}$ has maximal positive entries, then, for a positive scalar k, $A(k\mathbf{x}) + k\mathbf{c} = k(A\mathbf{x} + \mathbf{c}) = 0$, and so $(k\mathbf{x}, k\mathbf{c})$ is also a solution. The crucial thing to note here is that we can scale up or down the size of any such solution vector by any positive scaling factor. This means that a solution exists exactly when a solution exists that satisfies the additional condition that $\mathbf{c}$ has entries that are either exactly 0 or greater than or equal to 1. This allows us to take the final step in our transformation of this problem.

Break up $\mathbf{c}$ (with entries now guaranteed to be 0 or $\geq$1) into $\mathbf{b} + \mathbf{z}$, where b is bounded between 0 and 1 and $0 \leq z$, The optimisation goal "maximise the sum of $\mathbf{b}$" will produce a solution $(\mathbf{x}, \mathbf{b}, \mathbf{z})$ where $\mathbf{b}$ has entries that are either 0 or 1 (as $\mathbf{b} + \mathbf{z}$ is 0 or ≥ 1) and $\mathbf{z}$ is the slack necessary to make the sum cancel. Note that $\mathbf{b}_i$ is 1 exactly when $(A\mathbf{x})_i < 0$, thus the number of negative entries in $A\mathbf{x}$ is equal to the sum of $\mathbf{b}$. As the sum of b is maximised, the number of negative entries in $A\mathbf{x}$ is maximised.

Lastly, as $\mathbf{z}$ is only constrained to be ≥ 0, $\mathbf{z}$ is a slack vector of this new program. We may simplify the program by removing $\mathbf{z}$ and turning the equality constraints to inequality constraints. Thus, the maximal negative entries problem is equivalent to solving the linear programming problem specified in Algorithm 1. Note that the program *fails* to find a satisfactory solution for our purposes when it returns the all-zero vector.

Algorithm 1 LinMNE: A linear program for the MNE problem.

$$
\begin{array}{rcl}
\textbf{maximise} & \sum_{i=0}^{n-1} \mathbf{b}_i & \\
\textbf{subject to} & & \\
\mathbf{x}_i, \mathbf{b}_i & \in & \mathbb{Q} \\
A\mathbf{x} + \mathbf{b} & \leq & 0 \\
0 & \leq & \mathbf{x}_i \\
0 & \leq & \mathbf{b}_i \leq 1
\end{array}
$$

Uniqueness of the maximal negative entry set

Even though the above program will find a set with a maximal number of negative entries, we have not yet established that such a set is unique. If there are two solutions that identify a different set of rows that decrease, this may lead to a different order of row elimination in the lexicographic stage of our algorithm. Such a possibility threatens the complexity and completeness of our algorithm. However, there is, in fact, a *unique* maximal set of negative entries that solves the linear program. Note that this uniqueness is in the set of entries, not in the solution; there can be many solutions, but all of them will show the *same* set of recursive calls decrease.

▶ **Theorem 4** (Unique Maximal Negative Entries). *For every matrix $A \in \mathbb{Q}^{n \times m}$, if $\langle \mathbf{x}, \mathbf{b} \rangle$ and $\langle \mathbf{x}', \mathbf{b}' \rangle$ are solutions of the MNE problem for A, then $\mathbf{b} = \mathbf{b}'$.*

6.4 Finding Lexicographic Measures of Linear Combinations

The lexicographic phase of the algorithm solves the termination problem for a matrix of weights that may include ω; the value that marks that a finite bound on the size change could not be found. The algorithm finds a lexicographic combination of linear measures, such that whenever a linear measure was approximated to ω, there is a lexicographically larger linear measure that returns a natural number. This lexicographic algorithm follows the same structure as Isabelle/HOL's [7]. Our overall **T-Rex** algorithm proceeds as follows:

Algorithm 2 The T-Rex Termination Algorithm.

function T-Rex($m : [M], D : \text{Matrix } (\mathbb{Z} \uplus \omega)) : [[\mathbb{N} \times M]] \uplus$ **fail**
  (Precondition: length(m) = columnN(D).)
  $out \coloneqq []$
  repeat
    $N \coloneqq$ Extract the purely numeric ($\mathbb{Z}$) columns of D
    if $N = \varnothing$ **then return fail end if** ▷ (if there are no purely numeric columns, fail)
    $(x, b) \coloneqq$ LinMNE(N)
    $x \coloneqq$ select non-zero weights in x
    if $x = \varnothing$ **then return fail end if** ▷ (if no measures were selected, fail)
    $x \coloneqq x \cdot \text{lcm}(\text{denoms}(w))$ ▷ (normalise the weights to natural numbers)
    $D \coloneqq D$ without rows where b is 1
    (Note: recall the linear program establishes that $(D\mathbf{x})_i$ is negative when $\mathbf{b}_i = 1$.)
    $wm \coloneqq$ weights in x paired with their corresponding measures in m
    $out \coloneqq out ::_\text{r} wm$ ▷ ($::_\text{r}$ is concatenate to end)
  until $M = \varnothing$
  return out
end function
(Note: we elide the tracking of column indices for clarity.)

Since linear program solving is polynomial time [5] and the number of loops performed is bounded by the number of rows in D, we can deduce that our algorithm also runs in polynomial time. But of course, we would like to know slightly more than just the correctness and complexity of our algorithm. Our algorithm is complete in the following sense. If a lexicographic-linear combination of primitive measures exists, our algorithm decides termination.

▶ **Lemma 5.** *Let f be a function and let A be a matrix whose columns correspond to some set of measures P whose elements we allow to be any $\mathbb{N}$-linear combination of the columns of the difference matrix of f. Then, there exists a lexicographic combination of the elements of P which decreases at every recursive call of f if and only if we can reduce A to $\emptyset$ by removing rows per the lexicographic algorithm.*

▶ **Theorem 6** (Decision Power of T-Rex)**.** *Given a function f with an associated set of primitive measures P, if there exists some lexicographic order of $\mathbb{N}$-linear combinations of elements of P which decreases at every recursive call of f, then **T-Rex** will succeed.*

7 Related Work

There is a relevant body of work in the context of analysing the termination of imperative programs called synthesis of ranking functions. This was initially introduced by Floyd [10], and later incorporated into Hoare logic to allow for proving the total correctness of imperative programs. Ranking functions are directly analogous to measures. In the context of imperative programs, ranking functions map variables that are updated in a loop body to a well-ordered set, and to prove termination they must decrease at each iteration. Typically in this context, the termination of a single loop with a single branch is studied, which broadly corresponds in our context to studying functions with a single recursive call.

A line of work on termination analysis is concerned with finding classes of programs (e.g. programs that operate on reals) with *linear loops* for which termination is decidable, by relating this problem to stability in control theory [20]. Part of the linear loops literature is concerned with synthesising ranking functions for linear-constraint loops [6, 19]. Here, we have a set of variables, a loop guard describing a linear constraint on these variables, and at each iteration, we have updated our vector of variables $\mathbf{x}$ by an affine transformation $A\mathbf{x} + \mathbf{b}$ [13]. (These affine transformations can always be converted to a linear transformation $A'\mathbf{x}$ by adding more variables.) The kind of ranking functions synthesised for these problems are similar to the lexicographic combinations of linear transformations that we see in our work [6]. One main distinction is that in our work, there is almost always more than one recursive call, and in their work, there is almost always some non-trivial condition on the loop guard, providing different challenges and resulting in different algorithms for handling termination within different programming language paradigms.

Linear and lexicographic termination techniques have also been explored in the context of probabilistic imperative programming. Similar algorithms to the measure combination step of the algorithm, utilising linear programming to combine size-change information, have been independently developed in this context [4]. However, a direct comparison between them presents a non-trivial challenge as this work operates on generalised transition systems generated from analysis of imperative code, while our work operates on primitive measures. Even given these similarities, our improved method of primitive measure generation handles cases that the Isabelle/HOL algorithm does not, even with the purely lexicographic solver.

Our work is directly inspired by and most closely related to Isabelle/HOL's lexicographic termination algorithm [2, 7]. It constructs termination matrices based on the comparison of arguments between the initial and recursive calls, as we do, but they do not compute a numeric difference. Since this approach ignores the numerical difference and conflates increase with uncertainty, it cannot be used to find linear combinations of measures. While our matrices carry more information, our lexicographic matrix elimination algorithm is structurally similar to Bulwahn et. al.'s lexicographic matrix elimination algorithm [7]. Note that we do not drop columns, as this can cause unsoundness if the final column is not wholly decreasing. Our algorithm also works without access to types or higher-order logic simplification theories. Hence, it is extensible to various functional languages and can be used without theorem-proving experience.

Another approach is the size-change termination method [16, 12], which is orthogonal to our method, as it tracks the size change of *data* (i.e. the constructors around variables) in function arguments, whereas we track the size change of the function arguments, disregarding the flow of variables. Types can be used to restrict functions to use terminating recursion [1], but this method requires user annotation, and cannot find complex linear termination measures.

Other approaches for functional termination checking involve using an external term-rewriting checker to show the termination of function programs [15]. This has the advantage of bringing well-developed termination checkers for term rewriting systems to functional termination checking, but requires that the semantics of the functional language be reflected in a term rewriting system. Further, any witness of termination must be translated backwards through this semantics. Our approach gives simple measure functions as witnesses.

8 Conclusion

We have developed T-Rex: a novel termination algorithm for recursive functions. It can prove the termination of functions by lexicographically ordering linear combinations of primitive measures that decrease at every recursive call. The primitive measures are computed directly

by examining the structure of the recursive program. We define a language for simplifying and comparing primitive measures that are used as part of our termination algorithm and prove meta-theoretic properties of our measure language. We prove that T-Rex is sound, that it runs in polynomial time and that it covers a large class of programs and demonstrate the algorithm on an untyped first-order functional language. We provide an implementation of the untyped language, measure language and of T-Rex in Haskell.

References

1 Andreas Abel. Termination checking with types. *RAIRO - Theoretical Informatics and Applications*, 38(4):277–319, 2004. doi:10.1051/ita:2004015.

2 Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, 2002. doi:10.1017/S0956796801004191.

3 Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99:118–133, 1928. URL: https://api.semanticscholar.org/CorpusID:123431274.

4 Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotnỳ. Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–32, 2017.

5 Bengt Aspvall and Richard E Stone. Khachiyan's linear programming algorithm. *Journal of Algorithms*, 1(1):1–13, 1980. doi:10.1016/0196-6774(80)90002-4.

6 Amir M Ben-Amram and Samir Genaim. Ranking functions for linear-constraint loops. *Journal of the ACM (JACM)*, 61(4):1–55, 2014.

7 Lukas Bulwahn, Alexander Krauss, and Tobias Nipkow. Finding Lexicographic Orders for Termination Proofs in Isabelle/HOL. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics*, volume 4732, pages 38–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISSN: 0302-9743, 1611-3349 Series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-540-74591-4_5.

8 Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:354–363, 1936.

9 Raphael Douglas Giles and Vincent Jackson. T-Rex. Software, version 1.0.0., swhId: swh:1:dir:c7dd8ada9bdd696f86cd11bc6751817ed37ad120 (visited on 2024-06-18). URL: https://github.com/vjackson725/term_check.

10 R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical aspects of computer science: Proc. American Mathematics Soc. symposia*, volume 19, pages 19–31, Providence RI, 1967. American Mathematical Society.

11 Jürgen Giesl. Automated termination proofs with measure functions. In Ipke Wachsmuth, Claus-Rainer Rollinger, and Wilfried Brauer, editors, *KI-95: Advances in Artificial Intelligence*, pages 149–160, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. doi:10.1007/3-540-60343-3_33.

12 Pierre Hyvernat. The size-change termination principle for constructor based languages. *Log. Methods Comput. Sci.*, 10(1), 2014. doi:10.2168/LMCS-10(1:11)2014.

13 Toghrul Karimov, Engel Lefaucheux, Joël Ouaknine, David Purser, Anton Varonka, Markus A Whiteland, and James Worrell. What's decidable about linear loops? *Proceedings of the ACM on Programming Languages*, 6(POPL):1–25, 2022.

14 Shmuel M. Katz and Zohar Manna. A closer look at termination. *Acta Informatica*, 5(4):333–352, December 1975.

15 Alexander Krauss, Christian Sternagel, René Thiemann, Carsten Fuhs, and Jürgen Giesl. Termination of Isabelle functions via termination of rewriting. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving*, pages 152–167, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-22863-6_13.

16 Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 81–92, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/360204.360210.

17 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer, 2002.

18 Rozsa Peter. Über die verallgemeinerung der theorie der rekursiven funktionen für abstrakte mengen geeigneter struktur als definitionsbereiche. *Acta Mathematica Academiae Scientiarum Hungaricae*, 12:271–314, 1962. URL: https://api.semanticscholar.org/CorpusID:121988998.

19 Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 239–251. Springer, 2004.

20 Ashish Tiwari. Termination of linear programs. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 70–82, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

21 Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

22 Christoph Walther. Argument-bounded algorithms as a basis for automated termination proofs. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction*, pages 602–621, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

A Proofs

A.1 Proofs from Section 4

To prove Lemma 1, we need the following lemmas.

▶ **Lemma 7** (Measure Evaluation respects Substitution). *For all substitutions θ,*

1. *if $(m, t, i)\Downarrow = j$ then $(m, \theta(t), i)\Downarrow = j$; and*
2. *if $(m, t, i)\Downarrow = (m', t', j)$ then $(m, \theta(t), i)\Downarrow = (m', \theta(t'), j)\Downarrow$.*

Proof Sketch. By induction on $(m, t, i)\Downarrow$. We will only prove illustrative cases.

Stuck $(m, t, i)\Downarrow = (m, t, i)$: Note that t is a variable, function, or doesn't match m. If t is a variable or function, $\theta(t)$ could be a new term, but in this case $(m, \theta(t), i)\Downarrow = (m, \theta(t), i)\Downarrow$. If t is a non-matching term, then θ cannot change it, and $(m, \theta(t), i)\Downarrow = (m, \theta(t), i)$.

Uninr/Inl $(m \triangleleft \mathbf{uninr}, \mathbf{inl}\ t, i)\Downarrow = i$: as $\theta(\mathbf{inl}\ t) = \mathbf{inl}\ \theta(t)$, $(m \triangleleft \mathbf{uninr}, \theta(\mathbf{inl}\ t), i)\Downarrow = i$. The other base cases proceed similarly.

Unroll/Roll $((m \triangleleft \mathbf{unroll}), (\mathbf{roll}\ \{t\}), i)\Downarrow = (m, t, 1 + i)\Downarrow$:
As $\theta(\mathbf{roll}\ \{t\}) = \mathbf{roll}\ \{\theta(t)\}$,

$$(m \triangleleft \mathbf{unroll}, \theta(\mathbf{roll}\ \{t\}), i)\Downarrow = (m \triangleleft \mathbf{unroll}, \mathbf{roll}\ \{\theta(t)\}, i)\Downarrow = (m, \theta(t), i + 1)\Downarrow,$$

as required.

Fix $((\mathbf{fix}\, m), t, i)\Downarrow = (((\mathbf{fix}\, m) \triangleleft m), t, i)\Downarrow$:

$$((\mathbf{fix}\, m), \theta(t), i)\Downarrow = (((\mathbf{fix}\, m) \triangleleft m), \theta(t), i)\Downarrow,$$

as required.

The other inductive cases proceed similarly. ◀

▶ **Lemma 8** (Measure Evaluation Counter Export).
If $(m, \theta(t), i + j)\Downarrow = k$ iff $(m, \theta(t), i)\Downarrow = j + k$.

Proof Sketch. A simple proof by splitting the iff, then induction on $(-)\Downarrow$. ◀

With these lemmas in hand, we can prove Lemma 1.

Proof of Lemma 1. Recall that we are guaranteed that $(m, \theta(t_1), 0)\Downarrow$ and $(m, \theta(t_2), 0)\Downarrow$ are integers. We proceed by cases on $(m, t_1, 0)\Downarrow \mathbin{\dot{-}} (m, t_2, 0)\Downarrow$.

Case 1: $(m, t_1, 0)\Downarrow = (m', s, i)$ and $(m, t_2, 0)\Downarrow = (m', s, j)$. Then

$$\begin{aligned}
&(m, \theta(t_1), 0)\Downarrow - (m, \theta(t_2), 0)\Downarrow \\
&= (m', \theta(s), i)\Downarrow - (m', \theta(s), j)\Downarrow \quad \text{(Case assumptions \& Lemma 7)} \\
&= ((m', \theta(s), 0)\Downarrow + i) - ((m', \theta(s), 0)\Downarrow + j) \quad \text{(Lemma 8)} \\
&= i - j \\
&= (m', s, i) \mathbin{\dot{-}} (m', s, j) \\
&= (m, t_1, 0)\Downarrow \mathbin{\dot{-}} (m, t_2, 0)\Downarrow.
\end{aligned}$$

Case 2: $(m, t_1, 0)\Downarrow = i$ and $(m, t_2, 0)\Downarrow = (m', s, j)$. Then

$$\begin{aligned}
&(m, \theta(t_1), 0)\Downarrow - (m, \theta(t_2), 0)\Downarrow \\
&= i - (m', \theta(s), j)\Downarrow \quad \text{(Case assumptions \& Lemma 7)} \\
&= i - (m', \theta(s), j)\Downarrow \quad \text{(Lemma 8)} \\
&\leq i - j \\
&= i \mathbin{\dot{-}} (m', s, j) \\
&= (m, t_1, 0)\Downarrow \mathbin{\dot{-}} (m, t_2, 0)\Downarrow.
\end{aligned}$$

Case 3+4: $(m, t_1, 0)\Downarrow \mathbin{\dot{-}} (m, t_2, 0)\Downarrow$ evaluates to ω.

True as $(m, \theta(t_1), 0)\Downarrow - (m, \theta(t_2), 0)\Downarrow$ is an integer, and for all $k \in \mathbb{Z}$, $k \leq \omega$. ◀

A.2 Proofs from Section 6

For the following proof, we need to enrich ω with the properties that $\omega + k = k + \omega = \omega$ for all $k \in \mathbb{Z}$ and $n \cdot \omega = \omega$ for all $n \in \mathbb{Z}^+$.

Proof of Lemma 2. Recall the initial and recursive arguments are stored in R. Thus, for any recursive call in D, indexed by i, we have

$$\begin{aligned}
(D\mathbf{w})_i &= \sum_{j=0}^{k} \mathbf{w}_j \cdot R_{ij} \\
&= \sum_{j=0}^{k} \mathbf{w}_j \cdot ((\mathbf{m}_j, R_{i2}, 0)\Downarrow \mathbin{\dot{-}} (\mathbf{m}_j, R_{i1}, 0)\Downarrow) \\
&\geq \sum_{j=0}^{k} \mathbf{w}_j \cdot ((\mathbf{m}_j, \theta(R_{i2}), 0)\Downarrow - (\mathbf{m}_j, \theta(R_{i1}), 0)\Downarrow) \qquad \text{(Lemma 1)} \\
&= \sum_{j=0}^{k} \mathbf{w}_j \cdot ([\![\mathbf{m}_j]\!]\ \theta(R_{i2})) \ - \ \sum_{j=0}^{k} \mathbf{w}_j \cdot ([\![\mathbf{m}_j]\!]\ \theta(R_{i1})) \\
&= (\lambda x.\ \sum_{j=0}^{k} \mathbf{w}_j \cdot ([\![\mathbf{m}_j]\!]\ x))\ \theta(R_{i2}) \ - \ (\lambda x.\ \sum_{j=0}^{k} \mathbf{w}_j \cdot ([\![\mathbf{m}_j]\!]\ x))\ \theta(R_{i1}) \\
&= (\mathbf{w} \cdot [\![\mathbf{m}]\!])\ \theta(R_{i2}) \ - \ (\mathbf{w} \cdot [\![\mathbf{m}]\!])\ \theta(R_{i1}).
\end{aligned}$$

Thus $(D\mathbf{w})_i \leq 0$ implies $(\mathbf{w} \cdot [\![\mathbf{m}]\!])\ \theta(R_{i2}) - (\mathbf{w} \cdot [\![\mathbf{m}]\!])\ \theta(R_{i1}) \leq 0$, and $(D\mathbf{w})_i < 0$ implies $(\mathbf{w} \cdot [\![\mathbf{m}]\!])\ \theta(R_{i2}) - (\mathbf{w} \cdot [\![\mathbf{m}]\!])\ \theta(R_{i1}) < 0$. ◀

Proof of Theorem 4. Assume there are two solutions with a different set of negative entries $\langle \mathbf{x}, \mathbf{b} \rangle$ and $\langle \mathbf{x}', \mathbf{b}' \rangle$. Let $\mathbf{x}'' = \mathbf{x} + \mathbf{x}'$ and $\mathbf{b}''_j = \max(\mathbf{b}_j, \mathbf{b}'_j)$ for each $0 \leq j < n$. Then, the tuple $\langle \mathbf{x}'', \mathbf{b}'' \rangle$ is a more optimal solution.

Firstly, $\langle \mathbf{x}'', \mathbf{b}'' \rangle$ has a greater objective. As, by assumption, there is some component different between $\mathbf{b}$ and $\mathbf{b}'$, thus we have $\sum_{i=0}^{n-1} \mathbf{b}_i < \sum_{i=0}^{n-1} \mathbf{b}''_i$ and $\sum_{i=0}^{n-1} \mathbf{b}'_i < \sum_{i=0}^{n-1} \mathbf{b}''_i$. Thus the objective is larger.

Secondly, $\langle \mathbf{x}'', \mathbf{b}'' \rangle$ satisfies all constraints, as (i) by the fact we are taking a sum of non-negative values, $0 \leq \mathbf{x}''$, (ii) by the fact we are taking a maximum, $0 \leq \mathbf{b}'' \leq 1$, and (iii) $A\mathbf{x}'' + \mathbf{b}'' \leq 0$ as

$$\begin{aligned} A\mathbf{x}'' + \mathbf{b}'' \leq 0 &\iff A(\mathbf{x} + \mathbf{x}') + \max(\mathbf{b}, \mathbf{b}') \leq 0 \\ &\iff A\mathbf{x} + A\mathbf{x}' + \max(\mathbf{b}, \mathbf{b}') \leq 0 \\ &\impliedby A\mathbf{x} + A\mathbf{x}' + \mathbf{b} + \mathbf{b}' \leq 0 \\ &\impliedby A\mathbf{x} + \mathbf{b} \leq 0 \wedge A\mathbf{x}' + \mathbf{b}' \leq 0 \end{aligned}$$

Thus, our assumption was incorrect, and, by classical contradiction, the two solutions cannot have a different set of negative entries. ◀

To prove Lemma 5, we require the following auxiliary lemma establishing the soundness of lexicographic-linear combination:

▶ **Lemma 9** (Soundness of lexicographic-linear combination)**.** *Given a list of weighted measures* $\mathbf{wm} : [[\mathbb{N} \times M]]$ *returned by the algorithm (Algorithm 2), then for each initial-recursive argument pair* R_k*, and all substitutions* θ *where* $\mathbf{m}_{ij}\ \theta(R_{k1})$ *and* $\mathbf{m}_{ij}\ \theta(R_{k2})$ *are defined (for every* i *and* j*), then*

$$[\ldots, \mathbf{w}_i \cdot [\![\mathbf{m}_i]\!], \ldots]_{\text{lex}}\ \theta(R_{k2}) < [\ldots, \mathbf{w}_i \cdot [\![\mathbf{m}_i]\!], \ldots]_{\text{lex}}\ \theta(R_{k1}).$$

(Where $\mathbf{w}_i$ *and* $\mathbf{m}_i$ *are the unzipping of* $\mathbf{wm}_i$*, and* $(\cdot)$ *and* $[\![-]\!]$ *are lifted pointwise over lists.)*

Proof. By the construction of the output list and Lemma 2, there is a first $\mathbf{w}_i$, $\mathbf{m}_i$ such that $\mathbf{w}_i \cdot [\![\mathbf{m}_i]\!]\ \theta(R_{k2}) < \mathbf{w}_i \cdot [\![\mathbf{m}_i]\!]\ \theta(R_{k1})$ and for every $i' < i$, $\mathbf{w}_{i'} \cdot [\![\mathbf{m}_{i'}]\!]\ \theta(R_{k2}) = \mathbf{w}_{i'} \cdot [\![\mathbf{m}_{i'}]\!]\ \theta(R_{k1})$. This is the definition of a lexicographic decrease ◀

Proof of Lemma 5. Suppose there was some lexicographic combination of elements of P, which we will call $[m_1, m_2, \ldots m_n]_{\text{lex}}$, which decreases at every recursive call of f but which could not be found by removing rows from the associated matrix. At some point, by assumption, this sequence of measures $[m_1, m_2, \ldots m_n]_{\text{lex}}$ must not be reachable by removing rows from the matrix A, per the rule in the lexicographic algorithm. Call the first such measure m_i. Hence, by assumption once we remove all the rows corresponding to measures $m_1, \ldots, m_{i-1}$, we must not be able to proceed using the lexicographic algorithm. Hence, there can be no column in the corresponding matrix with all entries less than or equal to 0 with at least one negative entry, namely the column associated with the measure m_i cannot have this property. But this means that there exists a recursive call on which the measure $[m_1, m_2, \ldots m_i]_{\text{lex}}$ does not decrease, which is a contradiction. ◀

Proof of Theorem 6. Suppose this were not the case and there is a lexicographic order of linear combinations of the primitive measures that could not be found by our algorithm. By definition **T-Rex** can only fail if, at some stage of evaluation, it reaches a stage where there is no non-trivial solution to the MNE problem using the numeric columns of the matrix. But using Theorem 4, we know that each time we take a linear combination of numeric columns, we get a vector whose number of negative entries is maximal and where the location of these negative entries is unique. Hence, for any given numeric part of the matrix, there is a unique set of rows R that is removed by the lexicographic phase of the algorithm such that any other set of rows that could be removed by the process of taking linear combinations is a subset of R. ◀