Termination Analysis for Logic Programs

Chris Speirs

October 8, 1997

Abstract

I survey termination analysis of logic programs with particular attention to methods for automatically proving termination. I introduce the central concepts of automatic termination analysis and give an overview of the research on other types of termination analysis. I describe the main approaches covered in the literature and compare the accuracy and efficiency of each of them.

1 Introduction

The study of termination analysis for logic programs began in 1985 when Naish investigated the question of automatically generating control for Prolog programs [12]. A basic tenet of this work is that development of a logic program is best divided into two steps. Firstly, to generate a correct logic specification of the problem, then, to add control to this to ensure that the program is executed efficiently. This was beautifully described by Kowalski's equation *Algorithm* = *Logic* + *Control*. One of the benefits of an automatic control generator is that it can be used to transform non-terminating programs into terminating programs by adding control primitives. Therefore, termination analysis can provide an important source of information for an automatic control generator.

Since then, use of termination analysis has diversified. Termination analysis can be used as an initial stage of program verification, it is also an important tool for use in program transformation. Suppose a program transformation which may improve efficiency requires the order of two atoms to be swapped. Such a transformation would not be considered valid if the transformed program no longer terminated. Termination analysis can be used to prove whether the transformed program still terminates, and therefore whether the transformation is valid.

As the question of termination is undecidable in general it is not possible to prove termination for all predicates which do terminate, with an algorithm which will always terminate itself.

Throughout this thesis, I will assume that the reader is familiar with logic programs, and the syntax of Prolog. For an introduction to logic programs and Prolog, I refer the reader to [1].

The structure of this paper is as follows. The next section presents an overview of the different areas of research into termination analysis and explains which areas will be covered in more detail. Section 3 presents some examples of termination analysis as well as defining some relevant concepts. Section 4 describes the main approaches covered in the literature and section 5 uses a set of examples to demonstrate the strengths and weaknesses of each approach. Section 6 concludes with a brief discussion.

2 Taxonomy of Termination Research

What is meant by termination in the context of logic programs? A simple predicate like `append` can demonstrate the different types of termination that are possible.

```
append([],Y,Y).
append([H|X],Y,[H|Z]) :-
    append(X,Y,Z).
```

A query to `append` of the form `append([2],[3],X)` will always terminate, in fact all queries to `append` where the first 2 lists are bound to finite lists will terminate. This is an example of *universal termination.*

A predicate is universally terminating if it will produce all of its solutions and then terminate. The other type of termination is called *existential termination*. A predicate is existentially terminating if it produces at least one solution, or fails, in a finite time. For example the query `append(X,X,X)` will succeed exactly once (binding `X` to `[]`) and then enter an infinite computation. Therefore this query is existentially terminating (because it produces a solution in a finite time), but not universally terminating. This also shows that the termination properties of a predicate depend not only on the predicate itself, but also on how the predicate is called. For example the query `append([1],X,X)` will not terminate at all. Therefore, when proving termination of a predicate it is necessary to specify the type of query that termination analysis is being applied to.

Research into termination of imperative programs has been investigated, but with little success, as non-termination of an imperative program can be caused not only by recursion, but also by looping constructs and goto statements. Similar research has been carried out in the area of deductive databases. The aim of a deductive database is to find all possible facts which may be derived from a given set of facts and rules. In the context of deductive databases, the question is no longer whether a program terminates, but whether it is *safe*. A program is considered *safe* if there are only a finite number of facts that can be derived from it. Clearly if an infinite number of facts can be derived, then any attempt to find them all will not terminate.

2.1 Universal and Existential Termination

Almost all the reasearch into termination analysis has investigated the question of universal termination. This is because universal termination is easier to prove than existential termination and also it tends to be a more useful result. Universal termination proofs tend to be more useful because we often require more than one answer from a predicate. Also, it is normally not clear whether a predicate will be backtracked over, due to the non-determinism inherent in logic programs. If we can prove that a predicate is universally terminating, then it follows directly that it must be existentially terminating as well.

Existential termination is hard to prove because it requires proof that every possible non-terminating computation is preceded by a solution. The other difficulty is that, given two predicates which are both existentially terminating, the conjunction of these two predicates does not necessarily terminate. For example,

```
generate([1]).
generate([1|X]) :- generate(X).

is_empty([]).
```

Both `generate/1` and `is_empty/1` are existentially terminating (in fact `is_empty/1` is universally terminating), and yet the conjunction `generate(X), is_empty(X)` does not terminate, because `generate` produces an infinite number of answers, none of which are the empty list.

Existential termination is useful when considering situations where only a single answer is required from a predicate. The most common case of this is when a goal is negated. A negated goal is universally terminating if and only if the goal is existentially terminating. None of the analyses which I have seen take advantage of this fact, and require a goal to be universally terminating before the analysis assumes that the negated goal will terminate.

In this thesis, I will use termination to mean universal termination.

2.2 Types of Analyses

There are two main types of termination analysis which may be classified as automatic and manual analyses. The automatic analyses try to find well defined algorithms which can be used to prove the termination of programs. Manual termination analysis is concerned with finding what is required to prove termination, and investigating the boundary between predicates which can be proved to terminate and those that can't.

Automatic termination algorithms must restrict themselves to sufficient conditions for proving termination. If they covered sufficient *and necessary* conditions for termination, then the analyses themselves would not always terminate. As these algorithms are written to be made into programs, it is important that the algorithm is defined in such a way that it is possible for the algorithm to be translated into a program. The complexity of these algorithms is also of interest, and much of the comparison of different algorithms is based on their complexity. Until recently, few of these algorithms were actually encoded into programs, so there have not been many results with which to compare the different algorithms. Most of the automatic termination analyses have been based around static analysis, but some of them have investigated the problem of catching non-termination at runtime [16].

Manual termination analyses are used to give insight into the problem of termination analysis. The efficiency of the analysis is in general not very important. Instead the algorithm should attempt to clarify what is required to prove termination of a predicate. These algorithms also investigate the boundary of the halting problem, and attempt to find classes of programs which are on the boundary of decidability. In this thesis I will concentrate on the automatic termination algorithms, as that is where the bulk of the research has been carried out.

3 Reasoning about Termination

3.1 Types of Query

As was shown in Section 1, the termination of a predicate depends on the type of query that is used. Initially, termination analyses allowed only ground queries to be analysed. This was a somewhat unreasonable requirement, as most useful predicates produce output. Therefore, the type of query was specified through the use of modes. The mode of a predicate specifies which arguments are guaranteed to be ground when the predicate is called. This is sufficient information to identify the type of query that is to be made. The mode of a predicate can either be found by using mode analysis before termiantion analysis is run, or the mode can be specified as input to the termination analyser. Throughout this thesis I will specify the mode of a predicate by using the syntax of Mercury [9]. For example a mode of `append` is specified using

```
:- mode append(in, in, out).
```

This declaration states that the first 2 arguments of `append` will be ground, and the last argument will be free when `append` is called. It also guarantees that `append` will bind a value to its last argument when it is executed.

However, even this limits the types of queries that can be analysed, as they require inputs to be fully ground, which disallows any partially instantiated data structures. Therefore, some researchers have generalised the idea of groundness so that arguments do not need to be ground, they must only be rigid with respect to some function. This idea of *rigidity* enables termination analysis to be performed on partially instantiated data types such as list skeletons. These functions, which are called *norms*, will be covered in more detail later in the thesis. They are used to measure the size of the predicate's arguments by providing a mapping from the set of all possible terms to the set of natural numbers. For a term to be rigid with respect to one of these functions means that its size (that is, its value according to the specified function) cannot be changed by further instantiating it.

3.2 Proving Termination

What information is required to prove that a predicate terminates? The first point to notice is that the only way for a pure logic program to fail to terminate is through infinite recursion. As a result it is substantially easier to prove termination of a logic program than it is for an imperative program, where non-termination may be caused by looping constructs and goto statements, as well as by infinite recursion. Termination analysis for logic programs attempts to prove that whenever a predicate is called, the call will result in a finite amount of recursion. The process that is used to prove this is best described by some examples.

```
:- mode delete(out, in, out).
delete(X, [X|Xs], Xs).
delete(X, [Y|Ys], [Y|Zs]) :- delete(X, Ys, Zs).
```

This predicate nondeterministically removes any element from its second argument, and returns the removed element as its first argument. The list with the element removed is returned as the last argument. This predicate will always terminate because the only recursive call is with arguments which are strictly smaller than the arguments in the head. In fact what is important is that the *input* arguments decrease in size between the head and recursive call. This guarantees termination because each time `delete` is called, the recursive call is with smaller inputs, and the recursive call will in turn make another recursive call with smaller inputs until eventually the recursive call will be on an empty list and `delete(X, [], Zs)` will fail, so the `delete` predicate as a whole will terminate.

This indicates what is meant by 'smaller' in this context. It is important that it is not possible for an argument to get smaller indefinitly. If that were allowed, then showing that arguments have got smaller between the head and recursive call would not necessarily prove termination. This also gives an indication of the requirements that are placed on the norms that are used to prove rigidity (Section 3.1). If a certain argument position is rigid with respect to a particular function, then it is important that the same function is used when calculating whether a variable is now smaller. Because the function maps onto the set of natural numbers, it guarantees that a a variable may only get smaller a finite number of times.

Here is another example of a terminating predicate.

```
:- mode permute(in, out)
permute([], []).
permute(Xs, [Y | Zs] ) :- delete(Y, Xs, Ys), permute(Ys, Zs).
```

The reason that this predicate terminates is more complicated. It depends on the fact that `delete` returns as its third argument a list which is smaller than the second argument which is passed to it. Therefore, to prove the termination of `permute`, we need a relationship between the sizes of the arguments of `delete`. Termination analysis is often divided up into two passes. The first attempts to draw relationships between the sizes of the arguments of each predicate in the program. These relationships are often called *inter-argument relations*. The second then uses this information to prove the termination of as many predicates as possible. These two sections are quite independent. In fact a number of papers have been written which only investigate the first part of finding inter-argument relations [4, 5], whereas some others only investigate the second part [14].

3.3 Norms

Norms are used to associate a size with various terms. They are used to decide whether an argument is rigid and they are also used to determine how the sizes of different variables relate to each other. There have been a number of different definitions of norms proposed in the literature. As time progresses, the tendency is to use more and more general norms for proving termination.

One of the first norms to be proposed was the linear norm [13], which is defined as

Definition 3.1 (linear norm)

Let T be a set of terms. A norm $\|\ldots\|$ for T is said to be *linear* if for each $t \in T$ and each instantiation σ for t.

$$\|t\sigma\| = \|t\| + \sum_{i=1}^{n} \|v_i\sigma\|$$

holds, where $\{v_1, \ldots, v_n\}$ is the multiset of variable occurences in t. ■

Later on, Bossi, Cocco and Fabris introduced *semi-linear norms* [3] which are useful for handling partially instantiated data structures.

Definition 3.2 (semi-linear norm)

Let T be a set of terms. A *norm* $\|\ldots\|$ on T is *semi-linear* if it can be recursively defined, for any term in T, by using the following schema:

if t is a variable then $\|t\| = 0$

if $t = f(t_1, ..., t_n)$ then $\|t\| = c_0 + \|t_{i_1}\| + ... + \|t_{i_m}\|$, where $c_0 \geq 0$, and $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$ ■

The important point of a semi-linear norm is that it is not necessary to consider all the subterms of any particular term (as was required by the linear norm). This allows partially instantiated data structures to be considered, as the norm can be chosen in such a way that the uninstantiated sections of the data structure have no effect on the norm.

Some popular norms include the *list-size norm*, which defines the size of a list to be its length. However, this norm is only applicable to lists.

Definition 3.3 (list-size norm)
$\|t\| = 0$, if $t = []$ or if t is a variable
$\|t\| = 1 + \|xs\|$, if $t = [x|xs]$ ■

and the *term-size norm*, which defines the size of a term to be the number of elements in the data structure which defines the term.

Definition 3.4 (term-size norm)
$\|t\| = 0$, if t is a constant or if t is a variable
$\|t\| = 1 + \sum_{i=1}^{n} \|t_i\|$, if $t = f(t_1...t_n)$ ■

In their recent paper, Decorte and De Schreye [7] define an even more general norm:

Definition 3.5 (generalized semi-linear norm)
Let T be a set of terms. A *norm* $\|.\|$ on T is *semi-linear* if it can be recursively defined, for any term in T, by using the following schema:
If t is a variable then $\|t\| = 0$
If $t = f(t_1, ..., t_n)$ then $\|t\| = c_0 + c_{i_1}\|t_{i_1}\| + ... + c_{i_m}\|t_{i_m}\|$, where$c_0 \geq 0$, and $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$
■

In this norm, it is possible to assign different weights to each of the subterms of any particular term. Decorte and De Schreye propose a method for forming constraints on these weights, and then solving for them. This allows the norm to be flexible, and to be changed automatically to provide a suitable norm for proving the termination of any particular predicate. Unfortunatly, this idea is quite new, so there are no experimental results that can be used to evaluate the accuracy of this new method. It would be interesting to analyse the effect of varying the norm on the accuracy of termination analysis. So far, it seems that the choice of norm has very little effect on the accuracy of algorithms [10, 15]. Using a generalised norm presents a problem when analysing large programs as the choice of norm must not change throughout the analysis. Therefore when doing analysis using generalised norms, the whole program must be considered at the one time which could cause efficiency problems.

4 The Algorithms

4.1 Bezem's Approach

Recurrency is a method for manually proving termination of a predicate. The theory itself is not terribly useful for proving termination, but it does provide an insight into what is required for a program to terminate. The reason that this algorithm is not terribly useful is because it only addresses termination of ground atoms, and because it only proves the existence of a suitable level mapping, without proposing a method for finding it.

Definition 4.1 (level mapping)
Let B_P be an atom in the program P. A *level mapping* is a function $f : B_P \rightarrow N$ ■

Definition 4.2 (recurrency)
A program P is *recurrent* if there exists a level mapping, f, such that for each ground instance $A \leftarrow B_1, \ldots, B_n$ of a clause in P: $f(A) > f(B_i)$, for each $i = 1, \ldots, n$. ■

The following result is due to Bezem [2]

Theorem 4.3 A program P is recurrent, if and only if, it is terminating. ■

This provides the somewhat surprising result that for any ground terminating query, a level mapping exists such that the size (according to the level mapping) of each of the subgoals of the query are strictly smaller than the size of the head.

Proving this theorem consists of two seperate parts. Given any recurrent program, it follows directly that the program must terminate, as the definition of a recurrent program guarantees a well-founded ordering on each of the clauses defining the program. To prove that any terminating program is recurrent requires the generation of a suitable level-mapping. Given any variable free atom in a terminating logic program, its proof tree must be finite. Therefore, for any variable free atom A, define a level mapping $f : B_P \rightarrow N$ such that $f(A)$ is the number of nodes in the proof tree of A. Bezem proves that according to this level-mapping A is recurrent. The proof basically follows from the fact that any atom B_i which occurs in the clause which defines A has a proof-tree which is a proper subtree of the proof-tree of A.

4.2 Plümer's Approach

Plümers Phd thesis [13] encouraged a lot of work in the area of termination analysis. His thesis was one of the first to provide an algorithm for automatically proving termination of Prolog programs. His algorithm was implemented by Schmidt, and the results of it are included in Plümer's thesis. More importantly, in his thesis, Plümer identified many of the challenges of termination analysis, and proposed solutions for most of them. For example, Plümer identified the problem of *mutual recursion*, and showed how in most cases mutual recursion can be eliminated by using unfolding.

Plümers original algorithm is relativly inefficient, and also quite restrictive. It applies only to programs which satisy the following requirements.

1. it must be *normalised*
2. it must be well moded
3. it must be free of mutual recursion

Requirement 1 is easy to satisfy, and normalising the program simplifies the analysis. Well modedness (requirement 2) cannot always be achieved, and Plümers algorithm cannot process these predicates. Moreover, transforming a given non-terminating predicate so that it becomes well moded can transform it into a terminating predicate. Therefore, Plümers algorithm can only prove termination of the transformed program. It is always possible to satisfy requirement 3, but this can have the unfortunate side effect of makeing proof of termination more difficult.

Two years later, Gröger and Plümer [8] extended the method to include handling of mutual recursion. This newer algorithm is also substantially more efficient. In fact its complexity is polynomial in the number of clauses in the program, and in the maximal number of literals of any clause in the program, whereas Plümers original algorithm was exponential. This improvement in efficiency was made possible by adding another restriction to the class of programs that could be analysed. When using this new algorithm, programs must be shown to be *well annotated in a strong sense*. To be well annotated in a strong sense, there must be only one producer for each variable, and the producer must come before the consumer. By adding this restriction, Gröger and Plümer were able to form their inter-argument relations in polynomial time.

Plümer's approach is based on *predicate inequalities* which are of the form:

$$\sum_{i \in I} \|p_i\| + \gamma_p \geq \sum_{j \in J} \|p_j\|$$

where p_k stands for the argument k of predicate p, I is a subset of the set of input argument positions of p, and J is a subset of the set of output argument positions although normally I and J will be the complete set of input and output positions. The value of γ_p indicates the difference in size between the inputs and outputs of the predicate p. There are 2 phases to Plümer's method. The first phase analyses each predicate and attempts to find the minimum value of γ for which the previous inequality is guaranteed to hold. The second phase then uses this information to form a relationship between the size of arguments in a recursive call relative to the size of the corresponding arguments in the head. If the analysis can prove that the arguments of the recursive call are strictly smaller than those in the head, the termination is guaranteed.

Analysis of the previous `permute` example would proceed as follows. The first pass would analyse `delete` and form the predicate inequality (assuming that list-size norm is being used).

$$\|delete_2\| - 1 \geq \|delete_1\| + \|delete_3\|$$

γ_{delete} has been given the value -1, showing that the size of the outputs of `delete` are strictly smaller than the size of the input. The second phase needs to prove that $\|Ys\| < \|Xs\|$. To prove this it needs the predicate inequality for `delete` which states that $\|Xs\| - 1 \geq \|Y\| + \|Ys\|$. Given that $\|Y\|$ cannot be negative, this inequality proves that $\|Ys\| < \|Xs\|$, so `permute` is guaranteed to terminate.

Plümers method has been improved on by a number of researchers. Quite recently Decorte and De Schreye [7] have introduced the *generalised semi-linear norm* which was covered in section 3.3. Speirs, Somogyi and Søndergaard [15] made two improvements. Instead of considering predicate inequalities where the sum of the sizes of all input arguments is compared with the total size of the outputs, they consider the sum of the sizes of a *subset* of the input arguments, where the subset is chosen by using a fixed-point analysis. A similar modification is made to the second phase of analysis where instead of considering the total size of the input arguments in the head relative to the total size of the input arguments in the recursive call, they consider a subset of the input argument positions. This subset is also calculated using a fixed-point iteration. Sohn and Van Gelder [14] also modified the second phase of the analysis. Instead of trying to find a subset of input arguments which strictly decrease in size, they try to find a non-negative linear combination of the input arguments which is guaranteed to decrease in size. To find the optimal linear combination of input arguments, Sohn and Van Gelder create and then solve a set of constraints.

4.3 Monotonicity Constraints

Another large area of research into termination analysis involves so-called *monotonicity constraints*. These are inter-argument relations of the form $p_i + c \geq p_j$, where p_i and p_j are variables representing the sizes of any term that can occur in the i^{th} respectively, j^{th} argument positions, and c is any integer. The value of i is solved for while analysing the predicate. These monotonicity constraints are then used to prove that the size of arguments of recursive calls are strictly smaller than the size of the variables in the head, therefore proving termination.

Most of these algorithms use a directed graph type structure, where each node in the graph represents an argument of the head of the predicate, or an argument of one of the atoms in the body of the predicate. A pair of nodes are joined to each other by an edge if the arguments these nodes represent are the same size, or by a directed *arc* if they differ in size. The arc is connected from the larger argument to the smaller argument. Therefore, an arc from p_i to p_j is equivalent to the inequality $p_i > p_j$. Some analyses annotate the arcs with the relative sizes of the 2 arguments, allowing them to make use of the more general form of constraint $p_i + c \geq p_j$.

After these graphs have been formed, proof of termination amounts to showing that at least one argument position in the head of the clause is connected to the same argument position in the recursive call by a path which includes at least one arc. This is sufficient to prove termination, as it guarantees that this argument is strictly monotonically decreasing, and therefore the recursion must be finite.

Analysis of delete would produce the set of monotonicity constraints $\|delete_2\| > \|delete_1\|$ and $\|delete_2\| > \|delete_3\|$.

Given these constraints, the directed graph shown in Figure 1 would be produced. From the graph, it is clear that the only path from the first argument of `permute` in the head to the corresponding argument in the recursive call contains an arc. Therefore `permute` is guaranteed to terminate.

Ullman and Van Gelder [16] were one of the first researchers to use monotonicity constraints. Their termination analysis is designed to be used with the NAIL! system. The NAIL! system is a so-called "knowledge-base management system" which applies logic rules to a large database [11]. The NAIL! system chooses an execution strategy at runtime and the termination analysis is used to help choose the best execution strategy. As it is used at runtime, Ullman and Van Gelder's algorithm is very efficient

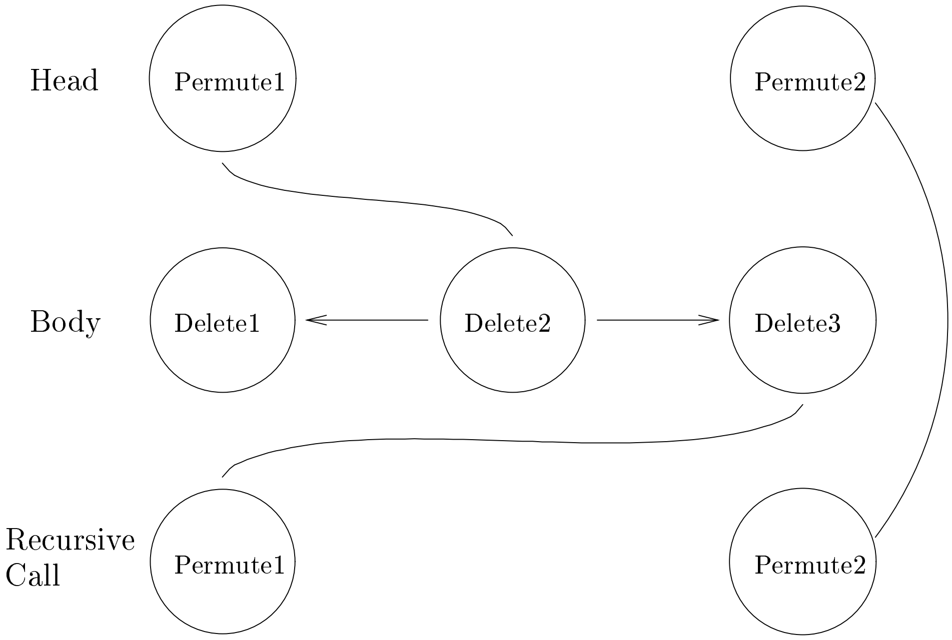

Figure 1: Directed graph for permute/2

but not very accurate. Brodsky and Sagiv [5] investigated the use of disjunctions of constraints, which can significantly improve accuracy of the analysis but does introduce some problems as some predicates need an infinite number of constraints to accurately describe the relationships between various arguments. Recently Lindenstrauss and Sagiv [10] implemented an automatic termination analyser which implements the algorithm described above.

5 Examples, and How the Analyses Apply

5.1 Permute

Consider the program:

```
perm([],[]).
perm(L,[H|T]) :-
    append(V,[H|U],L),
    append(V,U,W),
    perm(W,T).

append([],Y,Y).
append([H|X],Y,[H|Z]) :-
    append(X,Y,Z).
```

Analysis which uses monotonicity constraints is unable to prove the termination of `perm` as it is not possible to create a useful constraint between any pair of the arguments of `append`. The most accurate constraint that can be created is $append_1 + append_2 = append_3$. Given this constraint, it is relativly easy to show that $\|L\| > \|W\|$, which is the constraint which is required to prove the termination of `perm`. On the other hand, using monotonicity constraints, the most accurate set of constraints for `append` are $append_1 \leq append_3$ and $append_2 \leq append_3$. These constraints are not sufficient to prove the termination of `perm`, as they cannot show that $\|L\| > \|W\|$. In fact, when using these 2 constraints, it is not possible to place a limit on $\|W\|$.

5.2 Normal Mergesort

Consider the program:

```
:- mode mergesort(in, out).
mergesort([],[]).
mergesort([X],[X]).
mergesort([X,Y|Xs],Ys) :-
    split([X,Y|Xs],X1s,X2s),
    mergesort(X1s,Y1s),
    mergesort(X2s,Y2s),
    merge(Y1s,Y2s,Ys).

:- mode split(in, out, out).
split([], [], []).
split([X|Xs], [X|Ys], Zs) :- split(Xs, Zs, Ys).

:- mode merge(in, in, out).
merge([], [], []),
merge([X|Xs], [Y|Ys], [X|Zs]) :- X >= Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X < Y, merge([X|Xs], Ys, Zs).
```

Neither Plümer's method, nor monotonicity constraints can prove the termination of this form of mergesort. The reason why neither can prove termination is that termination is conditional on `split` being passed a list of at least 2 elements. If `mergesort` is modified in the following way, so that it is not guaranteed that `split` is passed a list of at least 2 elements, then `mergesort` will no longer terminate.

```
mergesort([],[]).
mergesort([X|Xs],Ys) :-
    split([X|Xs],X1s,X2s),
    ...
```

The problem is that `split` only returns output lists which are both smaller than its input list if the input list contains at least 2 elements. Therefore, proving termination of `mergesort` would require a complicated set of inequalities:

$$\|split_1\| > \|split_2\| \text{ and } \|split_1\| > \|split_3\|, \text{ if } \|split_1\| > 2 \text{ (assuming list}-\text{size norm is used)}$$

$$\|split_1\| \geq \|split_2\| \text{ and } \|split_1\| \geq \|split_3\|, \text{otherwise}$$

Automatically generating such inequalities would be very challenging.

5.3 Modified Mergesort

For this example, `mergesort` has been modified so that instead of calling `split`, it calls `split2`. `split2` guarantees that each of the output lists will be smaller than the input list.

```
mergesort([],[]).
mergesort([X],[X]).
mergesort([X,Y|Xs],Ys) :-
    split2([X,Y|Xs],X1s,X2s),
    ...

:- mode split2(in, out, out).
split2([X,Y|Xs],[X|Ys],[Y|Zs]) :- split(Xs,Ys,Zs).
```

Using monotonicity constraints, it is relativly easy to produce the following inequalities for split.

$$\|split2_1\| > \|split2_2\| \text{ and } \|split2_1\| > \|split2_3\|$$

These constraints allow us to prove the termination of mergesort, as each recursive call has strictly smaller inputs than the head. However, when using Plümers algorithm it is not possible to prove the termination of this version of `mergesort`. The best constraint that can be created using Plümers algorithm is.

$$\|split2_1\| \geq \|split2_2\| + \|split2_3\|$$

This constraint is no better than the constraint that than be created for normal `mergesort`, so the modification has not helped Plümers algorithm prove termination.

5.4 Unfolded Mergesort

Plümer [13] made an extensive investigation into the use of unfolding in proving termination. `mergesort` is one example of where unfolding allows proof of termination.

```
mergesort([],[]).
mergesort([X],[X]).
mergesort([X,Y|Xs],Ys) :-
    split(Xs,X1s,X2s),
    mergesort([X|X1s],Y1s),
    mergesort([Y|X2s],Y2s),
    merge(Y1s,Y2s,Ys).
```

Here we have unfolded `split` twice. It is now clear that each recursive call has strictly smaller inputs than the head so termination is guaranteed. Both monotonicity constraint algorithms and Plümers algorithm can prove the termination of unfolded `mergesort`.

6 Conclusion

There was a lot of interest in the area of termination analysis in the late 80's and early 90's but from then until quite recently, there has been little research in the field. These recent papers have shown that termination analysis is now at a stage where it is accurate and efficient enough to be useful not only as input to an automatic control generator or optimizer but also as feedback to the programmer, to give warnings of possible non-termination [15].

For a very detailed survey of research into termination analysis, I refer the reader to the paper by De Schreye and Decorte [6]. Although not very recent, it does cover essentially all of the research that was carried out before 1994, and provides an excellent starting point for further research. Another useful reference is Plümers Phd thesis [13] which details some program transformations that can be used to improve termination properties as well as describing Plümer's algorithm in detail.

This thesis introduced the problem of termination analysis for logic programs and gave an overview of the topic as a whole. It then provided some insight into proving termination and presented the main approaches that are covered in the literature. It then used some examples to demonstrate the strengths and weaknesses of each of these approaches. No particular aooroach described in the literature has universal appeal.

References

[1] Henri E. Bal and Dick Grune. *Programming Language Essentials*. Addison-Wesley, 1994.

[2] Marc Bezem. Strong termination of logic programs. *Journal of Logic Programming*, 15(1&2):79–98, 1992.

[3] A. Bossi, N. Cocco, and M. Fabris. Proving termination of logic programs by exploiting term properties. In S. Abramsky and T. S. E. Maibaum, editors, *Proc. TAPSOFT'91, vol. 2*, Lecture Notes in Computer Science 494, pages 153–180. Springer-Verlag, 1991.

[4] A. Bossi, N. Cocco, and M. Fabris. Typed norms. In B. Krieg-Brückner, editor, *Proc. ESOP'92*, Lecture Notes in Computer Science 582, pages 73–92. Springer-Verlag, 1992.

[5] Alexander Brodsky and Yehoshua Sagiv. Inference of inequality constraints in logic programs (extended abstract). In *Proc. Tenth ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 227–240. ACM Press, 1991.

[6] Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.

[7] Stefaan Decorte and Danny De Schreye. Demand-driven and constraint-based automatic left-termination analysis for logic programs. In L. Naish, editor, *Proceedings of the 1997 International Conference on Logic Programming*. MIT Press, 1997.

[8] Gerhard Gröger and Lutz Plümer. Handling of mutual recursion in automatic termination proofs for logic programs. In K. Apt, editor, *Proc. Joint Int. Conf. Symp. Logic Programming*, pages 336–350. MIT Press, 1992.

[9] Fergus Henderson, Thomas Conway, Zoltan Somogyi, and David Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1996. Available from `http://www.cs.mu.oz.au/mercury`.

[10] Naomi Lindenstrauss and Yehoshua Sagiv. Automatic termination analysis of logic programs (with detailed experimental results), 1997. `http://www.cs.huji.ac.il/~naomil`; Shorter version in L. Naish, editor, *Proceedings of the 1997 International Conference on Logic Programming*. MIT Press, 1997.

[11] K. Morris, J. D. Ullman, and A. Van Gelder. Design overview of the NAIL! system. In *Proceedings of the 3rd International Conference on Logic Programming*, pages 554–568. Springer-Verlag, 1986.

[12] Lee Naish. Automating control of logic programs. *Journal of Logic Programming*, 2(3):167–183, October 1985.

[13] Lutz Plümer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1990.

[14] Kirack Sohn and Allen Van Gelder. Termination detection in logic programs using argument sizes (extended abstract). In *Proc. Tenth ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 216–226. ACM Press, 1991.

[15] Chris Speirs, Zoltan Somogyi, and Harald Søndergaard. Termination analysis for mercury. In P. Van Hentenryck, editor, To appear in *Static Analysis: Proceedings of the 4th International Symposium*, Lecture Notes in Computer Science. Springer, 1997.

[16] Jeffrey D. Ullman and Allen Van Gelder. Efficient tests for top-down termination of logical rules. *Journal of the ACM*, 35(2):345–373, 1988.