The University of Melbourne
Department of Computer Science
&
Software Engineering

Honours Thesis

Termination Analysis for Mercury using Convex Constraints

Julien Fischer

Supervisors: Zoltan Somogyi & Harald Søndergaard

August 2002

Abstract

We present a new termination analysis for Mercury that approximates interargument size relationships with convex constraints. These relationships are derived using an analysis based upon abstract interpretation. While this analysis is more expensive than that of the existing termination analyser, it is able to prove the termination of a larger class of predicates. We consider how this analysis interacts with aspects of Mercury such as the module system, declarative I/O, higher-order code and the foreign language interface. Finally, we present some experimental results that demonstrate the accuracy of this method.

1 Introduction

Automatic termination analysis is a program analysis technique that attempts to automatically prove that a program will always terminate. It has been successfully applied to logic programs because the only possible source of non-termination in logic programs is due to recursion. Automatic termination analysis can be a useful part of the compilation process, for a number of reasons, not the least of which is an indication of probable programming errors.

The study of automatic termination analysis for logic programs has its roots in the work of Ullman and Van Gelder [29] and Plümer [23, 24]. These techniques were limited in the type of programs for which they could successfully generate termination proofs. Subsequent work in the area [1, 20], has successfully lifted many of these restrictions. Based largely on the work of Gröger and Plümer [13], Speirs, Somogyi and Søndergaard [28] designed and implemented a termination analysis system for the logic programming language Mercury.

Termination of a logic program is often shown by demonstrating that for every recursive call there is a strict decrease over some well-founded ordering between a subset of the input arguments in the head of a predicate and the corresponding subset of input arguments in the recursive call. In order to show that such a decrease occurs, it is often necessary to derive relationships between the sizes of the terms in the intervening goals. Consequently many termination analysis techniques begin by performing some form of interargument relationship (IR) analysis in order to discover such relationships.

The problem of inferring interargument relationships for the purposes of proving termination was initially tackled by Ullman and Van Gelder [29]. The relationships they derived were of limited use because they took the form of inequalities between the sizes of two argument positions. Plümer [23] extended this work to allow interargument relationships to be described by linear inequalities of the form:

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

where p_k stands for the size of argument of k of predicate p, I is a subset of the input argument positions of predicate p, J is a subset of the set of its output arguments and $\gamma_p \in \mathbb{Z} \bigcup \{\infty\}$.

Verschaetse and De Schreye [26] used abstract interpretation [6], as a basis for deriving argument size relationships. In their work the abstract interpretation took place over a domain of linear equalities. The resulting interargument relationships were expressed as systems of linear equalities. Van Gelder [30] used a different approach and approximated interargument relationships using systems of linear *inequalities*. These are naturally more expressive than equalities. However Van Gelder's work was incomplete because the analysis could fail to find a fixpoint. The work of Cousot and Cousot [7] addressed this failing and the technique was successfully applied by Benoy and King [2] to automatically derive interargument relationships expressed as systems of linear inequalities.

In this thesis we present the design and implementation of a new automatic termination analyser for Mercury. The new analysis is more precise than that of [28] and can prove termination in many instances where the old analysis could not. For this new analysis we use abstract interpretation over a domain of convex constraints [1] to perform the IR analysis. The use of convex constraints as an abstract domain is well understood and has had many applications [9, 14, 15], including time complexity analysis [19] and termination analysis of logic programs [5].

Although we assume no particular knowledge of Mercury, it is assumed that the reader is familiar with the basic principles of logic programming. For an overview of these matters we refer the reader to Lloyd [21]. We provide a short introduction to the relevant aspects of Mercury in section 2. It is also assumed that the reader is familiar with the concept of abstract interpretation; for a concise summary of which, specifically with reference to logic programming, we refer the reader to [8]. Some knowledge of linear programming techniques is also assumed, see Schrijver [27] for example.

The remainder of this thesis is structured as follows. In section 2 we give a brief introduction to the relevant aspects of Mercury. In section 3 we give an overview of the termination analysis of logic programs and recall some of the fundamental definitions that are used in the construction of termination proofs. Section 4 describes the IR analysis that we have implemented for the new analyser and 5 contains a description of the termination analysis. Section 6 discusses how the termination analysis can be extended to encompass various features of Mercury like modules, higher-order code and the foreign language interface. In section 7 we discuss some issues encountered during the implementation of the analysis. Section 8 presents the results of running the analyser on some test suites. We finish by presenting some ideas for further research that have arisen before giving some concluding remarks.

2 Mercury

In this section we give a brief overview of the aspects of Mercury that are relevant to the remainder of this thesis. For a more complete description of Mercury we refer the reader to the Mercury language reference manual [17]. Most of the material in this section is due to [28].

[1] Or equivalently *convex polyhedra*.

Mercury is a pure logic programming language designed to support teams of programmers working on large application programs. Apart from its lack of non-logical constructs, its main distinguishing features are its module system, its strong type-, mode- and determinism systems, and its support for higher-order programming.

The module system of Mercury is based on the module systems of languages such as Modula-2 and Ada. Each source file is a module, with an interface part and an implementation part. The interface part contains declarations of the types, predicates and functions exported from the module; the implementation part contains the definitions of the exported types, predicates and functions as well as the declarations and definitions of any types, predicates and functions private to the module. Modules cannot access anything that is private to another module; they can access the things exported from another module only be explicitly importing that module.

The syntax of Mercury is based on the syntax of Prolog, although the semantics is very different. The bodies of clauses in Mercury may contain negations, disjunctions and if-then-elses. Although programmers may define functions , the compiler internally translates functions with n arguments into predicates with $n+1$ arguments, the extra argument representing the return value.

The type system for Mercury is based on many-sorted logic with parametric polymorphism. It is quite conventional, being very similar to the type systems of modern functional languages such as ML and Haskell, as well as to the type system of Gödel. Programmers must define the types they use, and they must declare the types of the arguments of all predicates/functions exported from modules; the compiler will infer the type of the arguments of predicates/functions local to a module and the types of all variables in the program. If it cannot do so, the compiler rejects the program.

The mode of a predicate or function describes which argument should be computed from which other arguments. The arguments which are supposed to already known at the time of a call to the predicate have the mode "in", while the arguments whose values are supposed to be computed by the predicate have the mode "out". Some predicates have more than one mode. For example `append` can be used both to concatenate two lists (in the mode `append(in,in,out)` or to divide up an existing list (in the mode `append(out,out,in)`). Since the different modes of a predicate often don't have much in common (they often have different number of solutions, they sometimes have different termination properties and they almost always require different code for efficient implementation), the compiler treats the various modes of a predicate or function completely independently. We call each mode of a predicate or function a *procedure*. The mode of a procedure specifies a contract between the procedure and its callers. On the one hand, the caller promises to pass ground terms as the input arguments of calls to the procedure. On the other hand the procedure promises to return ground terms as the output arguments of the calls to the procedure. When it analyses the definition of a procedure, the compiler will pick an order for each conjunction and a mode for each call, unification and other builtin in a way that satisfies all the contracts involved. If it cannot do so, it rejects the program. One consequence of this is that the compiler knows the exact instantiation state of every variable at every point in the program.

The compiler associates with each procedure a piece of information we call a determinism. This determinism says whether the procedure will succeed exactly once (det), at most once (semidet), at least once (multi) or an arbitrary number of times (nondet).

Mercury supports higher-order programming. A predicate or function may take other predicates and/or functions as arguments (both input and output), their definitions may contain lambda expressions (each defining either a predicate or a function), and they can partially apply higher-order terms. However, the semantics of Mercury is still first-order, since the language does not allow higher-order terms to be compared or unified (except for unifications that implement assignment).

The compiler converts any predicate definition with two or more clauses into a single clause with an explicit disjunction. Most parts of the Mercury compiler, including the termination analyser, operate on clauses that have been converted into *superhomogenous form*, where each atom (including the heads of clauses) is one of the forms

```
p(X1, ..., Xn)
Y = X
Y = f(X1, ..., Xn)
```

Every atom inside the compiler is annotated with mode information. For unifications, we show this mode information by writing `<=` for construction unifications, `=>` for deconstruction unifications, `==` for equality tests, and `:=` for assignments. As an example, the internal form of `append(in,in,out)` is:

```
append(H1, H2, H3) :-
   (
      H1 == [],
      H3 := H2
   ;
      H1 => [X | Xs],
      append(Xs, H2, Zs),
      H3 <= [X | Zs]
   ).
```

`H1`, `H2`, `H3` stand for "head variables 1, 2, and 3". Note that due to mode reordering the internal form of `append(out,out,in)` is considerably different.

3 Termination Analysis of Logic Programs

In this section we give an overview of the area of termination analysis as applied to logic programs and briefly discuss some existing termination analysis systems. For a more detailed exposition of the topic of termination analysis we refer the reader to the survey paper by De Schreye and Decorte [10].

3.1 So just what do we mean by *termination* anyway?

Consider the termination behavior exhibited by the well known predicate for list concatenation:

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :-
   append(Xs, Ys, Zs).
```

This predicate displays a number of termination behaviors depending on which arguments are instantiated when it is called. A query such as

```
?- append(Xs, Ys, [a,b]).
```

terminates, yielding up to three answers.

```
?- append(Xs, [a,b], Ys).
```

can produce infinitely many answers, but will not withhold control from the user indefinitely. More subtly, a query such as

```
?- append(Xs, [a,b], Xs).
```

will fail to terminate completely, withholding control from the user indefinitely.

This example illustrates two things, firstly that there are several types of possible termination behavior for logic programs and secondly that the termination property is closely related to the *mode* in which the predicate is called. With respect to the second point we note here that in order to perform termination analysis we require that a program be well-moded. For Mercury programs this will always be the case because programs that are not well-moded will be rejected by the compiler's mode analyser.

3.2 Different types of termination

There are two main notions of termination for logic programs: *existential* and *universal* termination.

Definition 1 *A program is said to be* ***existentially terminating*** *if it will produce* at least *one solution, or fail in a finite amount of time.*

After producing at least one solution the program may then not terminate if it is asked for another solution.

Definition 2 *A program is said to be* ***universally terminating*** *if, for all input, it will produce all of its solutions or fail in a finite amount of time.*

Universal termination essentially poses the question "*Will this program always terminate for all input?*". While the universal termination of logic programs has been widely studied, less attention has been focused upon existential termination; [22] provides some references. Universal termination of a logic program implies that it is existentially terminating; the converse is obviously false. The termination analysis systems we discuss in the remainder of this thesis are concerned with proving universal termination.

3.3 But what about the *halting problem*?

The result from basic computation theory known as the *halting problem* tells us that the general problem of deciding if an arbitrary program terminates is undecidable. This limits systems that automatically derive termination proofs to finding *sufficient* but not *necessary* conditions for termination. There have been several systems developed that, with some human intervention, can derive both necessary and sufficient conditions for termination; [10] contains a number of references for these systems. The fact that the only possible source of non-termination in logic programs is recursion makes the problem of finding sufficient conditions for termination at least somewhat tractable. As an aside, it is worth noting that similar techniques to those we describe here have also been applied to functional programs, [12] provides further details.

3.4 Some methodologies for proving termination

There are two main approaches to proving termination for logic programs.

- norm based approaches - these reason about term sizes and try to show that for every recursive call there is a well-founded decrease in the size of the input arguments.

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

append(H1, H2, H3) :-
   (
      H1 == [],
      H3 := H2
   ;
      H1 => [X | Xs],
      append(Xs, H2, Zs),
      H3 <= [X | Zs]
   ).
```

Figure 1: `append(in,in,out)` in superhomogenous form.

- transformational approaches – the termination of term rewriting systems is a well studied problem and transformational approaches to the termination of logic programs take advantage of this fact. The basic idea is to transform a logic program into an equivalent term rewriting system so that a proof for termination of the term rewriting system is also a proof for the termination of the logic program. References on this approach to termination analysis are given in [25].

The methods we consider here are exclusively norm based.

3.5 Reasoning about the termination of `append`

Let us return to the `append` predicate, this time specifying the mode and rewriting the body in superhomogenous from as shown in figure 1. How can we tell if this procedure terminates? A programmer might reason about it thus: At each successive recursive call the length of the list `H1` is getting smaller by one element. Eventually `H1` must be bound to `[]`. At that point `H1` will match the base case of `append`. Since there is no recursive call in the base case and no call to any other predicate which may itself not terminate, `append` must terminate. There are two concepts that we need to define carefully here, that of the "length", or more generally "size", of terms and "smaller".

3.6 Measuring term sizes

In the above example our programmer's proof depended upon the fact that the list `H1` became "smaller" at each recursive call. The "proof" relied on reasoning about the length of the list. While this is a nice way of reasoning for (and about) lists, the concept of "length" does not extend to non-list terms very well. Instead of reasoning about length it is better to reason about the *size* of lists. The notion of *size* extends to other non-list terms in more natural fashion.

One reasonable notion of the size of a list *is* in fact its length. Thus we could say that `[a,b,c]` has a larger size than `[a]` on the basis that it contains more elements. However, perhaps the size of the elements contained *in* the list ought to have some bearing upon the size of the list? In most termination analysis systems term sizes are measured by what is called a *norm*. A norm is a function that maps terms to natural numbers. Here we define the concept of a "semi-linear norm" [3] as a measure of term size. The following definition is taken from [28].

Definition 3 *Let Term be the set of terms, Func be the set of function symbols, and $\mathcal{N}$ be the set of natural numbers (including 0). A* semi-linear norm *is any mapping $\cdot \to \mathcal{N}$ which can be defined using the scheme*

$$\|t\| = \begin{cases} 0 & \text{if } t \text{ is a variable} \\ \Gamma(f/n) + \sum_{i \in I_f} \|t_i\| & \text{if } t = f(t_1, \ldots, t_n), \text{where } I_f \subseteq \{1, \ldots, n\} \end{cases}$$

with $\Gamma : Func \to \mathcal{N}$ being some function that is not zero everywhere.

There are many different norms that follow this scheme. The one that we will use in our all our examples chooses $I_f = \{1, \ldots, n\}$ and

$$\Gamma(f/n) = n$$

3.7 What is "smaller"?

What our programmer has done in the above proof is recognise that is possible to construct a *well-founded* ordering on the set of list lengths. This set is, in theory at least, infinite but it has `[]` as a least element. We briefly recall some basic definitions regarding orderings:

Definition 4 *A* ***partially ordered set*** *$(S, >)$ consists of a set S and a transitive and irreflexive binary relation '$>$' defined on the elements of S.*

Definition 5 *A partially ordered set is said to be* ***totally ordered****, if for any two distinct elements s and r either $s > r$ or $r > s$.*

Definition 6 *A partial ordering on a set '$>$' is said to be* ***well-founded*** *if it admits no infinite descending sequences $s_1 > s_2 > s_3 > \ldots$ of elements of S.*

The canonical example of a well-founded ordering is $(\mathbb{N}, >)$, the set of natural numbers with '$>$' as the relation. Since norms map onto the natural numbers we can see that it possible to place a well-founded ordering on term sizes.

Definition 7 *Let '$>$' be a partial order on some set S. An n-tuple $(s_1, \ldots, s_n)$ with $s_i \in S$ is* ***lexicographically*** *greater than another such tuple $(s'_1, \ldots, s'_n)$ if $s_i > s'_i$ for some $i \in \{1, \ldots, n\}$ while $s_j = s'_j$ for all $j < i$.*

If '$>$' is well-founded then its lexicographic extension, '$>^x$', is also well-founded [23].

3.8 *quicksort*

In figure 2 we give a more complicated example; an implementation of `quicksort`. We can extend our reasoning about `append` to give a proof that this too terminates. Firstly, in order for `quicksort` to terminate, the calls to both `partition(in, in, out, out)` and `append(in, in, out)` must also terminate. We will assume for the time being that they do so.

Following our line of reasoning with the `append` example we need to show that the size of the input to each of the recursive calls of `quicksort` is strictly smaller than that in the head of the procedure. In order to show that such a decrease exists we need to take account of the effect that the call to `partition` has upon the size of the variables.

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

quicksort([], []).
quicksort([X | Xs], Ys) :-
   partition(X, Xs, Littles, Bigs),
   quicksort(Littles, Ls),
   quicksort(Bigs, Bs),
   append(Ls, [X | Bs], Ys).
```

Figure 2: `quicksort`.

We will quickly present some notation. We let x denote the size of the variable `X` and so forth. The size of a variable is always a non-negative quantity. Each term in the program can be replaced by an expression that describes its size. This expression can be calculated by using the norm described in section 3.6. For this example we note that:

$$||\texttt{[]}|| = 0$$

$$||\texttt{[X | Xs]}|| = x + xs + 2$$

The "2" in the second case is the contribution of the size of the list constructor. We can find a similar expression for the size of the term `[X | Bs]`. We abuse the notation slightly by writing $||f(T_1, \ldots, T_n)||$ and meaning the size of any ground components in the term *plus* some symbols representing the sizes of an variables that occur in the term.

Returning to the problem at hand we need to find some relationship amongst the sizes of the arguments of `partition` that will allow us to show a decrease between the head of `quicksort` and the recursive calls. The obvious relationship, which in this case is sufficient, is:

$$xs = littles + bigs$$

Knowing this relationship it is easy to see that:

$$||\texttt{[X | Xs]}|| > ||\texttt{Littles}|| \wedge ||\texttt{[X | Xs]}|| > ||\texttt{Bigs}||$$

Using the above notation this can be expressed as:

$$x + xs + 2 > littles \wedge x + xs + 2 > bigs$$

Since the size of the input argument is strictly decreasing with every recursive call we can infer that `quicksort` terminates. In this particular case the only interargument size relationship that was required in the termination proof was that of `partition`. This is because the size of the output of the first recursive call does not effect the size of the input to the second recursive call. In general this may not be the case. Also, although the relationship we gave for the argument sizes of `partition` is the "obvious" one, it is not the only possible relationship that could describe the interargument sizes.

3.9 A framework for termination analysis

Because of the requirement to know the effect of interargument size relationships (or equivalently interargument size constraints), automatic termination analysis is typically carried out in two phases. The first phase infers the interargument size relationships within a program. The second phase then uses this information to try and establish the termination of the program.

Many other termination analysis systems [5, 11, 20] begin by carrying out some sort of mode analysis. For Mercury this step is unecessary as the Mercury compiler already performs a mode analysis and the results of this are available to the termination analyser. Some work has also been done on the automatic inference of appropriate norms and at least one analyser, [11], also performs automatic norm inference.[2]

For Mercury, since the termination analysis does not need to do its own mode analysis and we do not currently perform any automatic norm inference, the termination analysis is performed in two passes. IR analysis is performed first and then that information is used to try and show termination. Both passes are run on each strongly-connected component (SCC) of the program's call-graph. The SCCs are topologically sorted and the analysis performed in a bottom-up fashion. Results from the bottom of the call-graph are used to try and prove the termination of SCCs further up. In order to incorporate Mercury's module system this scheme must be extended. We discuss how this is done in section 6. For now we will assume that the programs we are analysing are completely defined within a *single* module.

3.10 Termination analysis for Mercury

The current termination analyser for Mercury [28] was based largely on the work of [13, 23], although it made some improvements to those methods. The analysis employs the two stage framework outlined above. Briefly, it works as follows:

In the first stage, for each procedure, it calculates an upper bound on the difference between the total size(s) of the input arguments and the output arguments. In the second stage, it uses this information to attempt to establish the termination of procedures.

The existing analyser was quite notable for the speed of the analysis; not only was it fast but it was uniformly fast. Furthermore this performance was achieved at very little loss of precision. This made it possible for the analysis to be run during every compilation.

For the new analysis our primary interest was improving upon the precision of the old analyser. This means that we need to infer stronger interargument relationships. The cost of doing so is a reduction in the speed of the analysis. In the next two sections we describe the details of the new analysis that we have implemented.

4 IR Analysis using Abstract Interpretation

In this section we describe the IR analysis that we have implemented for the new Mercury termination analyser. This analysis uses abstract interpretation over a domain of convex constraints to derive *linear approximations* to the interargument size relationships.

We begin by describing the abstract domain in which the analysis takes place, then describe the fixpoint iteration before showing how the term-size relationships in the goals of a Mercury program can be abstracted in terms of convex constraints. Finally we consider some examples analyses and discuss how to ensure that the analysis converges. The IR analysis is essentially that of Benoy and King in [2] extended to apply to Mercury, we refer the reader to that paper for correctness proofs of the basic operations over this domain.

[2] Actually, the major contribution of [11] was to show how all the aspects of a typical termination analysis system, mode analysis, norm inference, IR analysis and generating termination proofs could be performed at once by finding solutions to an appropriately generated set of constraints.

4.1 Abstract domain

4.1.1 Convex constraints

The following definitions are adapted from Cousot and Halbwachs [9].

Definition 8 *Let $x_i \in \mathbb{R}$. The set of solutions to the inequality*

$$\sum_{i=1}^{n} (a_i x_i) \leq b$$

with scalars $a_i, b \in \mathbb{R}$, is a ***closed half space*** *of $\mathbb{R}^n$.*

Definition 9 *A* ***linear constraint***[3] *is either a linear equality or a linear inequality. Any linear equality can be readily decomposed into two opposite inequalities.*

Definition 10 *A subset C of $\mathbb{R}^n$ is said to be* ***convex*** *if and only if*

$$\{\forall \tilde{x}_1, \tilde{x}_2 \in C, \forall \lambda \in [0,1], \lambda \tilde{x}_1 + (1-\lambda)\tilde{x}_2 \in C\}$$

Intuitively a set is *convex* if we can join any two points within the set by a line segment such that every point in that segment is also contained within the set.

Definition 11 *The set of solutions to a finite system of linear inequalities in $\mathbb{R}^n$ can be interpreted geometrically as a* ***closed convex polyhedron*** *in some n dimensional space. The polyhedron is formed by the intersection of the closed half-spaces corresponding to each inequality in the system.*

We use the term *convex constraints* to refer to a system of linear inequalities whose solution set can be interpreted as a closed convex polyhedron. The polyhedron may be unbounded. Convex polyhedra can also be represented as a system of rays and vertices [9]. In the past both representations have been used simultaneously while operating over this domain, as some operations are better performed on one form than the other. One of the main contributions of Benoy and King [2] was to show that when implementing the operations within this domain, if a language with suitable constraint support is used, the only representation required is a system of linear inequalities.

4.1.2 Operations on the abstract domain

The following definitions are due to [9, 16]:

Definition 12 *The* ***conjunction*** *of two sets of convex constraints is defined geometrically by the intersection of the convex polyhedra represented by the sets of constraints.*

That is, the intersection defines another polyhedra whose constraint representation is the conjunction of the original sets of constraints. It is well known that the intersection of two convex polyhedra is itself a convex polyhedron.

Definition 13 *The* ***convex hull*** *of two polyhedra P and Q is the* least *convex polyhedron containing both P and Q. We denote the convex hull operation as $\bar{\cup}$, that is, $P \bar{\cup} Q$ is the convex hull of P and Q.*

[3] Cousot and Halbwachs use the term *restraint* rather than *constraint.*

We state some properties of the convex hull operation:

- $P \bar{\cup}\, true = true$
- $P \bar{\cup}\, false = P$
- $(P \bar{\cup} Q) \bar{\cup} R = P \bar{\cup} (Q \bar{\cup} R) \ldots$ (associativity)
- $P \bar{\cup} Q = Q \bar{\cup} P \ldots$ (commutativity)

The convex hull operation gives the *convex* union of two polyhedra. The actual union of any given set of convex polyhedra may be non-convex, the convex-hull is the smallest convex approximation to the actual union.

During the IR analysis we reason about all the variables of a procedure, including those that are local to the procedure. However, once we have obtained a result, we are only interested in the input/output variables. Local variables are removed by *projection*.

Definition 14 *Let $\tilde{x}$ be a set of variables occurring in a set of constraints C. Let $\tilde{y}$ be those variables of C that do not occur in $\tilde{x}$. We denote the* ***projection*** *of the constraints C onto $\tilde{x}$ by:*

$$\exists \tilde{y} C$$

Projection distributes over the convex hull operation. It does not distribute over conjunction.

4.2 Fixpoint calculation

We use a fixpoint calculation to derive the interargument size relationships. We can order the convex polyhedra by using the subset ordering, $\subset$. With this ordering they form a complete lattice with $\wedge$ and $\bar{\cup}$ as the meet and join operations respectively. $false$ is the least element of this lattice and $true$ is the greatest element. The analysis iterates to a fixpoint that characterises the interargument size relationships. The set of constraints representing the interargument size relationships obtained by the i^{th} iteration of the analysis is denoted as I_i. Each iteration of the fixpoint calculation takes I_{i-1} as input and generates I_i as output. The bottom element, I_0, is $false$.

Because the procedures we are analysing will always be in superhomogenous form, the analysis here is simpler than that of Benoy and King. For each iteration we trace through the procedure body, building up a set of implicitly conjoined constraints. We will delay discussion of how to ensure that the fixpoint iteration actually converges to a fixpoint until we have described how a Mercury program is mapped into the abstract domain.

4.3 Abstracting terms

We abstract program variables by representing them as *size variables*. The analyser maintains a map from each variable in the program to its corresponding size variable. All size variables are non-negative integral values. As in the `quicksort` example we use x to denote the size variable that abstracts the program variable `X`. Terms are abstracted by applying the norm to them. For example, abstracting

```
[X | Xs]
```

yields the following:

$$\|[X|Xs]\| = \|X\| + \|Xs\| + 2 = x + xs + 2$$

4.4 Abstracting atomic goals

In what follows we will use Φ_g to denote the set of convex constraints obtained by abstracting the goal `g`. We begin by examining atomic level goals. Mercury has some types of atomic goal that involve calls to foreign code or higher-order constructs. We will delay discussion of these until section 6. The type of atomic goal that we must deal with here are:

- unifications
- procedure calls

The analysis abstracts each atomic goal by replacing it with a set of convex constraints that approximate the effect of the goal upon the size of variables.

4.4.1 Unifications

Unifications are converted directly to constraints approximating their effect on variable sizes. The analysis does not need to make any distinction between the different types of unifications discussed in section 2.

```
A => [X | Xs]
```

Abstracting this goal yields the following constraint:

$$a = 2 + x + xs$$

In general, any unification:

$$\mathtt{X} = \mathtt{f}(\mathtt{T}_1, \ldots, \mathtt{T}_n)$$

can be abstracted as a set of constraints of the following form:

$$\{\|\mathtt{X}\| = ||\mathtt{f}(\mathtt{X}_1, \ldots, \mathtt{X}_n)||\}$$

4.4.2 Procedure calls

The IR analysis distinguishes between two types of call. Calls to procedures that are mutually recursive to the one that is being traced through and calls to those that are not. We call the former *scc-calls* and the latter *non-scc-calls*. Self-calls are *scc-calls*.

In order to abstract a procedure call as a set of convex constraints we substitute the size variables from the call into the constraints for the procedure that is being called. If the calls is a *scc-call* then this set of constraints will change with each iteration. If not, then we will have already completed an IR analysis on the callee procedure. Consequently the constraints on its argument sizes will not change with each iteration. In section 7 we describe how to take advantage of this fact in order to simplify the analysis.

4.5 Abstracting compound goals

Atomic goals are abstracted as sets of convex constraints. The abstraction of compound goals in Mercury corresponds to combining these sets of constraints in such a way that the result is an approximation of the variable sizes in the goal. Certain compound goal types, such as implications, are rewritten by the compiler in terms of other goal types before termination analysis takes place. The goal types we need to consider here are negations, existential quantifications, conjunctions, disjunctions and conditionals.

4.5.1 Negations & existential quantifications

Negations are not of any particular interest during the IR analysis because they will not produce any output; they cannot contribute to the size of any of the terms outside of the negation. For the purposes of termination analysis, we are of course interested if a negated goal contains a call to a non-terminating procedure. Existentially quantified goals are treated as normal goals from the perspective of the IR analysis. We derive argument size constraints for them by examining the quantified goals.

4.5.2 Conjunction

Consider a conjunction of goals `a, b, c`. Each conjunct is approximated as a set of convex constraints. The effect of this conjunction can be approximated by the intersection of these sets. That is for each conjunct `a,b,c` we derive a set of constraints Φ_a, Φ_b, Φ_c such the set of convex constraints approximating the conjunction is given by:

$$\Phi a \cap \Phi b \cap \Phi c$$

4.5.3 Disjunction

Each branch of a disjunction represents a possible branch of computation, consequently the abstraction of each branch may (and usually will) result in different sets of constraints. Consider the goal `q ; r`. The strongest approximation that could be inferred for this disjunction is:

$$\Phi_q \vee \Phi_r$$

It would become prohibitively expensive to have to maintain disjunctive information like this. An alternative, originally proposed by Cousot and Halbwachs [9], is to approximate all the branches of the disjunction by a single over-estimate, the convex hull. The disjunction is abstracted as:

$$\Phi_q \mathbin{\bar{\cup}} \Phi_r$$

4.5.4 Conditionals

In Mercury a conditional goal of the form `if C then T else E` is logically equivalent to

$$(\mathtt{C} \wedge \mathtt{T}) \vee (\neg\exists\mathtt{C} \wedge \mathtt{E})$$

As before, we treat existential quantifications by examining the quantified goal(s). The negated goal, $\neg\mathtt{C}$, will of course not return any solutions, so it cannot contribute to the size of any variables outside the negation. We can therefore abstract the above conditional goal as:

$$(\Phi_C \wedge \Phi_T) \mathbin{\bar{\cup}} \Phi_E$$

Before considering the issue of how we can ensure that the analysis will always converge to a fixpoint, we present a small example where the analysis converges naturally.

4.6 `append` again

Consider the `append` example from figure 1 again: To perform the IR analysis on this we begin by setting:

$$I_0 = false$$

We now trace through the procedure body. The first goal encountered is a disjunction. We must find the sets of convex constraints corresponding to each disjunct and then take the convex hull of the sets. Tracing through the first branch of the disjunction we encounter the following conjunction:

```
H1 == [],
H2 := H3
```

Abstracting the conjunction yields the following set of constraints:

$$\{h1 = 0, h2 = h3\}$$

For the second branch of the disjunction we have the following conjunction:

```
H1 => [X | Xs],
append(Xs, H2, Zs),
H3 <= [X | Zs]
```

Abstracting the two unifications is trivial. What of the recursive call to `append`? We must substitute the size variables representing the arguments, here xs, $h2$ and zs, into the constraints obtained during previous iteration of the analysis. In this case they are $I_0 = false$. So for the second branch we have the obtain the following:

$$\exists x, xs, zs(\{h1 = x + xs + 2\} \wedge false \wedge \{h3 = x + zs + 2\}) = false$$

Projection is used to eliminate size variables that are local to a goal from the constraints. We will omit the projection operation from further examples, assuming that it is performed wherever possible. We complete the calculation of I_1 by taking the convex hull of these sets:

$$\begin{aligned} I_1 &= \{h1 = 0, h3 = h2\} \bar{\cup} false \\ &= \{h1 = 0, h3 = h2\} \end{aligned}$$

The constraints obtained from the first iteration correspond to the argument size relationships entailed by the base case of `append`. We continue the analysis by calculating I_2:

$$\begin{aligned} I_2 &= \{h1 = 0, h3 = h2\} \bar{\cup} \{h1 = x + xs + 2, xs = 0, h2 = zs, h3 = x + zs + 2\} \\ &= \{h1 + h2 = h3\} \end{aligned}$$

This time I_1 is used to approximate the effect of the recursive call on the variable sizes. Continuing the calculation we find that:

$$\begin{aligned} I_3 &= \{h1 = 0, h3 = h2\} \bar{\cup} \{h1 = x + xs + 2, zs = xs + h2, h3 = x + zs + 2\} \\ &= \{h1 + h2 = h3\} \end{aligned}$$

For this example the analysis converges naturally to a fixpoint that describes the interargument size relationships of `append(in,in,out)` after three iterations. [4] However, consider the

[4] This property is *norm* dependent. If we were using a different norm to measure term size this convergence may not occur.

`quicksort` example again. Omitting the intermediate steps for the sake of brevity, the interargument size constraints at each iteration are:

$$\begin{aligned}
I_0 &= false \\
I_1 &= \{h2 \leq 2, h1 = h2\} \\
I_2 &= \{h2 \leq 6, h1 = h2\} \\
I_3 &= \{h2 \leq 14, h1 = h2\} \\
I_4 &= \{h2 \leq 30, h1 = h2\} \\
&\vdots
\end{aligned}$$

In this case the analysis will not converge at all. The constant c in the constraint $h2 \leq c$ will grow larger with each iteration. We can enforce convergence of the analysis by employing widening [7].

4.7 Widening

We adopt the widening for polyhedra defined in [2]. The widening $C \bigtriangledown C'$ selects only those inequalities $i \in C$ that are entailed by C'. That is:

$$C \bigtriangledown C' = \{i \in C | C' \models i\}$$

Typically we do not invoke widening until a few iterations have already taken place. This strategy has been found to improve the precision of the analysis [2]. We call the number of iterations *after* which widening is invoked for the first time, the *widening limit*. Let w be the widening limit and i be the iteration number. When $i > w$ we do the following. We denote the provisional set of constraints obtained by the $i + 1th$ iteration as I'_{i+1}. We generate this set of constraints in the usual manner. We then perform an additional operation to generate the final set of constraints I_{i+1} for the $i + 1$th iteration. The additional operation is defined as follows:

$$I_{i+1} = I_i \bigtriangledown I'_{i+1}$$

Widening constricts the growth of the sets of constraints by selecting only those inequalities in I'_{i+1} that are entailed by those in I_i. After invoking widening the analysis is *guaranteed* to terminate since the sets of constraints cannot grow any further. Setting the widening limit to 4, the analysis of `quicksort` proceeds as follows:

$$\begin{aligned}
I'_4 &= \{h1 = h2, h2 \leq 30\} \\
I_4 &= I_3 \bigtriangledown I'_4 = \{h1 = h2\} \\
I_5 &= I_4 \bigtriangledown I'_5 = \{h1 = h2\}
\end{aligned}$$

For `quicksort` this is precisely interargument size relationship that we would hope for.

4.8 IR analysis involving mutual recursion

Thus far we have only seen examples of the IR analysis that involve direct recursion. The IR analysis extends quite naturally to include those SCCs of the call graph that contain several mutually recursive procedures.

For each procedure in the SCC we maintain a set of constraints. At the beginning the set of constraints for each procedure is $false$. With each subsequent iteration we trace through every procedure in the SCC and generate a new set of constraints. The analysis terminates once every procedure in the SCC has converged to a fixpoint.

```
append(H1, H2, H3) :-                 append(h1, h2, h3) :-
   (                                  (
        H1 == [],                            h1 = 0,
        H3 := H2                             h3 = h2
   ;                                  ;
        H1 <= [X | Xs],                      h1 = x + xs + 2,
        append(Xs, H2, Zs),                  append(xs, h2, zs),
        H3 => [X | Zs]                       h3 = x + zs + 2
   ).                                 ).
```

Figure 3: `append(in,in,out)` with constraints for some atomic goals.

5 Proving Termination

Once we have derived the interargument size relationships we use them to attempt to prove termination. The termination proof we adopt here is based on a direct examination of the constraints that are obtained by abstracting the goals in the body of the procedure under consideration. We present the method through a series of examples.

5.1 Termination of *append*

Consider the procedure `append(in,in,out)` again. In figure 3 we show the constraints obtained by abstracting each atomic goal, except scc_calls, in the body of the procedure. In section 3.5 we argued that it is sufficient for the size of `H1` to decrease at each successive recursive call in order to prove termination. To prove termination in general, it is sufficient to show that there is a strict decrease in some well-founded *measure* of a procedure's input arguments for each recursive call. The well-founded *measure* is often some *linear sum* of the procedure's input arguments but it is also possible to use some well-founded ordered (sub)set of the procedure's input arguments. This latter possibility admits the use of lexicographic orderings for demonstrating a strict decrease in the chosen measure. The analyser can use lexicographic orderings but we will not discuss them here as they add little to our exposition. To formalise the notion of a well-founded measure we introduce the concept of a metric:

Definition 15 *A **metric** for a procedure is some function of a subset of the procedure's input arguments. The chosen subset need not be a proper subset, so it may include* all *of the input arguments. We denote the chosen metric for some procedure* **p** *as* δ_p. *That is:*

$$\delta_p = f(x_i, \ldots, x_n)$$

where $x_i, \ldots, x_n$ *are the sizes of the input arguments of* **p** *and* f *maps into any well-founded ordered set.*

In many termination analysis systems, the function f is chosen so that it maps into the natural numbers; f is chosen to be some linear function of the chosen subset of input arguments. For the purposes of the following discussion we will restrict f to be some linear function that maps into then natural numbers.

Even after limiting ourselves to linear functions that map into the natural numbers, there are still a potentially infinite number of possible metrics. We will further limit ourselves by considering only sums of input arguments; that is linear functions of the input arguments whose coefficients are binary values. [5] Using the notation arg_1 to specify the size of the first *input* argument of a procedure, arg_2 for the size of the second and so on, the possible metrics for `append` are:

1. $\delta_p = arg_1 + arg_2$
2. $\delta_p = arg_1$
3. $\delta_p = arg_2$

In order to show that there is a decrease in this metric we construct, what we call the *termination condition*. This is a strict inequality between the metric applied to the head of the procedure and to the recursive call. In this case there are three possible termination conditions:

1. $h1 + h2 > xs + h2$.
2. $h1 > xs$
3. $h2 > h2$

To prove `append(in,in,out)` terminates it is sufficient to show that one of these conditions is entailed by the constraints implicit in the body of the procedure that occur *before* the recursive call.

Here, the last condition is trivially false and the first two are equivalent. For the first condition we need to check that the following holds:

$$\{a = x + xs + 2\} \vdash a + b > xs + b$$

By substitution it is trivial to verify that this is true. Notice that in proving the termination of `append(in,in,out)` we did not actually make use of the interargument size constraints that were derived during the IR analysis. However, in order to prove the termination of procedures that call `append(in,in,out)` we would need to make use of these constraints.

5.2 Termination of *quicksort*

Consider the `quicksort` example in figure 2 again. Below, this procedure is shown in superhomogenous form. Again the constraints obtained by the abstraction of the atomic goals in the body of the procedure are given. As before scc-calls are untouched.

```
quicksort(H1, H2) :-                   quicksort(h1, h2) :-
    (                                      (
        H1 = [],                               h1 = 0,
        H2 = [],                               h2 = 0,
    ;                                      ;
        H1 = [X | Xs]                          h1 = x + xs + 2,
        partition(X, Xs, Littles, Bigs)        xs = littles + bigs,
        quicksort(Littles, Ls),                quicksort(littles, ls),
        quicksort(Bigs, Bs),                   quicksort(bigs, bs),
        V10 = [X | Bs],                        v10 = x + bs + 2,
        append(Ls, V10, H2)                    h2 = v10 + ls
    ).                                     ).
```

[5] This is what our implementation currently does – with the addition that if this fails it then attempts to use ordered subsets and lexicographic orderings to prove termination.

```
:- type nat ---> zero ; s(nat).

:- pred even(nat::in) is semidet.
even(A) :-
  (
     A == zero
  ;
     A <= s(X),
     odd(X)
  ).

:- pred odd(nat::in) is semidet.

odd(B) :-
   B <= s(Y),
   even(Y).
```

Figure 4: Mutual recursion involving a simple cycle.

There are two recursive calls. Since there is only one input argument in this case, we need only check that it decreases in both recursive calls. We can only prove termination if the input does decrease in *both* recursive calls. This time we will need to use the results of the IR analysis of `quicksort` to show that it terminates. As pointed out in section 3.8, the output of the first recursive call of `quicksort` does not contribute to the input of the second. We require the interargument size constraints on `quicksort` to *ensure* that this is the case. To prove termination we need to show that the following hold:

$$\{h1 = x + xs + 2 \wedge xs = littles + bigs\} \vdash h1 > littles$$

$$\{h1 = x + xs + 2 \wedge xs = littles + bigs \wedge littles = ls\} \vdash h1 > bigs$$

5.3 Mutual recursion - a simple cycle

So far the procedures for which we have given termination proofs have only involved direct recursion. Consider the program in figure 4, which is taken from [23]. It tests if a natural number, represented in successor notation, is odd or even.

The mutual recursion in this SCC forms a single simple cycle. In order to show that all the procedures in this SCC terminate, it is sufficient to show that there is a metric (there is only one possible in this case) that strictly decreases around the cycle, `even` to `odd` to `even`.

Each of the calls from `even` to `odd`, `odd` to `even` has associated with it a set of constraints, $\Phi_{even-odd}$ and $\Phi_{odd-even}$ respectively, that approximate the effect of any goals to the left of the scc-calls. Here $\Phi_{even-odd} = \{a = 1 + x\}$ and $\Phi_{odd-even} = \{b = 1 + y\}$. In order to show that the procedures in this SCC terminate it is sufficient to show that the following holds:

$$\{a = 1 + x\} \wedge \{x = b\} \wedge \{b = 1 + y\} \vdash a > y$$

This condition is true (it can be trivially verified by substitution) so both of the mutually recursive procedures in the program of figure 4 terminate.

```
:- type ---> true ; false ; or(expr, expr) ; and(expr, expr).

:- pred dis(expr::in) is semidet

dis(or(B1, B2)) :-
   con(B1),
   dis(B2).
dis(B) :-
   con(B).

:- pred con(expr::in) is semidet.

con(and(B1, B2)) :-
   dis(B1),
   con(B2).
con(B) :-
   bool(B).

:- pred bool(expr::in) is semidet.

bool(true).
bool(false).
```

Figure 5: Program for parsing boolean expressions.

In general when showing that a given metric decreases around a given cycle it is not necessary that the metric decrease along every edge in the cycle. Indeed along some edges the metric *may* actually increase. This is acceptable as long as an increase in the metric along one edge is counteracted by (a sufficiently large) decrease along some of the other edges in the cycle. What we need to ensure is that when *all* the edges of the cycle are taken into consideration there is a strict decrease in the chosen metric. It should be obvious that the existence of a decrease in the cycle is independent of where we begin in the cycle. In the case of the program in figure 4 showing that there is a strict decrease around the cycle `even` to `odd` to `even` is sufficient to prove that both `odd` and `even` terminate. However when the SCC contains more than one cycle this may not be the case.

5.4 More mutual recursion

Consider program for parsing boolean expressions in figure 5, taken from [10]. The procedures `dis` and `con` are mutually recursive. Their relationship is more complicated than that of the previous example. In addition to calls to each other they also call themselves. The procedure `bool` is not involved in the mutual recursion and obviously terminates since it involves no recursion.

In order the see the relevant relationships in this program, we will construct a directed graph (actually a multigraph) with a vertex for every procedure and an arc between any two vertices `p` and `q`, if there is a call from procedure `p` to `q`. Figure 6 shows the resulting graph for the program in figure 5. In this case we have labelled each edge with the call that it represents. Normally we

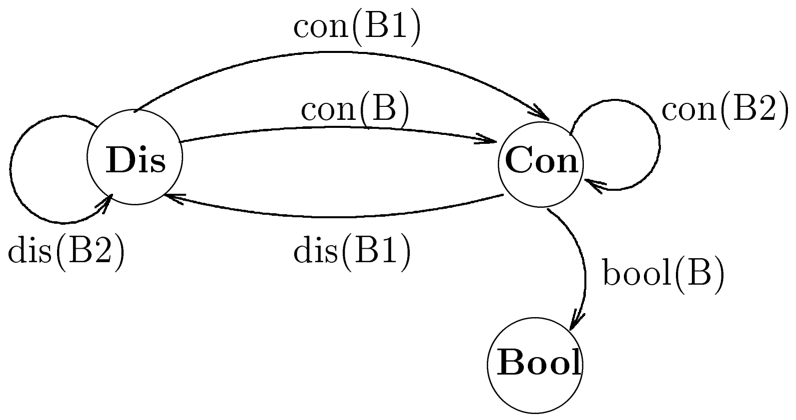

Figure 6: Multigraph showing the calls in figure 5.

label each edge with the set of constraints that occur to the left of the call represented by the edge. This allows us to reason about the sizes of arguments in a very natural fashion.

The graph in figure 6 contains four elementary cycles. In order to prove termination we need to find some metric that strictly decreases around *all* of the cycles. The questions arises, which metric? Do we start with the procedure `dis`, select some subset of its input arguments and try to show that they strictly decrease around all cycles or do we start with `con` and try something similar? In this case we are fortunate, since both `con` and `dis` have only a single argument. This means that it does not matter where we start looking for a decrease. Showing that the size of the input argument decreases around the cycles `dis` to `dis`, `con` to `con`, `dis` to `con` and the other `dis` to `con` cycle is sufficient to prove termination. It must be so because no matter which path the computation follows the size of the *single* input argument decreases. In the general case, where procedures have multiple input arguments, the situation is more complicated.

5.5 Mutual recursion - the true story

Consider the abstract program in figure 7. Each of the procedures, `p` and `q` has multiple input arguments. There are three elementary cycles in the SCC, `p` to `p`, `p` to `q` to `p` and `q` to q; we will denote the cycle `p` to `p` as `p-p` and so forth. How do we show that there is a strict decrease in some measure of the input around all the cycles of this SCC? In the previous example we could choose a metric based on either `dis` or `con` and check that it decreased around all cycles. However, the only reason doing so worked is because `dis` and `con` have only one argument. In that case if something is decreasing it *must* be that argument; there is no other choice. Here the situation is not so simple. If, as before, we choose a metric based on one procedure we will be unable to show that there is a decrease around all of the cycles.

Consider what happens when we choose δ_p, a metric based upon `p`'s input arguments and then try to show a strict decrease around all cycles. For both of the cycles that involve the procedure `p` it is quite easy to see if δ_p is strictly decreasing or not. What of the self-cycle involving `q`? It is possible that the size variables in δ_p becoming very large around the cycle `q-q`. Moreover, there is nothing to stop the computation traversing `q`'s self-cycle a large number of times. The problem here is that given we have some metric δ_p that strictly decreases around all the elementary cycles that p is involved in (in this case `p-p` and `p-q-p`), we need to find conditions such that:

1. The number of consecutive repeated traversals of the self-cycle `q-q` is finite. That is `q` only

```
:- mode p(in, ..., in).

p(a1, ..., an) :-
   (
         ⋮
      q(b1, ..., bm)
   ;
         ⋮
      p(c1, ..., cn)
   ).

:- mode q(in, ..., in).

q(d1, ..., dm) :-
   (
         ⋮
      p(e1, ... en)
   ;
         ⋮
      q(f1, ... fn)
   ).
```

Figure 7: Mutual recursion in general.

calls itself a finite number of times before again recursively calling p or the computation halting.

2. The (possibly) repeated self-calls to q do not cause the value of δ_p to increase by such an amount that the effect of the calls from p to q and from q to p upon δ_p do not counteract this increase.

It should be clear that we cannot apply δ_p to q's self-cycle directly; for one thing q has (possibly) a different number of input arguments and in any case we do no know the exact correspondence between the input argument positions of p and those of q. In all our previous examples we have been fortunate in that the procedure on which we based our metric was always involved in *all* cycles of the call-graph. To some extent this allowed us to avoid the issue. Our problem in this case is that p is clearly not involved in the cycle q-q.

One solution, that in practice works quite well, is to simply set a metric for each procedure in the SCC as being the sum of *all* of the sizes of the input arguments. In the example of figure 7, we would set δ_p to be the sum of all the sizes of the input arguments of p and similarly δ_q to be the sum of all the sizes of the input arguments of q. It is then sufficient to prove termination if δ_p strictly decreases around all those cycles containing p and δ_q strictly decreases around all those cycles involving q. This approach succeeds in many cases, however it is certainly possible to envisage a situation where a decrease in the size of one set of input arguments is counteracted by an increase in the size of another set and the sum the argument sizes remains constant.

We now present an alternative method for proving the termination of the program in figure

```
reverse([], L, L).
reverse([X|Xs], L0, L) :-
   reverse(Xs, [X | L0], L).
```

Figure 8: `reverse` with an accumulator.

7, that does not rely on the metrics consisting of all of the input arguments. We need to ensure that the number of consecutive self-calls to `q` is finite. We can ensure this by making sure that there is a metric δ_q that strictly decreases around `q-q`. We also need to ensure that however many times the computation consecutively traverses the cycle `q-q` it does not cause δ_p to increase in such a fashion that δ_p is then not strictly decreasing around all the cycles involving `p`. There are four conditions necessary to achieve this:

1. Around all cycles involving `p`, δ_p strictly decreases.
2. Around the cycle `q-q`, δ_q strictly decreases.
3. The choice of the metric δ_p and the constraints along every path from `p` to `q` place an upper bound on the size δ_q.
4. The choice of the metric δ_q and the constraints along every path from `q` to `p` place an upper bound on the size of δ_p.

We conjecture that if we can find a set of metrics, $\{\delta_p, \delta_q\}$, that satisfy these four conditions then this is sufficient to prove termination in this case. It seems likely that if this is the case, then this approach can be extended to encompass more complicated SCCs.

5.6 Strategies for choosing the termination metric

In section 5.1 we saw that in general a metric could be any linear sum of the input arguments (in fact if we admit lexicographic orderings, any ordered subset of them). Because this choice of metric was too large to be practical we limited the choice to the sum of a subset of the input arguments. This still leaves us a rather large selection of possible metrics. There are several strategies we could try when selecting a metric. The analyser currently uses a brute force approach. It begins by making the metric the sum of *all* the input arguments. If this approach fails, the analyser searches through all possible subsets of the input arguments, beginning with single arguments and gradually increasing the size of the subsets. For each subset the analyser attempts to see if the sum of the input arguments in the subset exhibit a strict decrease. If all of this fails, the analyser again generates subsets of the input arguments and attempts to use lexicographic orderings to show a strict decrease. [6] We leave lexicographic orderings until last because they are often unnecessary and more expensive to use than the simple sums.

In practice this strategy has been found to work rather well. In the `list` module from the Mercury standard library, of those recursive procedures that can be shown to terminate, only one cannot be shown to terminate by choosing the metric to be the sum of all input arguments. [7] The one *recursive* procedure that cannot be shown to terminate using the sum of all the input arguments is `reverse` using an accumulator: In order to show a decrease in this case we need

[6] Although for lexicographic orders we obviously do not generate singleton subsets as these have already been checked.

[7] The procedures that cannot be shown to terminate make use of higher-order constructs.

consider only the first input argument.

Consider the metrics that work for proving the termination of `append`. Both the sum of the first two arguments, and just the first argument can be shown to decrease. In fact both of these metrics are equivalent because the size of `H2` does not change in the recursive call. This is not perhaps so useful in this case but it does suggest a way of reducing the size of the search space for the metrics. If we can identify argument positions in which the argument size is going to be the same size in both the procedure head and any recursive calls, then we can remove that argument from the set of input arguments from which we construct our metric.

In the case of *append* we can tell the second argument will not change size because it is the same variable in the head and in the recursive call. In general though, operations may be performed on the argument so that although the identity of the argument changes between the procedure head and the recursive call, its size may not. This is quite prevalent inside the Mercury compiler for example, where large dictionary type data structure are often passed between predicates without their size being altered. [8] We reduce the search space for useful metric by identifying and removing these argument positions from consideration.

6 Extending the analysis

In this section we examine how we can extend the techniques described in previous sections to deal with aspects of Mercury such as modules, higher-order code and foreign code.

6.1 Modules

Until now we have assumed that our programs contain no calls to procedures defined in other modules. Mercury was designed to support the development of large multi-module programs. We cannot rely on the assumption that the whole program will be accessible to the termination analyser at one time. Doing so would result in failure to prove termination in a large number of cases due to dependencies on procedures defined in other modules.

Speirs, Somogyi and Søndergaard [28] addressed this problem by writing the termination information about every procedure exported from a module to an auxiliary file. Lindenstrauss and Sagiv [20] also used a similar idea. Because the information contained in the auxiliary file of module A contains information derived not only from module A, but also from any other modules imported by module A, from modules imported by those modules and so on transitively, and because the information used in these files was intended for optimisation, these auxiliary files were called trans_opt files. A module whose source is stored in `foo.m` will have an auxiliary file named `foo.trans_opt`.

For the new analyser we must write two pieces of information about every procedure exported from the module into the trans_opt file.

1. Argument size constraints for the procedure.
2. The termination status of the procedure.

The work of [28] places two major restrictions on the use of the .trans_opt files. The first restriction concerns what happens when the information contained in a .trans_opt file changes. In some cases these changes will affect the termination status of procedures whose termination status was inferred on the basis of information from the trans_opt file that changed. The solution adopted in [28] for the old analyser was to remove the trans_opt files and object files of every

[8] One place this occurs is in the new termination analyser itself.

module in the program when such a change occurs. Although this approach is conservative it was considered acceptable because changes that effect the termination properties of procedures are relatively rare.

The second restriction placed upon .trans_opt files is the correct handling of recursive dependencies amongst the modules of a program. These dependencies occur when module A imported module B and vice versa. The following scenario from [28] illustrates this problem: Suppose module B defines the procedure q, which always terminates, and module A defines the procedure p via the clause p :- q. Obviously p also terminates. Suppose then that the definition of q is changed to q :- p. When recompiling module B, the compiler should not read the trans_opt file for module A. If it did so it would read that p always terminates; it would then conclude that q also always terminates, which is incorrect. This problem is caused by the fact that the compiler cannot see the loop between p and q. The solution adopted was to impose an ordering on the modules of a program. This means that if module A occurs after module B in this order, then the information from module B's trans_opt file can be used to derive information to be put in module A's trans_opt file, but not vice versa. In the above case where there is a loop in the module dependency graph the inconsistency caused by the loop was resolved by imposing an ordering on the modules in the loop. The fact that it is necessary to impose an ordering on the modules of a program comes at a cost. It effectively means that neither the old or new analyser can prove the termination of mutual recursion when it occurs across module boundaries. For the old analyser there is little that can be done to avoid this restriction. For the new analyser we conjecture that by writing out not only the termination status and argument size constraints to the trans_opt file, but also the constraints entailed by the *body* of the procedure, it ought to be possible to prove the termination of mutually recursive predicates across module boundaries.

6.2 Higher-order code

As mentioned in section 2, Mercury supports the use of higher-order code through variables that can have a predicate (or function) value. The use of higher-order code can present some difficulties for the termination analysis. At the moment the approach taken by the analyser with procedures that involve higher-order code is very conservative. It assumes that if a procedure contains such a call, then the call may not terminate.

Potentially we could do better than this. A alternative approach would be to perform a global analysis of the program and work out exactly what the value of any predicate or function valued variables could possibly be. If any of the possible values of the predicate or function valued variable do not terminate then we know that the higher-order call itself cannot terminate. We can extend this reasoning to the IR analysis by approximating the effect of the higher-order call by taking the convex hull of the interargument size constraints of the possible values of the higher-order variable.

6.3 Foreign code

Mercury allows the code of some predicate and functions to be written in a foreign language. This language is usually C but could alternatively be C++, C# or MSIL. The termination analysis cannot analyse any of these foreign languages. In order not to affect the analysis of any Mercury code that relies on these predicates the analyser assumes, unless it is told otherwise, that they do terminate. It also assumes that there are no constraints, other than trivial non-negativity constraints, on the sizes of the arguments. This second assumption is less easy to justify and may result in the analyser being unable to prove termination of other predicates that call the predicate that contains the foreign code. One solution to this is to allow programmers to supply the constraints for procedures that are defined in a foreign language.[9]

7 Implementation

The implementation consists of approximately 5000 lines of Mercury code. Because Mercury does not have builtin constraint support it was necessary to build a module that provided the necessary machinery for manipulating sets of constraints. Instead of using constraints over the real numbers, we use constraints over the rational numbers. It has been found that the use of real numbers tends to lead to a loss of precision [2, 9]. We now discuss some aspects of the implementation that are of interest.

7.1 Constraint operations

7.1.1 Projection

We implement projection using Fourier-Motzkin elimination; we refer the reader to [27] for the details of the basic algorithm. This method is known to generate a large number of redundant constraints as variables are eliminated. We adopt the variation of the algorithm described in [18] that eliminates many (although not all) of the redundant constraints.

7.1.2 Convex hull

The method used for calculating the convex hull is that of [2]. This method has the advantage that the polyhedra require only a single representation; a set of linear inequalities. Earlier approaches to calculating the convex hull, for example [9], required the dual representation of the polyhedra.

The convex hull calculation is the most expensive operation performed by the analysis. The method described in [2] introduces many auxiliary variables and the main expense incurred is due to having to eliminate these variables via projection. The efficiency of the convex hull calculation can be improved by using the fact that projection distributes over the convex hull operation. In calculating

$$\exists \bar{x}(\Phi_{x_1} \bar{\cup} \cdots \bar{\cup}\, \Phi_{x_n})$$

where $\bar{x}$ is a set of variables that we wish to eliminate, it is useful to use this distributivity property and push the projection "inwards" as follows:

$$(\exists \bar{x}\Phi_{x_1}) \bar{\cup} \cdots \bar{\cup}\, (\exists \bar{x}\Phi_{x_n})$$

Doing this may reduce the number of variables that the convex hull calculation needs to introduce, improving the efficiency of the subsequent projection operation.

7.2 Simplification of Mercury programs

The Mercury compiler represents programs internally using what is known as the High-level Data Structure (HLDS). This is the Mercury program transformed into superhomogenous form and annotated with various types of information such as types and modes. Our implementation splits the IR analysis into two sub-passes. The first sub-pass traverses the HLDS representation of the program and builds up an abstract representation of the program. This abstract representation

[9]We currently do not allow this.

contains only the information about the program that is relevant to the termination analyser. The initial sub-pass also checks that procedures do not contain calls to other procedures that are known to be non-terminating. The second sub-pass performs the fixpoint iteration using this simplified abstract representation.

Each Mercury program can be reduced to an abstract program that consists of four types of goal:

1. `primitive` - a set of convex constraints, possibly derived from a unification or call to non-scc procedure.

2. `scc_call` - a call to a procedure in the same SCC of the call-graph as the procedure being examined.

3. `conjunction` - a conjunction of abstract goals.

4. `disjunction` - a disjunction of abstract goals.

There are several reasons for using this simplified representation for the remainder of the IR analysis:

- Avoiding the repetition of various low-level operations such as generating constraints from unifications.

- For calls to non-scc procedures we need only perform the substitutions once during the initial pass.

- We can remove negated goals altogether, after checking that they do not call non-terminating procedures.

- Since pass 2 of the termination analysis requires the term size constraints in the body procedure we can just look them up using this abstract representation, rather than having to generate them yet again.

8 Experimental Evaluation

In this section we present the results of running the analyser on a set of small test programs. The programs, originally written in Prolog, were gathered together by Lindenstrauss and Sagiv [20]. They were gathered from other papers on termination analysis [1, 4, 10, 23]. The programs were translated to Mercury when the old termination analyser [28] was written. Tables 1 and 2 show our results. The first three columns specify the benchmark set, the program within the benchmark set and the query form. The fourth and fifth columns describe the results of running the old termination analyser. The fourth column gives the result of the analysis. Using the notation in [20], the results are specified as follows:

- `T` means that the procedure matching the query always terminates and that the analyser can find a proof of this.

- `N+` means that the procedure cannot be shown to terminate by the analysis because the procedure is actually non-terminating.

- `N-` means that the procedure cannot be shown to terminate by the analysis although it will always terminate.

As usual, by "terminate" we mean universal termination. The fifth column shows the time (in seconds) taken by the old termination analyser to analyse that program. (This time covers all query forms in the program, so if a program has more than one, it is listed only against the first.) The sixth and seventh columns give the corresponding result and time for running the new termination analyser on the same program. For the new analysis the widening limit was set to 4 for most of the test programs with the exception of *ack*, for which it was set to 3 and for *pl4_5_2*, for which it was set to 2. The reason why this was necessary will be described shortly.

The test programs were run on machine with a 1600Mhz Intel Pentium IV CPU, 256kb L2 cache and 512Mb RAM which was running Debian GNU/Linux (testing). In order to obtain the timings, each analyser was run on each test program 10 times. The highest and lowest times obtained for each test were then discarded. The final result was calculated as the average of the remaining times. Both analyses (particularly the old one) ran so fast on some examples that it was not possible to obtain accurate timing information. In those cases the time is given as 0.00s.

As expected, the new analyser is slower than the old analyser. In most cases the performance difference is not particularly noticeable. There are several test programs for which the performance of the new analyser degrades significantly. In the cases of *pl5_2_2*, which simulates a Turing machine, *pl7_6_2a* and *pl7_6_2b*, both of which relate to graph reachability, the performance degradation seems to be due to a combinatorial explosion in the number of constraints generated during Fourier-Motzkin elimination. For these three program we are confident that the performance of the new analyser can be improved.

For *pl4_5_2*, a program that evaluates simple arithmetic expressions, the performance degradation is so severe that unless the widening limit is set quite low, the analysis will fail to terminate in a reasonable amount of time. Again, this behavior is due to combinatorial explosion in the number of redundant constraints generated during Fourier-Motzkin elimination.

For the test program *ack*, which is an implementation of Ackermann's function, the new analyser can prove termination, while the old analyser cannot. In order to obtain this result we had to set the widening limit to 3, rather than the usual setting of 4. With the widening limit set to the latter value the growth of the constraint sets during Fourier-Motzkin elimination resulted in unacceptably slow performance. The *ack* program is an example that requires the use of a lexicographic ordering in order to prove termination.

For the test programs *pl4_5_3a* and *pl3_5_6a*, the results indicated that the new termination analysis is marginally faster than the old analysis. For these programs, both analyses are so fast that it is difficult to obtain accurate timing measurements. Consequently it is likely that this finding is not particularly significant.

Proving the termination of mergesort has often posed a problem for termination analysis systems. The test suites contains five varied implementations of this particular algorithm. For the implementation in the program *mergesort* neither analyser can prove termination. Figure 9 shows the code for this version of mergesort. Both analyses can prove that `split` and `merge` terminate. The difficulty in proving the termination of mergesort is that the interargument relationships concerning `split` are not strong enough. Specifically for both analyses, they do not ensure that when the input argument of split is a list containing two or more elements then both of the output arguments contain at least one element. For the new analyser the problem is that the constraints we derive admit the possibility of the size of either `X1s` or `X2s` being zero.

This specific problem is addressed in the program *mergesort_t*, where the `split` predicate is transformed. Here the new analyser can prove termination but the old analyser cannot. The test programs *mergesort_ap*, *pl8_2_1* and *pl8_2_1a* are also implementations of mergesort. The results of both analyses are the same for these three programs.

```
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).
```

Figure 9: mergesort predicate from *mergesort* test program.

The only test program for which the old analyser can prove termination when the new analyser cannot is *pl8_4_2*. The reason for this is that the constraints derived by the IR analysis of the new analyser are not strong enough to show a decrease around some of the cycles in the call-graph. It is not clear if it is possible to improve the IR analysis so as to obtain better results for this program. Finally we should note that due to a bug in pass 2 (at the time of writing), it was not possible to obtain results for the *vangelder* program for the new analyser. Pass 1 of the new analyser runs on the *vangelder* in a time comparable to that of the old analyser.

9 Further Work

There are several improvements we hope to make to the analyser. The most critical work to be done involves addressing the performance degradation that sometimes occurs with the projection operation. Although we have made progress in this area, limiting the growth of the constraint sets is still of major importance if the analysis is to be truly useful.

As mentioned in section 5.6 we can refine the choice of metric by working out which argument sizes do not change between the head of a procedure and the recursive call(s). We intend to make use of this idea to reduce the search space for the metric. Another idea we intend to implement is that described in section 6.1 for detecting the termination of mutual recursion across module boundaries. This should be reasonably straightforward particularly as the abstract representation described in section 7.2 is almost exactly what needs to be written out to the trans_opt files to allow this to work.

We described how the onset of widening is delayed for several iterations during the IR analysis. This improves the precision of the fixpoint calculation. Conversely there are some cases, the `quicksort` example for instance, in which delaying the onset of widening results in carrying out more iterations than are strictly necessary to obtain a precise characterisation of the interargument size constraints. Benoy and King [2] conjectured that for certain classes of predicate the IR analysis will infer all common invariants within a specific number of iterations. If ways could be found to efficiently detect the optimum widening limit then we could potentially speed up the IR analysis.

10 Conclusion

In this thesis we have described a new termination analysis system for Mercury. The key aspect of our system is the use of convex constraints to approximate interargument size relationships. This allows us to describe these relationships with a greater degree of precision than the existing analyser. The use of convex constraints also leads to termination proofs that are somewhat more natural than those of the old analysis.

We have compared the performance of this new analyser against the old Mercury termination analyser and have found that in some cases the new analyser can infer termination when the old analyser cannot. The new analyser runs at a speed that is, in general, comparable to that of the old analyser although its worst-case performance is clearly quite bad.

Finally, we have outlined how the termination analysis we have described in this thesis could be extended to improve some aspects of the termination analysis of Mercury programs, such as the detection of the termination of mutual recursion across module boundaries.

11 Acknowledgements

I would like to thank my supervisors, Zoltan Somogyi and Harald Søndergaard for their help and guidance on this project. I would also like to thank Vanessa Teague and Thomas Conway for their earlier work on the implementation of the analyser and Nicole Ronald for proofreading the early drafts of this thesis.

Suite	Program	Query	Old Result	Old Time	New Result	New Time
apt	append	append(out,out,in)	T	0.00s	T	0.02s
apt	fold	fold(in,in,out)	T	0.01s	T	0.06s
apt	list	list(in)	T	0.00s	T	0.01s
apt	lte	goal	T	0.00s	T	0.02s
apt	map	map(in,out)	T	0.00s	T	0.03s
apt	member	member(out,in)	T	0.00s	T	0.03s
apt	mergesort	mergesort(in,out)	N-	0.01s	N-	0.22s
apt	mergesort_ap	mergesort(in,out)	T	0.01s	T	0.40s
apt	naive_rev	reverse(in,out)	T	0.01s	T	0.05s
apt	ordered	ordered(in)	T	0.01s	T	0.02s
apt	overlap	overlap(in,in)	T	0.01s	T	0.03s
apt	permutation	perm(in,out)	T	0.00s	T	0.11s
apt	quicksort	qs(in,out)	T	0.01s	T	0.38s
apt	select	select(out,in,out)	T	0.01s	T	0.04s
apt	subset	subset(in,in)	T	0.00s	T	0.06s
apt	subset	subset(out,in)	N+		N+	
apt	sum	sum(out,out,in)	T	0.01s	T	0.04s
bgh	mmatrix	mmultiply(in,in,out)	T	0.01s	T	0.09s
bgh	money	solve(out)	T	0.01s	T	0.12s
bgh	occur	occurall(in,in,out)	T	0.01s	T	0.10s
dds	dds1_2	permute(in,out)	T	0.00s	T	0.06s
dds	dds3_13	duplicate(in,out)	T	0.00s	T	0.02s
dds	dds3_14	sum(in,in,out)	T	0.00s	T	0.11s
dds	dds3_15	merge(in,in,out)	T	0.01s	T	0.06s
dds	dds3_17	dis(in)	T	0.00s	T	0.06s
dds	dds3_17	con(in)	T		T	
dds	dds3_8	reverse(in,out,in)	T	0.00s	T	0.04s

Table 1: Results of analysing the test suites

Suite	Program	Query	Old Result	Old Time	New Result	New Time
plu	mergesort_t	mergesort(in,out)	N-	0.01s	T	0.47s
plu	pl1_1	append(in,in,out)	T	0.01s	T	0.04s
plu	pl1_1	append(out,out,in)	T		T	
plu	pl1_2	perm(in,out)	T	0.01s	T	0.10s
plu	pl2_3_1	p(in,out)	N+	0.01s	N+	0.00s
plu	pl3_5_6	p(out)	N+	0.00s	N+	0.01s
plu	pl3_5_6a	p(out)	N+	0.00s	N+	0.02s
plu	pl4_01	append3(in,in,in,out)	T	0.01s	T	0.05s
plu	pl4_01	append3(out,out,out,in)	T		T	
plu	pl4_4_3	merge(in,in,out)	T	0.00s	T	0.05s
plu	pl4_4_6a	perm(in,out)	T	0.00s	T	0.06s
plu	pl4_5_2	s(in,out)	N+		N+	
plu	pl4_5_3a	p(in)	N-	0.01s	N-	0.00s
plu	pl5_2_2	turing(in,in,in,out)	N+	0.01s	N+	13.69s
plu	pl6_1_1	qsort(in,out)	T	0.01s	T	0.37s
plu	pl7_2_9	mult(in,in,out)	T	0.02s	T	0.05s
plu	pl7_6_2a	reach(in,in,in)	N+	0.01s	N+	8.01s
plu	pl7_6_2b	reach(in,in,in,in)	N-	0.00s	N-	35.42s
plu	pl7_6_2c	reach(in,in,in,in)	T	0.00s	T	0.79s
plu	pl8_2_1	mergesort(in,out)	N-	0.01s	N-	0.54s
plu	pl8_2_1a	mergesort(in,out)	T	0.01s	T	0.36s
plu	pl8_3_1	minsort(in,out)	N-	0.01s	N-	0.09s
plu	pl8_3_1a	minsort(in,out)	T	0.01s	T	0.10s
plu	pl8_4_1	even(in)	T	0.00s	T	0.02s
plu	pl8_4_1	odd(in)	T		T	
plu	pl8_4_2	e(in,out)	T	0.01s	N-	0.02s
misc	ack	ack(in,in,out)	N-	0.01s	T	11.81s
misc	arit_exp	e(in)	T	0.00s	T	0.12s
misc	associative	normal_form(A::in,B::out)	N-	0.01s	N-	0.13s
misc	queens	queens(in,out)	T	0.01s	T	0.09s
misc	vangelder	q(in,in)	N-	0.01s	-	—-

Table 2: Results of analysing the test suites

References

[1] K.R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. In *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.

[2] F. Benoy and A. King. Inferring argument size relationships with CLP(R). In J. Gallagher, editor, *Logic Program Synthesis and Transformation: Proc. LOPSTR 96*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223. Springer-Verlag, 1997.

[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*, volume 494 of *Lecture Notes in Computer Science*, pages 153–180. Springer-Verlag, 1991.

[4] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of global analysis in strict independence-based automatic program parallelization. In M. Bruynooghe, editor, *Proc. Int. Logic Programming Symp.*, pages 320–336. MIT Press, 1994.

[5] M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs bye constructrion or approximation of fixpoints. In *Conf. Record of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[7] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 1990.

[8] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

[9] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. Fifth ACM Symp. Principles of Programming Languages*, pages 84–96. ACM Press, 1978.

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

[11] S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based termination analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 21(6):1137–1195, 1999.

[12] J. Giesl, C. Walther, and J. Brauburger. Termination analysis for functional programs. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume 10 of *Applied Logic Series*, chapter 6, pages 135–164. Kluwer, 1998.

[13] G. Gröger and L. 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.

[14] N. Halbwachs. Delay analysis in synchronous programs. In *Computer Aided Verification*, pages 333–346, 1993.

[15] N. Halbwachs, Y.E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *First International Static Analysis Symposium*. Springer Verlag, 1994.

[16] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[17] F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1996. A more recent version may be found at http://www.cs.mu.oz.au/mercury.

[18] J. Jaffar, M.J. Maher, P.J. Stuckey, and R.H.C Yap. Projecting CLP(R) constraints. *New Generation Computing*, 11:449–469, 1993.

[19] A. King, K. Shen, and F. Benoy. Lower-bound time-complexity analysis of logic programs. In J. Małuszyński, editor, *Logic Programming: Proc. 1997 Int. Symp.*, pages 261–275. MIT Press, 1997.

[20] N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs (with detailed experimental results). http://www.cs.huji.ac.il/~naomil; Shorter version to appear in L. Naish, editor, Logic Programming: Proc. Int. Conf. Logic Programming, MIT Press, 1997.

[21] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, New York, second edition, 1987.

[22] M. Marchiori. Proving existential termination of normal logic programs. In *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 375–390. Springer-Verlag, 1996.

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

[24] L. Plümer. Automatic termination proofs for logic programs operating on nonground terms. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proc. 1991 Int. Symp.*, pages 503–517. MIT Press, 1991.

[25] M.R.K. Krishna Rao, D. Kapur, and R. K. Shyamasundar. Transformational methodology for proving termination of logic programs. *Journal of Logic Programming*, 29(1–3):1–41, 1998.

[26] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analysing the termination of definite logic programs with respect to call patterns. In *Proc. FGCS'92, International Conference on Fifth Generation Computer Systems*, pages 481–488, 1992.

[27] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

[28] C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. In P. Van Hentenryck, editor, *Static Analysis: Proceedings of the Fourth International Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 1997.

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

[30] A. Van Gelder. Deriving constraints among argument sizes in logic programs. *Annals of Mathematics and Artificial Intelligence*, 3, 1991.