Termination Analysis for Mercury

Chris Speirs, Zoltan Somogyi and Harald Søndergaard*

Dept. of Computer Science, University of Melbourne, Parkville Vic. 3052, Australia

Abstract. Since the late eighties, much progress has been made in the theory of termination analysis for logic programs. However, the practical significance of much of this work is hard to judge, since experimental evaluations rarely get published. Here we describe and evaluate a termination analyzer for Mercury, a strongly typed and moded logic-functional programming language. Mercury's high degree of referential transparency and the guaranteed availability of reliable mode information simplify termination analysis. Our analyzer uses a variant of a method developed by Plümer. It deals with full Mercury, including modules and I/O. In spite of these obstacles, it produces state-of-the-art termination information, while having a negligible impact on the running time of the compiler of which it is part, even for large programs.

1 Introduction

The purpose of termination analysis is to answer the question "is this predicate guaranteed to terminate for this class of queries?" Answers to such questions are useful in their own right, as indications of possible programming errors. They may also be useful during program optimization, as it is not usually acceptable for an optimizing compiler to transform a terminating program to one that fails to halt. This is a significant concern when implementing program transformations that change the order of calls in a logic program.

We report on our experience with automatic termination detection for the logic-functional programming language Mercury. We have implemented a termination analyzer and incorporated it into the Mercury compiler. Our method is mainly based on the work of Gröger and Plümer [9]. However, we have made a number of modifications to that method which have significantly increased its precision. We explain both the original approach and our enhancements.

The implementation of a termination analyzer for a full programming language raises problems that have not been discussed so far, including the handling of modules and I/O. This paper therefore includes what is to the best of our knowledge the first experimental evaluation of a termination analyzer applied to large "real-world" programs that exploit all the features of a "real-world" logic and/or functional programming language. The results are very satisfactory. The precision of our analysis matches that of other recent proposals, and the analysis is extremely fast: it takes less than a second on source files containing more than 3,000 lines of code.

Section 2 introduces Mercury. Section 3 describes our termination analysis algorithm. Section 4 discusses how we handle real-world language features that are usually glossed over in the literature. Section 5 gives experimental results.

* This research has been supported by ARC grant no. A49601340.

2 Mercury

Mercury [10] is a pure logic-functional 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 and mode systems, and its support for higher-order programming.

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 we allow programmers to define functions, the compiler internally translates functions with n arguments into predicates with $n + 1$ arguments, the extra argument being the return value.

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 and predicates exported from the module; the implementation part contains the definitions of the exported types and predicates as well as the declarations and definitions of any types and predicates private to the module. Modules cannot access anything that is private to another module; they can access the entities exported from another module only by explicitly importing that module.

Mercury has a strong type system based on many-sorted logic with parametric polymorphism. It is quite conventional, being very similar to the type systems of languages such as ML, Haskell and Gödel.

The mode of a predicate describes which arguments should be computed from which other arguments. An argument which is ground when the predicate is called has the mode "in", while one whose value is supposed to be computed by the predicate has 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 two different modes of a predicate often don't have much in common (they often have different numbers 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 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. The caller promises to pass ground terms as the input arguments of calls to the procedure. The procedure promises to return ground terms as the output arguments of calls to the procedure. When processing 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 program point.

Mercury supports higher-order programming in the style of functional programming. A predicate may take other predicates as arguments (both input and output), its definition may contain lambda expressions each defining a predicate, it can apply currying to existing higher-order terms, and it can partially apply higher-order terms.

The compiler converts any predicate definition with two or more clauses into a single clause with an explicit disjunction. (For simplicity, we will show disjunctions as separate clauses.) The type-, mode and termination analysis algorithms work with clauses that have been converted into *superhomogeneous form*, where each atom (including each clause head) is of one of the forms `p(X1,...,Xn)`, `Y = X`, or `Y = f(X1,...,Xn)`. To the compiler, every atom 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.

3 The termination analysis

We follow the approach proposed by Plümer [12]. The method has two stages:

1. Assign an integer to each procedure, giving the upper bound on the difference between the total size of the output arguments and the total size of the input arguments. This involves producing and solving a set of linear inequalities.
2. Use the solution to check that in every cycle in the program's call graph, the input arguments decrease in size.

More precisely we use an improved version of the algorithm given by Gröger and Plümer [9]. For Gröger and Plümer, there is a further initial stage, since programs must be shown to be "well-annotated in the strong sense". Fortunately the mode analysis component of the Mercury compiler already performs an equivalent check, and our termination analyzer has full access to this mode information.

We have not said exactly what "smaller" means in the context of termination analysis. However, the basic idea is obvious: "smaller" has to be with respect to some well-founded ordering. The concept of a "semi-linear norm" [4] is a common base for most proposals for termination orderings.

Definition 3.1 Let $Term$ be the set of terms, $Func$ the set of function symbols, and $\mathcal{N}$ be the set of natural numbers (including 0). A *semi-linear norm* is any mapping $\|\cdot\| \rightarrow \mathcal{N}$ which can be defined using the scheme

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

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

Many different norms follow this scheme. The two we have used always choose $I_f = \{1, \ldots, n\}$ and they both choose $\mathit{\Gamma}(f/n)$ to be 0 for $n = 0$; with $\mathit{\Gamma}(f/n)$ for $n > 0$ being 1 for one norm and n for the other. We have found that the choice has very little impact on the performance of the analyzer. For example, the norms yield the same results for all of the benchmark programs discussed in section 5. In our examples below, we will use the norm where $\mathit{\Gamma}(f/n) = 1$.

We now explain the essentials of the two stages through examples. After that we present the detailed algorithms.

3.1 First stage

The purpose of Stage 1 is to compute information about the relative size of a procedure's input and output. To compute this, we need relationships between input and output arguments of each atom used in the procedure's definition.

Obviously, these relationships are not immediately available for recursive procedures. However, by introducing variables that stand for the relative sizes, we can turn the problem into one of creating a set of constraints that are subsequently solved, giving the values of the variables. This can be done in a compositional fashion, if we use *inequalities* as constraints. This approach has been common since Ullman and Van Gelder's seminal work [15]. We follow Plümer in using slightly more general inequalities than did Ullman and Van Gelder: A predicate inequality is 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 k of predicate p, I is a subset of the set of the input argument positions of predicate p, and J is a subset of the set of its output argument positions. Initially we will choose J to be the set of all output arguments and I to be the set of all input arguments, but later we will relax this and consider making I a proper subset of the input arguments. The aim is to solve for an integer value of γ_p, which is the value that will serve as an indicator of the maximum possible growth of arguments. Since we want γ_p to be as tight as possible, we will pick the smallest value (which may be negative) that satisfies the inequalities we infer for the predicate. We call the γs *change variables* or *change constants* depending on whether we have fixed their value yet.

The process of inferring these predicate inequalities is a backwards dataflow analysis. The annotation that we generate at each program point is a linear inequality of the form above, with the sum of the sizes of the input arguments plus the γ associated with the current procedure on the left hand side. The right hand side is initially the sum of the sizes of the output arguments, but we will update it during the backward scan. In general, it is the sum of the sizes of a bag (multiset) of variables, plus an integer expression composed of change variables and change constants, as well as integer constants representing the weights of function symbols.

The algorithms we use represent the annotation at each program point as a pair (V, e) where V is the bag of *active* variables, that is, those that contribute their size to the predicate's output variables, and e is an integer expression. Since the left-hand side is the same at each program point, the algorithms need not carry it along. Our examples will also omit the left hand sides; they will use the shorthand that x denotes the size of the term bound to `X`.

We illustrate how the process works by tracing how we process `append` in the mode `(in,in,out)`. Let us start with the recursive clause:

```
append(H1, H2, H3) :-          % h1 + h2 + γappend + 1 + (−1)
    H1 => [X | Xs],            % x + xs + h2 + γappend + 1
    append(Xs, H2, Zs),        % x + zs + 1
    H3 <= [X | Zs].            % h3
```

The annotation on the program point at the end of the clause reflects the right hand side of the initial form of the predicate inequality for append: $h_1 + h_2 + \gamma_{append} \geq h_3$. The last literal '`H3<=[X|Zs]`' defines the value of an active variable (the only one). We therefore remove h_3 from the right hand side and replace it with $h_3 = x + zs + 1$ (the 1 represents the size of the cons cell) in the annotation for the program point before the unification. This is justified, as the sizes of terms that have been unified must be the same. The recursive call also defines one of the active variables, Zs. We don't know the size of Zs precisely, but we do know an upper bound for it: $xs + h_2 + \gamma_{append}$. When deriving the annotation on the program point before the call, we exploit the fact that we can replace a variable by its upper bound on the right hand side of an inequality. Finally, 'H1=>[X|Xs]' tells us that $h_1 = x + xs + 1$, so we can replace $x + xs$ on the right hand side by $h_1 + (-1)$. The inequality we infer for the clause is the annotation for the program point at the start of the clause: $h_1 + h_2 + \gamma_{append} \geq h_1 + h_2 + \gamma_{append} + 1 + (-1)$, which is equivalent to the vacuous constraint *true*.

Note how our analysis depends on mode information, both for calls and for unifications. In general, we derive different constraints for different modes of the same predicate.

The annotations for the first clause are:

```
append(H1, H2, H3) :-          % h2
    H1 == [],                  % h2
    H3 := H2.                  % h3
```

The only thing to notice here is that the literal 'H1==[]' has no output variable which is currently active (in fact it has no output variable at all), and therefore processing it has no effect on the annotation. Therefore the inequality for the clause is $h_1 + h_2 + \gamma_{append} \geq h_2$.

In general, other clauses from the given procedure, and from procedures mutually recursive with it, will contribute similar constraints. The task then is to minimize the change variables while satisfying the constraints. The constraint above is simply $h_1 + \gamma_{append} \geq 0$. Since $h_1 \geq 0$, minimizing γ_{append} yields $\gamma_{append} = 0$. The change constant 0 indicates that whenever append(in,in,out) succeeds, the difference in size between output and input arguments is bounded from above by 0, so the output arguments cannot be larger than the input arguments. This interpretation, and the fact that $\geq$ is transitive, guarantees that backwards analysis with its step-by-step substitutions preserves correctness.

The information given by $\gamma_{append} = 0$ can be used when analyzing procedures that call append(in,in,out). To make this possible, the analysis considers the strongly connected components (SCCs) in the program's call graph, processing each SCC in turn, in topologically sorted order, from the bottom up.

For example, consider the analysis of double(in,out), which uses the newly determined value of γ_{append}:

```
double(Xs, Zs) :-              % xs + xs
    Ys := Xs,                  % xs + ys
    append(Xs, Ys, Zs).        % zs
```

The derived inequality of the clause is $xs + \gamma_{double} \geq xs + xs$. Since the single occurrence of xs on the left hand side of the inequality for the clause is not sufficient to satisfy the demand $xs + xs$ on the right hand side, the minimal solution is $\gamma_{double} = \infty$. This agrees with our intuition that if we consider successful calls to double, there is no finite upper bound on the difference in size between the arguments. As it turns out, double is still guaranteed to terminate as it does not call itself directly or indirectly, but this sort of reasoning is the task of the method's second stage.

3.2 Second stage

The call graph for append has a single node and a single arc from this node to itself. Initially we label all arcs in the call graph by $-\infty$. For each procedure we then identify calls to procedures in its own SCC. In our example, the only such call is the directly recursive call to append. The analysis starts from that call and works backwards in a way similar to the first stage's analysis. However, since we now want to prove that the input arguments of the recursive call are strictly smaller than the input arguments of the head, the implicit left hand side of the inequality contains just the sizes of the input variables of the head (no γ) and the inequality is not $\geq$ but a strict $>$. The analysis produces these annotations:

```
append(H1, H2, H3) :-          % h1 + h2 + (-1)
    H1 => [X | Xs],            % xs + h2
    append(Xs, H2, Zs),
    H3 <= [X | Zs].
```

The atom '`H1=>[X|Xs]`' defines the value of `Xs` and tells us that $h_1 = x + xs + 1$, that is, $xs = h_1 - x + (-1)$. Unfortunately, we do not know the size of x; all we know is that (like all sizes) it is greater than or equal to zero. Therefore the least upper bound on $-x$ is zero. This is why the annotation for the program point at the start of the clause is $h_1 + h_2 + (-1)$.

The constraint $h_1 + h_2 > h_1 + h_2 + (-1)$ is obviously satisfiable. It also tells us that the size of input arguments of append decreases by at least 1 on each recursive call. We therefore change the label on the edge in the call graph representing the recursion in append to -1.

The SCC for append has a loop. However, it has only loops in which the input arguments are guaranteed to strictly decrease in each iteration. So we conclude that append terminates in this mode. Similarly double only calls terminating procedures, and its SCC has no cycles at all, so it must terminate.

3.3 Tracking used variables

In the next section we give the algorithms we use to execute the two stages. The debt we owe to Gröger and Plümer's Algorithms 3.3 and 5.1 [9] will be clear to anybody familiar with their work, but our algorithms differ in a number of important ways:

1. Gröger and Plümer use bags of variables to check that input variables are not used excessively, witness the double example. However, the bags can just as well provide information about variables that are *not* used. This can increase the precision of the analysis. In the algorithm we therefore restrict our attention to the input variables whose size can affect the sizes of the output variables.
2. The Mercury compiler translates programs to a form in which there is only one clause per procedure. In one respect this simplifies the algorithm, but it also means we have to handle conditionals, disjunctions, and switches. To keep things simple, we disregard these features in the algorithm below—in section 3.5 we then discuss these features and how they are dealt with.
3. We have improved the algorithms in a number of minor ways, such as turning the core of Stage 2 into a single backwards pass, rather than a sequence of backward passes. We also disregard this improvement until section 3.5.

As regards point (1), consider the annotations of the quicksort program, as they would be generated by a naive Stage 1 analysis.

```
:- mode split(in, in, out, out) is det.
:- mode qsort(in, out) is det.

split(Pivot, List, Lo, Hi) :-    % 0
    List == [],                  % 0
    Lo := [],                    % lo
    Hi := [].                    % lo + hi

split(Pivot, List, Lo, Hi) :-    % pivot + list + \gamma_{split} + 1 + (-1)
    List => [Head | Tail],       % head + pivot + tail + \gamma_{split} + 1
    Pivot > Head,                % head + pivot + tail + \gamma_{split} + 1
    split(Pivot, Tail, Lo0, Hi), % head + lo0 + hi + 1
    Lo <= [Head | Lo0].          % lo + hi

split(Pivot, List, Lo, Hi) :- ...% (similar code and analysis)

qsort(In, Out) :-                % 0
    In == [],                    % 0
    Out := [].                   % out

qsort(In, Out) :-                % in + head + 2 \gamma_{qsort}
    In => [Head | Tail],         % 2 head + tail + 2 \gamma_{qsort}
    split(Head, Tail, Lo, Hi),   % lo + hi + head + 2 \gamma_{qsort}
    qsort(Lo, SLo),              % slo + hi + head + \gamma_{qsort}
    qsort(Hi, SHi0),             % slo + shi0 + head
    SHi <= [Head | SHi0],        % slo + shi
    append(SLo, SHi, Out).       % out
```

The analysis of `split` above tells us that the procedure contributes change 0 and uses `Pivot` and `List` to produce output. This leads the analysis of `qsort` to the inequality $in + \gamma_{qsort} \geq in + head + 2\gamma_{qsort}$, and there is no finite value of γ_{qsort} that makes the inequality true for all values of the other variables. This requires the analysis to consider γ_{qsort} to be ∞, which can significantly degrade the accuracy of the analysis of the callers of `qsort`. This type of problem was also identified by Plümer [12], and quite likely by others before him.

The problem is that the above analysis does not distinguish between those input arguments whose values contribute to the size of the output arguments and those that do not. In this case, the first argument in the call to `split` does not affect the size of any of the output variables of `split`. If we modify our analysis to take this into account, we derive the inequality $in + \gamma_{qsort} \geq in + 2\gamma_{qsort}$. Together with the inequality $in + \gamma_{qsort} \geq 0$ from the first clause, this yields $\gamma_{qsort} = 0$.

3.4 The algorithms

Our algorithm has two stages. In Stage 1 we execute the algorithm shown in figure 1 for each SCC in the call graph. In Stage 2 we execute the algorithm shown in figure 2, again for each SCC. Note that Step 1 utilizes a result by

```
Global variables:
S = {p1, ..., pn} is the SCC being analyzed
I is the generated linear inequality
output_suppliers gives for each procedure the subset of input variables
    that are needed to produce output – this is used by Stage 2

For each procedure p ∈ S set new_output_suppliers(p) to [ ]
Do
    old_output_suppliers := new_output_suppliers
    Step1(old_output_suppliers, new_output_suppliers)
while new_output_suppliers ≠ old_output_suppliers
output_suppliers := new_output_suppliers
If I is unsatisfiable then for each p ∈ S set γp to ∞
else set (γ1, ..., γn) to (d1, ..., dn),
    where (d1, ..., dn) is the least solution satisfying I

Procedure Step1(in old_output_suppliers, out new_output_suppliers):

I := true
For j := 1 to n:
    Let   B0 :- B1, ... Bm   be the clause defining pj   (m ≥ 0)
    M := outvars(pj)
    δ := 0
    For i := m to 1:
        If M ∩ outvars(Bi) ≠ ∅ then
            M := (M \ outvars(Bi)) ∪ old_output_suppliers(Bi)
            δ := δ + change(Bi)
    If M ⊈ invars(pj) then for each p ∈ S set γp to ∞ and exit
    new_output_suppliers(pj) := M
    If δ contains ∞ then I := false
    else I := I ∧ γpj ≥ δ
```

Fig. 1. The first stage

Gröger and Plümer, showing the existence of an integral least solution to a satisfiable linear inequality I of the given form ([9], Corollary 4.4).

Both algorithms use a function *change*, defined as follows. When its argument is a call to a procedure q, the value of the function is γ_q, which will be a variable or a known value depending on whether q is in S. When its argument is a construction (a unification of the form `Out <= f(In1,...,Inn)`), the value of the function is $\Gamma(f/n)$. When it is a deconstruction (a unification of the form `In => f(Out1,...,Outn)`), the value of the function is $-\Gamma(f/n)$. When it is a test or an assignment unification, the value of the function is zero.

In Plümer's algorithm, the first stage of the termination analysis is based around the inequalities

$$\sum \textit{input variables} + \gamma \geq \sum \textit{output variables}$$

```
For each procedure p ∈ S, new_recursive_input_suppliers(p) := invars(p)
Do
    old_recursive_input_suppliers := new_recursive_input_suppliers
    Step2(old_recursive_input_suppliers, new_recursive_input_suppliers)
while new_recursive_input_suppliers ≠ old_recursive_input_suppliers
Remove from the call graph all edges with negative weight
Mark all procedures in S as always terminating provided that
    the modified call graph of S has no edges with strictly positive weight
    and the modified call graph of S is cycle free
    and every procedure in a lower SCC called from S is terminating

Procedure Step2(in old_rec_input_suppliers, out new_rec_input_suppliers):

Assign the weight −∞ to every arc in the call graph for S
For j := 1 to n:
    Let    B0 :- B1, ... Bm    be the clause defining pj    (m ≥ 0)
    For i := 1 to m:
        If Bi is a call to a procedure r in S then
            M := old_rec_input_suppliers(r)
            δ := 0
            For k := i-1 to 1:
                If M ∩ outvars(Bk) ≠ ∅ then
                    M := (M \ outvars(Bk)) ∪ output_suppliers(Bk)
                    δ := δ + change(Bk)
            If M ⊈ old_rec_input_suppliers(pj) or δ > 0
            then weight(pj, r) := ∞
            else weight(pj, r) := max(δ, weight(pj, r))
    new_rec_input_suppliers(pj) := M
```

Fig. 2. The second stage

In our implementation, the first stage of the analysis is instead based around the inequalities

$$\sum \textit{output supplier variables} + \gamma \geq \sum \textit{output variables}$$

The *output supplier variables* are the variables representing the input arguments whose values contribute to the size of the output variables. We calculate the set of output supplier variables as a fixed point. First we assume that each procedure in the current SCC uses none of its input arguments to produce output. We then run the first pass of the analysis, and update the set of output suppliers of each procedure from the final value of M (which must be a set if it is a subbag of the set $invars(p_j)$). We repeat this process until we reach a fixed point.

Plümer's original inequalities are not always as strong as they could be. For example, when analyzing `split`, the final inequality would be "$pivot + list + \gamma_{split} \geq lo + hi$". By using the fixed point analysis, we can form a set of constraints which are at least as strong as the original inequalities, while still correct. After calculating the output supplier variables for `split`, our termination analysis generates the stronger "$list + \gamma_{split} \geq lo + hi$", which leads to a change constant of 0 for `qsort`.

The aim of Stage 2 is to prove that the size of a set of input variables strictly decreases in each loop of the call graph. This does not have to mean *all* input variables. A second fixpoint calculation can derive a better set of input variables for each procedure in the SCC. Each set must be consistent in the sense that for any (mutually) recursive call in the SCC, the chosen input variables for the head suffice to determine the size of the chosen input variables for the call. We call the variables chosen for a procedure the *recursive input suppliers* of the procedure.

The reason for using a least fixed point to calculate the output suppliers in Stage 1 is that it gives the strongest possible linear inequalities which remain sound. For Stage 2, we can choose any subset of the input variables to be the recursive-input suppliers, but a set that is too small will cause the last step of the iteration in Step 2 to compute weights that significantly degrade accuracy. For example, choosing the recursive-input suppliers to be all empty sets would give a trivially valid least fixed point, but the algorithm would compute a weight of zero for all edges, which implies the possibility of an infinite loop. Choosing recursive-input suppliers sets that are not consistent would mean that M is not a subbag of this set and would thus set the weights of edges to ∞, which also implies the possibility of an infinite loop. We therefore use a greatest fixpoint calculation to give us a consistent set of variables that is big enough; this seems to give quite good accuracy.

3.5 The true story

The algorithms as given assume that each procedure is defined by a single clause, and that the clause contains only conjunctions. In the internal representation used by the Mercury compiler, each procedure is defined by a single clause, so the first assumption is valid. However, the body of the clause may contain if-then-elses, disjunctions, switches (indexable disjunctions), negations and existential quantifications as well as conjunctions. Negations and existential quantifications do not affect our algorithm, but the other constructs do, since they establish multiple execution paths. To handle these, our algorithm carries along not a single δ/M pair, but a set of such pairs, one for each possible execution path. New elements in the set are created by duplication whenever a backwards traversal encounters the end of an if-then-else, disjunction or switch; they are kept even after traversal encounters the front of the construct. In the first stage, there is an element in the set for each possible program execution path. In the second stage there is an element in the set for each possible program path that leads to a (possibly mutually) recursive call. Whenever the algorithm as shown operates on a single δ or M, the actual implemented algorithm performs that operation on each element in the set.

The implemented version of the algorithm Step 1 calculates a single final value of M for each procedure by taking the *least upper bound* on the bags M that it calculates for each execution path. Similarly, its version of the last statement in figure 1 adds to I the requirement that γ_{p_j} must be greater than or equal to the δ value it calculated for every control path. Our implementation solves the final conjunction of constraints using a third-party package called *lp_solve*, a version of the simplex algorithm implemented in C [2].

Another refinement is that in the implementation of Step 2, we avoid the repeated traversal of clauses implied by the algorithm in figure 2. As these separate traversals are independent, they can be executed in parallel. As the implementation already used sets of δ/M pairs, this improvement was easy to implement.

Finally, we have enhanced the method by adding a simple analysis which is run whenever the main termination analysis implementation fails to prove termination of a procedure. It can only handle directly recursive procedures, and proves termination of these procedures if any one of the input arguments is smaller in the recursive call than in the head of the procedure. This simple analysis does not use the values calculated in either stage of the main analysis, and fails to prove termination if there is a call between the recursive call and the head. However, it occasionally proves termination where the main method fails.

4 Real-life aspects of termination analysis

The algorithm in the previous section shows how our termination analysis works for pure first-order logic programs. However, real-world programming languages contain several features that complicate termination analysis. In Mercury, these features are higher-order constructs, which hide the identities of the predicates called through predicate-valued variables; the module system, which prevents the analysis from seeing all of the program at once; and the foreign language interface, which allows the code of some predicates to be written in a language (C) which is not amenable to analysis.

The current version of our termination analyzer makes the conservative assumption that higher order calls may not terminate. For our ideas on how to improve this, as well as for improving the handling of I/O (which is defined via foreign language code in the Mercury library), see the full version of this paper [14].

The module system. The Mercury compiler supports separate compilation of modules, since this is essential for the development of large programs. However, separate compilation complicates termination analysis (and other analyses) because the analyzer never has access to the code of the entire program at once. The problem is that when the analyzer works on module A and it finds that predicate p in module A calls a predicate q which is defined in module B, it needs to know the termination properties of q for a proper analysis of p; in the absence of this information, it must assume that predicate p may not terminate.

In a program written in a modular language, almost every predicate calls, directly or indirectly, predicates defined in other modules, including library modules. This problem is therefore very serious. Without a solution, even a perfect termination analysis algorithm would be forced by the lack of inter-module data to return "don't-know" for most predicates in the program. Our solution associates with each module an auxiliary file containing termination information about every procedure exported from the module. This information contains three items:

- a boolean that says whether all calls to the procedure terminate,
- the change constant computed for the procedure (an integer or ∞),
- for each argument of the procedure, a boolean that says whether or not that argument contributes to the size of the outputs (this will be "no" for output arguments as well as for ignored input arguments).

The last item will be missing if the constant for the procedure is ∞.

When the compiler works on module A, it will read in the auxiliary files associated with all of the modules imported into A (but see below). It can then rely on the information in these files about the termination properties of the procedures whose code it does not have access to. When it is finished, the compiler will write out the termination properties it has inferred about the procedures exported from the module A to the auxiliary file of module A. Because the auxiliary file of module A contains information derived not only from module A but also from the modules imported by module A, from modules imported by those modules and so on transitively, and because we intend to use the contents of these auxiliary files for optimization, we call these auxiliary files trans_opt files.

The correct treatment of trans_opt files requires some care, for example in the handling of recursive dependencies among modules. Consider the situation in which module A imports module B and vice versa. Suppose module B defines q, which always terminates, and module A defines p via the clause p :- q. Obviously p also terminates. The trans_opt file of each module will record that the procedure it defines always terminates. Suppose that the definition of q is now changed to q :- p. If, when recompiling module B, the compiler reads the trans_opt file for module A, it will read that p always terminates and incorrectly conclude that q also terminates. The problem is that the loop between p and q is invisible unless one considers both modules at once, which the compiler does not.

Our solution is to impose an order on the modules in the program. If module A is before module B in this order, then information from module B's trans_opt file can be used to derive information to be put into module A's trans_opt file, but not vice versa. Now there can be no circular dependencies among trans_opt files, so the problem cannot occur.

If a module A imports several other modules, the more of these modules that come after module A in the order, the more information termination analysis will have access to when creating module A's trans_opt file. So the chosen order should agree with the module dependencies whenever possible. The only inconsistencies should be those involving loops in the dependency graph. In those cases, we must impose an order on the modules in the loop, thus breaking the cycle. The cost is that if two procedures defined in different modules are mutually recursive and yet both always terminate, our analysis will never be able to prove this.

Foreign language code. Mercury allows the code of some predicates to be written in C. Given that analyzing C code for termination is not a feasible proposition, our termination analysis algorithm must rely on the programmer for information about the termination properties of any procedure defined in C.

Typical use of Mercury's C interface involve short C code fragments such as

```
:- pred length(string::in, int::out) is det.
:- pragma c_code(length(Str::in, Length::out),
   "Length = strlen(Str);").
```

Most of these code fragments do not even contain loops, and the functions they call are usually library functions. Programmers who write such code will want termination analysis to know that the code always terminates (although in some cases they may be wrong). The analysis therefore assumes that any procedure defined in C will always terminate, unless the programmer says otherwise.

Since the termination analysis algorithm cannot analyze C code, it has no way of deriving information from C code about the relative sizes of the input and output arguments. In the absence of information, the only thing it can do is set the change constants of procedures defined in C to ∞. This is what the current implementation does. However, programmers should be able to supply this information to the analyzer if they wish to override the default.

5 Experimental evaluation

Lindenstrauss and Sagiv [11] have gathered a large set of small Prolog programs that have previously been used by other researchers [1, 7, 8, 12] to test the performance of termination analysis algorithms. We have translated most of these programs to Mercury in order to try out our implementation of termination analysis. (The others either depend on non-logical features of Prolog, or are not well typed and/or well moded according to Mercury's rules.) For three of the benchmark sets, our termination analysis gives answers which are identical to the ones given by Lindenstrauss and Sagiv. For the fourth benchmark set (the one from [12]) the accuracy of our analyzer is comparable to theirs; while we obtained a more precise result for `perm(in,out)`, they obtained a more precise result for one particular variant of `mergesort(in,out)`.

The algorithm of Lindenstrauss and Sagiv [11] is usually reasonably fast (taking 0.1 to 0.2 seconds), but for a significant number of test programs it is quite slow (taking 20 to 120 seconds). Our termination analysis did not suffer from this problem. On a machine comparable to the one used by Lindenstrauss and Sagiv, it took only 0.3 seconds in the worst case and 0.05 seconds on average.

We also tried out our analyzer on the Mercury compiler and standard library. These are interesting test cases for several reasons. First, they represent real code that people use every day. Second, they make use of the full range of Mercury features. Third, they are much bigger than the test cases for which results have been reported in other papers on termination analysis. With more than 100,000 lines of source, the compiler alone is bigger than all but ten of the 500 programs listed on the Prolog 1000 list of real Prolog applications.

Our timing results show that our analyzer is very fast and has excellent worst-case behavior. On a Sun SPARCstation Ultra 1/170, the average time to perform termination analysis on a module was about a quarter of a second, and no module took more than one second, even though some contain more than 3,000 lines. To the best of our knowledge, this is the first time that such results have been reported for a test performed on a large real-world program. The practical consequence is that termination analysis need not be an action that programmers perform once in a great while; it is cheap enough to be invoked for every compilation.

We believe that neither the library nor the compiler contain non-terminating procedures. We thus assume that if analysis cannot prove the termination of a given procedure, that is the fault of the analyzer and not of the procedure. Our algorithm can prove the termination of most of the procedures in the library (72%) and in the compiler (55%). The reasons for the difference are that whereas most modules in the library are self-contained, the dependencies among the modules in the compiler are quite complex, and that the code within each module tends to be more complex in the compiler than in the library.

For such a large program, we consider the accuracy we have achieved to be very good. For example, the largest program that Lindenstrauss and Sagiv's prototype analyzer [11] handles successfully has 57 clauses; the Mercury compiler is two orders of magnitude larger than that. We believe that matching our level of accuracy for large *Prolog* programs would be very difficult. The main reason is that mode inference is a necessary first step in any termination analysis for an unmoded logic programming language, and the accuracy of mode inference provides an upper bound on the accuracy of the termination analysis algorithm itself.

More details of our experimental evaluation are available in the full version of this paper [14].

6 Conclusion

While much has been written about termination analysis for logic programs, including many ideas for how to improve the precision of analysis, there are very few reports of implementations. The obvious question of whether the theory can be turned into useful practical tools for full programming languages has mainly been left unanswered, as have questions about efficiency/precision trade-offs *in practice*. The only clear exception that we know of is the work by Lindenstrauss and Sagiv [11] who give detailed timing and precision results for a large collection of small Prolog programs. In our design and implementation of a termination analyzer for Mercury, this work has helped us in many ways and set a goal for us to match. In the process, we have had to address the problems of analyzing a full programming language, including features such as modules and input/output which are not normally considered in research papers. On the other hand, some of the features of Mercury, such as the language's referential transparency and the presence of reliable type and mode information, have made the implementation of an analyzer for Mercury considerably simpler than it would have been for other real-world logic programming languages.

For a detailed account of the history of termination analysis for logic programs, and an instructive taxonomy, we refer the reader to the survey paper by De Schreye and Decorte [8]. The approach used in our implementation has its roots in the seminal work by Ullman and Van Gelder [15], and in the development it spurred, including [3, 5, 6, 13]. The immediate inspiration was the work by Plümer [12], and Gröger and Plümer's handling of mutual recursion [9].

We have obtained greatly improved results by enhancing the basic method with two kinds of "usage analysis": in Stage 1 we track input variables whose sizes contribute to the size of the output variables, while in Stage 2 we track input variables whose sizes contribute to the sizes of the input variables in recursive calls. In both cases we achieve precise information by iterating to a fixed point. An important conclusion from our work is that this can be done without incurring a high performance penalty.

We have tested the performance of our analyzer on most of the benchmark programs used by Lindenstrauss and Sagiv [11]. Our results show that our analyzer has comparable accuracy, somewhat better average performance and much better worst case performance. No previous papers on termination analysis that we know of have reported results on large programs, so we used one of our own, the Mercury compiler and standard library. Our timing results are again very satisfactory; our analyzer processed every module in less than a second, even though several modules contain more than 3,000 lines. The level of accuracy we have achieved is also quite good, although there is room for improvement.

If the programmer's intuition says that a procedure always terminates, then the analyzer's failure to prove termination should be scrutinized carefully. Even if the procedure does not have a termination bug, the failure may well be caused by a performance bug. In one case, such scrutiny helped us to identify a performance bug in a frequently used predicate in the Mercury library. Fixing the bug sped up the Mercury compiler by 35%.

We thank Fergus Henderson for stimulating discussions and Naomi Lindenstrauss for responding quickly to questions and for making benchmark programs available.

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. Michel Berkelaar. lp_solve v. 2.0. ftp://ftp.es.ele.tue.nl/pub/lp_solve.
3. Roland N. Bol. *Loop Checking in Logic Programming.* PhD thesis, University of Amsterdam, The Netherlands, 1991.
4. 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*, LNCS 494, pages 153–180. Springer-Verlag, 1991.
5. A. Bossi, N. Cocco, and M. Fabris. Typed norms. In B. Krieg-Brückner, editor, *Proc. ESOP'92*, LNCS 582, pages 73–92. Springer-Verlag, 1992.
6. A. Brodsky and Y. Sagiv. Inference of inequality constraints in logic programs (extended abstract). In *Proc. Tenth ACM Symp. Principles of Database Systems*, pages 227–240. ACM Press, 1991.
7. 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. ILPS'94*, pages 320–336. MIT Press, 1994.
8. Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.
9. 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.
10. F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. The Mercury language reference manual. Technical Report 96/10, Dept. of Computer Science, University of Melbourne, 1996.
11. 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.
12. Lutz Plümer. *Termination Proofs for Logic Programs.* LNAI 446. Springer-Verlag, 1990.
13. Kirack Sohn and Allen Van Gelder. Termination detection in logic programs using argument sizes (extended abstract). In *Proc. Tenth ACM Symp. Principles of Database Systems*, pages 216–226. ACM Press, 1991.
14. C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. Technical Report 97/9, Dept. of Computer Science, University of Melbourne, 1997. http://www.cs.mu.oz.au/publications/tr_db/mu_97_09.ps.gz.
15. 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.