Termination Analysis for Mercury

Chris Speirs, Zoltan Somogyi and Harald Søndergaard
Department of Computer Science
The University of Melbourne
Parkville, Victoria 3052, Australia

Abstract

Since the late eighties, much progress has been made in the theory of termination analysis for logic programs. However, from a practical point of view, the significance of much of the work on termination 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 the termination analysis of Mercury compared with that of other logic programming languages. We describe our termination analyzer, which uses a variant of a method developed by Plümer. It deals with full Mercury, including modules, declarative input/output, the foreign language interface, and higher-order features. In spite of these obstacles, it produces high-quality termination information, comparable to the results recently obtained by Lindenstrauss and Sagiv. Most important, in stark contrast with Lindenstrauss and Sagiv's experimental results, our analyzer has a negligible impact on the running time of the compiler of which it is part, even for large programs. This means that the Mercury compiler can produce valuable termination information at no real cost to the programmer.

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 the program (e.g., constraint propagation [12, 19]). For some time now there has been much interest in automatic termination analysis for logic programs, but it appears that implementations are rare and seldom go beyond the prototype stage.

In this paper 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 [10]. However, we have modified their algorithm slightly, and our modifications have had a great impact on its precision. We explain the Gröger-Plümer approach and the changes we have made.

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 input/output. This

paper therefore includes what is to the best of our knowledge the first experimental evaluation of a termination analyzer applied to a large "real-world" program that exploits all the features of a "real-world" logic-functional programming language. The results, as will be seen, are very satisfactory. The precision of the analysis matches that of other recent proposals. Perhaps more important, the analysis is extremely fast. The time it takes to analyse small-to-medium sized benchmark programs is so small that it cannot be measured reliably, and analysis takes less than a second even on source files containing more than three thousand lines of code.

The structure of the paper is as follows. Section 2 gives a brief introduction to Mercury. Section 3 gives our termination analysis algorithm. Section 4 discusses how we handle real-world language features that are usually glossed over in the literature. In section 5 we consider how our method performs both on a test suite taken from other papers on termination and on a large real-world program, and we compare our results with results obtained by others. Section 6 contains concluding remarks and discusses related work. An appendix gives some anecdotal evidence for the usefulness of termination analysis information.

2 Mercury

Mercury 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-, mode- and determinism 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 representing 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, predicate and functions exported from the module; the implementation part contains the definitions of the exported types, predicates and functions as well 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 by explicitly importing that module.

The type system of 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 predicates/functions exported from modules; the compiler will infer the types 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 arguments should be computed from which other arguments. The arguments which are supposed to be 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 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 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 calls to the procedure. When it analyzes 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, they can apply currying to existing higher-order terms, 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).

For more details on Mercury, please refer to the language reference manual [11].

The compiler converts any predicate definition with two or more clauses into a single clause with an explicit disjunction. The type and mode analysis algorithms work with clauses that have been converted into *superhomogeneous form*, where each atom (including the heads of clauses) is of 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 => [X | Xs],
        append(Xs, H2, Zs),
        H3 <= [X | Zs].
    ;
        H1 == [],
        H3 := H2.
    ).
```

`H1`, `H2` and `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. For the sake of simplicity we will show disjunctions as separate clauses in the rest of the paper.

3 The termination analysis

We follow the overall approach proposed by Plümer [16]. 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 [10]. For Gröger and Plümer, there is a further initial stage, since programs must be shown to be "well-annotated in the strong sense" [10]. Fortunately the mode analysis component of the Mercury compiler already performs an equivalent check, and our termination analysis algorithm, which is also a part of the Mercury compiler, 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. We adopt the variant of the definition used by Gröger and Plümer.

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

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

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

Many different norms follow this scheme. The two we have used always choose ${}_fI = \{1, \ldots, n\}$ and they both choose $\Gamma(f/n)$ to be 0 for $n = 0$; with $\Gamma(f/n)$ for $n > 0$ being 1 for one norm and n for the other. In practice, we have found very little difference between the performance of the analyzer with these two norms. For example, they yield the same results for all of the benchmark programs we present results for in section 5.1. In our examples below, we will use the norm where $\Gamma(f/n) = 1$.

We now explain the essentials of the two stages through examples. After that we present the detailed algorithms, and we close this section by discussing implementation details that in the interest of clarity we have avoided during the presentation of the algorithms.

3.1 First stage

Consider a recursive clause in which one of the input arguments of the recursive call is the output of a non-recursive call. To be able to decide whether the input arguments of the recursive call are smaller than the input arguments of the head of the clause, a termination analyzer must have some knowledge about the sizes of the output arguments of the non-recursive call. The only kind of information about this that seems attainable is a relationship between the sizes of the input and output arguments of the non-recursive call. Since the input arguments of the non-recursive call are a function of the input arguments of the head of the clause, termination analysis algorithms can make good use of information about the relative sizes of the input and output arguments of predicates. The purpose of Stage 1 is to compute such information.

Since a paper by Ullman and Van Gelder [20], researchers in the field have represented this information via predicate inequalities. Here 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 a bound 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, i.e. 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 also 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 => [X | Xs],
    append(Xs, H2, Zs),
    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 one of the active variables (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 OK because the sizes of terms that have been unified must be the same. The annotated form of the clause is now

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

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: $xs + h_2 + \gamma_{append}$. Since we can replace a variable by its upper bound on the right hand side of an inequality, we can update the annotated clause to read

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

Finally, '`H1=>[X|Xs]`' tells us that $h_1 = x + xs + 1$, so we can replace $x + xs$ on the right hand side by $h_1 + (-1)$:

```
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 inequality we infer for the clause is the annotation for the program point at the start of the clause, which is

$$h_1 + h_2 + \gamma_{append} \geq h_1 + h_2 + \gamma_{append} + 1 + (-1)$$

that is, the vacuous constraint *true*.

Note how our analysis depended on mode information, both for calls and for unifications. In general, we derive different constraints for different modes of the same predicate. Because of this, our algorithms work with procedures (individual modes of a predicate), not predicates themselves.

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 this procedure and from procedures mutually recursive with the procedure we are analysing will contribute similar constraints, and the task then is to minimize the change variables while satisfying the constraints. Clearly the constraint above is simply $h_1 + \gamma_{append} \geq 0$. Since $h_1 \geq 0$, minimising 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 bigger 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.

To see how information from a lower SCC is used when analyzing a procedure from a higher SCC, consider `double(in,out)`, where the definition of `double` is

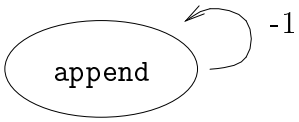

Figure 1: Annotated call graph for `append`

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

The analysis uses the newly determined change γ_{append} to obtain the annotations

```
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, this will not create a termination problem, as `double` is not called through direct or indirect recursion, but this sort of reasoning is the task of the method's second stage.

3.2 Second stage

Let us continue with the `append/double` example. Consider the call graph of append, shown in figure 1. 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 are now trying 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 $h_1 = x + xs + 1$, i.e. $xs = h_1 - x + (-1)$. Unfortunately, we don't 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, in all loops, the input arguments are guaranteed to strictly decrease on every iteration. We therefore conclude that `append` terminates for the given mode. Similarly `double` only calls terminating procedures, and its SCC has no cycles at all, and 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 [10] will be clear to anybody familiar with Gröger and Plümer's work, but our algorithms differ in a number of important ways:

1. In Gröger and Plümer's algorithm, bags of variables serve to check that input variables are not used excessively, witness the `double` example. However, the bags can just as well (or in addition) provide information about variables that are *not* used. This allows us to increase the precision of the analysis. In the algorithm we therefore often 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 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.

In this section we discuss point (1) in some detail. As an example consider the annotations of the quicksort program, as they would be generated by a naive Stage 1 analysis. This is the situation when the analysis reaches the call to `split` in `qsort`:

```
:- 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 + γ_split + 1 + (−1)
    List => [Head | Tail],         % head + pivot + tail + γ_split + 1
    Pivot > Head,                  % head + pivot + tail + γ_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 γ_qsort + 1 + (−1)
    In => [Head | Tail],           % 2 head + tail + 2 γ_qsort + 1
    split(Head, Tail, Lo, Hi),     % lo + hi + head + 2 γ_qsort + 1
    qsort(Lo, SLo),                % slo + hi + head + γ_qsort + 1
    qsort(Hi, SHi0),               % slo + shi0 + head + 1
    SHi <= [Head | SHi0],          % slo + shi
    append(SLo, SHi, Out).         % out
```

The analysis of `split` has told 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 [16], 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 2 for each SCC in the call graph, from the bottom up. Once all SCCs have been processed in this way, we proceed to Stage 2, in which we perform the algorithm shown in figure 3 for each SCC S in the call graph, again from the bottom up. Note that Step 1 utilises a result by Gröger and Plümer which establishes the existence of an integral least solution to a satisfiable linear inequality I of the given form ([10], Corollary 4.4).

The algorithms in both figures use a function *change*, which is defined as follows. When its argument atom 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 atom is a construction, that is, a unification of the form `Out <= f(In1,...,Inn)` the value of the function is $\Gamma(f/n)$. When its argument atom is a deconstruction, that is, a unification of the form `In => f(Out1,...,Outn)` the value of the function is $-\Gamma(f/n)$. When its argument atom is a test or an assignment unification the value of the function is zero. In the internal representation of Mercury, these are the only possible forms of atoms, so this completely defines the function.

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

$$\sum input\ variables + \gamma \geq \sum output\ variables$$

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

$$\sum output\ supplier\ variables + \gamma \geq \sum output\ variables$$

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 by using a fixed point analysis. 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

Global variables:
$S = \{p_1, \ldots, p_n\}$ is the SCC being analzsed
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 Stage2

For each procedure $p \in S$ set $new_output_suppliers(p)$ to [ ]
Do
$old_output_suppliers := new_output_suppliers$
Step1($old_output_suppliers, new_output_suppliers$)
while $new_output_suppliers \neq old_output_suppliers$
$output_suppliers := new_output_suppliers$
If I is unsatisfiable then for each $p \in S$ set γ_p to ∞
else set $(\gamma_1, \ldots, \gamma_n)$ to $(d_1, \ldots, d_n)$,
where $(d_1, \ldots, d_n)$ is the least solution satisfying I

Procedure Step1(in $old_output_suppliers$, out $new_output_suppliers$) is

Set I to $true$
For $j := 1$ to n:
Let B_0 :- $B_1, \ldots B_m$ be the clause defining p_j $(m \geq 0)$
$M := outvars(p_j)$
$\delta := 0$
For $i := m$ to 1:
If $M \cap outvars(B_i) \neq \emptyset$ then
$M := (M \setminus outvars(B_i)) \cup old_output_suppliers(B_i)$
$\delta := \delta + change(B_i)$
If $M \not\subseteq invars(p_j)$ then for each $p \in S$ set γ_p to ∞ and exit
$new_output_suppliers(p_j) := M$
If δ contains ∞ then $I := false$
else $I := I \wedge \gamma_{p_j} \geq \delta$

Figure 2: The first stage

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 then repeat this process until we reach a fixed point.

Plümer's original inequalities always create a set of logically consistent inequalities, but these 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, and are still logically consistent. 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 in recursive calls is

For each procedure $p \in S$, $new_recursive_input_suppliers(p) := invars(p)$
Do
  $old_recursive_input_suppliers := new_recursive_input_suppliers$
  Step2($old_recursive_input_suppliers, new_recursive_input_suppliers$)
while $new_recursive_input_suppliers \neq old_recursive_input_suppliers$
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 call from S to a procedure in a lower SCC is marked as always terminating

Procedure Step2(in $old_recursive_input_suppliers$, out $new_recursive_input_suppliers$) is

Assign the weight $-\infty$ to every arc in the call graph for S
For $j := 1$ to n:
  Let B_0 :- $B_1, \ldots B_m$ be the clause defining p_j $(m \geq 0)$
  For $i := 1$ to m:
    If B_i is a call to a procedure r in S then
      $M := old_recursive_input_suppliers(r)$
      $\delta := 0$
      For $k :=$ i-1 to 1:
        If $M \cap outvars(B_k) \neq \emptyset$ then
          $M := (M \setminus outvars(B_k)) \cup output_suppliers(B_k)$
          $\delta := \delta + change(B_k)$
      If $M \not\subseteq old_recursive_input_suppliers(p_j)$ or $\delta > 0$ then $weight(p_j, r) := \infty$
      else $weight(p_j, r) := max(\delta, weight(p_j, r))$
  $new_recursive_input_suppliers(p_j) := M$

Figure 3: The second stage

strictly smaller than the size of the corresponding set of input variables in the head. This does not have to mean *all* input variables. We can use another fixpoint iteration to derive a set of input variables for each procedure in the SCC. We want these sets to be consistent in the sense that for any (mutually) recursive call in the SCC, the size of the chosen input variables for the head is sufficient 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 logically consistent. For Stage 2, it is valid to choose any subset of the input variables to be the recursive-input suppliers, but sets that are 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 iteration to give us a consistent set of variables that is big enough; this seems to give us 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 don't 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 an algorithm we have shown operates on a single δ or M, the actual implemented algorithm performs that operation on each element in the set.

The actual 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 2 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 version 2.0 of a third-party package called *lp_solve*, which is 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 3. 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 very simple termination 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 the module system, which prevents the analysis from seeing all of the program at once; higher-order constructs, which hide the identities of the predicates and functions called through predicate- and function-valued variables, and the foreign language interface, which allows the code of some predicates and functions to be written in a language (C) which is not amenable to analysis. (The Mercury implementation uses foreign language code to implement I/O.) In this section, we discuss how these features can be dealt with.

4.1 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 program 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 or function calls, directly or indirectly, predicates or functions 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 and functions in the program.

One possible solution is to require the programmer to give to the compiler all the modules in the program at once. This is not a feasible solution for several reasons. First, the program may be so large that even a compiler employing only linear algorithms would have unacceptably large runtime and/or unacceptably large memory consumption. For example, the Mercury compiler and standard library together consist of more than 170 modules totaling more than 120,000 lines and 4.5 Mb of source code; the internal compiler representation of all this code would probably occupy somewhere around a hundred megabytes. Second, the source code of some modules may not be readily available. For example, the usual Mercury installation provides an interface description file and an object file for each module in the standard library, but does not provide source code, since that is of interest only to Mercury developers, not to ordinary users of Mercury.

Our chosen 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 are guaranteed to terminate or not,
- an indication of the change constant associated with the procedure (an integer or ∞), and
- 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 (modulo the issue we discuss below). Its termination analysis algorithm 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, it will write out the results of its analysis as they apply to the procedures exported from the module A to the auxiliary file associated with module A, provided the user has given the compiler the option requesting it to do so.

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; a module whose source is in, say, the file tree234.m will have its auxiliary file named tree234.trans_opt.

This approach requires two restrictions for correctness. The first restriction concerns what happens when the information contained in a .trans_opt file changes. Some changes are innocuous

and require no intervention. However, if the old version of a trans_opt file says that a procedure always terminates where the new one says it may not, if the change constant associated with the procedure is larger in the new trans_opt file than in the old, or if the set of arguments used to produce the outputs includes a given argument in the new trans_opt file but not in the old, then any computations that took the old trans_opt file as input now have to be redone. One way this can be done is by removing the trans_opt files and object files of every module in the program. This is overkill, but it is safe and simple to implement, and it may be an acceptable solution, since we expect that changes that degrade the termination properties of procedures will occur relatively rarely.

The second restriction concerns the correct way of handling recursive dependencies among modules, that is, situations in which module A imports module B and vice versa. The danger can be illustrated by an example. Suppose module B defines procedure q, which always terminates, and module A defines procedure 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. When recompiling module B, the compiler should not read the trans_opt file for module A. If it does, it will read that p always terminates, and will conclude that q also terminates, which is incorrect. 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. Since now there can be no circular dependencies among trans_opt files, 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. Therefore the chosen order should be consistent with the module dependency graph 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, we will never be able to decide that they terminate.

The restriction that when working on module A, the compiler must not read the trans_opt files of modules that precede A in the order applies only when the compiler is generating the trans_opt file for module A. If it is not doing that, but is instead generating code for module A, it *can* read the trans_opt files of modules that precede A in the order, because the object file will not influence the outcome of any other compilation step.

4.2 Higher order constructs

What should termination analysis do with predicates that contain calls through predicate- or function-valued variables?

```
:- pred map(pred(X, Y), list(X), list(Y)).
:- mode map(pred(in, out) is det, in, out) is det.

map(_, [],  []).
map(P, [H0|T0], [H|T]) :-
    call(P, H0, H),
    map(P, T0, T).
```

It is easy to be conservative and say that since the procedure called through `P` may not terminate, `map` may not terminate either. This is what our implementation currently does. If the program uses higher-order facilities sparingly, this may be acceptable. However, in some programs the inability of the analyzer to prove the termination of even a small number of low-level higher-order predicates may propagate up the call graph and thus prevent the analyzer from proving the termination of a large fraction of the predicates in the program. A robust analyzer should therefore be able to deal effectively with higher-order constructs. The basis of such an ability must be information about the termination properties of the procedure bound to the higher-order variable as its value.

If the compiler can see at the same time both the definition of `map` and a call to `map` that specifies the identity of this predicate (say, `add1` on integers), it can create a specialized version of `map` in which the call to `P` is replaced by a call to `add1`, and it can then change the call to `map` to call this specialized version instead. The Mercury compiler already has an optimization that does this. Since the specialized version of `map` has no calls through higher-order variables, it presents no problems for termination analysis, which should therefore be able to show that it always terminates and that its change constant is zero.

Although this approach requires little extra work, it has two flaws. First, since the programmer does not know about the specialized version of the procedure, issuing understandable diagnostic messages about it (for example if even the specialized version does not terminate) is at least somewhat difficult. Second, the compiler does not always have access to both the call to the higher-order procedure and its definition at the same time.

An alternative solution that avoids both problems at the price of significant complexity is to express the termination properties of the higher-order procedure as a function of the termination properties of its procedure-valued arguments. In the case of `map`, this function would say that `map` always terminates if `P` always terminates, that the change constant for `map` is zero if the constant for `P` is zero or negative and infinite otherwise,[1] and that both input arguments are used in constructing the output argument.

4.3 Foreign language code and I/O

Mercury allows the code of some predicates and functions to be written in C. This functionality has two main uses. First, it gives us, the Mercury implementors, a convenient way to implement some of the low-level features of the language, for example the operations on the builtin datatype "string". Second, it gives every Mercury programmer a relatively simple way to interface a Mercury program to code written in other languages, such as tcl and the tk library [15]. (Even if the code one wants to interface to is not written in C, one can usually count on the existence of an interface between that language and C.)

Given that analyzing C code for termination is not a feasible proposition, not least because of lack of access to the definitions of any library functions called in the code, our termination analysis algorithm must rely on the programmer for information about the termination properties of any procedure defined in C. One could require programmers to specify for each such procedure whether it is always terminates or not, but this would be tedious. It would be better for the compiler to assume a value for this boolean, and allow the programmer to override this default assumption.

The typical uses of Mercury's C interface involve short C code fragments, for example

```
:- pred length(string, int).
:- mode length(in, out) is det.
```

[1] Even if the constant for `P` is one, that is, the size of `H` is at most one more than the size of `H0`, this size difference can be repeated as many times as there are elements in the list, and this number is unbounded.

```
:- 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 very frequently library functions. Programmers who write such code will want termination analysis to know that the code will always terminate (although in some cases they may be wrong). We have therefore decided to make termination analysis assume that any procedure defined in C will always terminate, unless there is an indication to the contrary. One possible indication is an explicit directive from the programmer to this effect. Another is a declaration by the programmer that the C code fragment may contain a call back to code written in Mercury. (This facility is necessary, for example to allow programmers to write the code to handle certain kinds of tk events in Mercury.) Since the compiler doesn't know what Mercury predicates may be called in this way, it should not assume that such C code terminates.

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 do so. For example, consider a predicate that takes a string as input and if it is not empty returns the first character and the rest of the string as separate outputs. If the programmer can tell the analyzer that the size of the output string is strictly smaller than the input string, the analyzer ought to be able to show the termination of any recursion in which the head has the whole string and the recursive call has the tail of the string, as returned by this predicate.

This ability to supply the change constant to be associated with procedures defined in C is particularly important for the predicates that perform I/O, which are defined via C code in the Mercury standard library. These predicates have two arguments of type "io__state", which is a builtin abstract type representing the state of the world outside of the Mercury program which the Mercury program can access by performing I/O. As an example, consider the hello world program:

```
:- pred main(io__state, io__state).
:- mode main(di, uo) is det.

main(S0, S) :-
    io__write_string("Hello, ", S0, S1),
    io__write_string("world\n", S1, S).
```

The first argument of `main`, `S0`, describes the I/O state of the computation on entry to `main`, while the second argument `S` describes the I/O state of the computation on exit from `main`. The modes di and uo stand for destructive input and unique output respectively. The di mode of `S0` requires the caller of `main` to supply as first argument an I/O state to which no other references remain; the uo mode of `S` promises that `main` will return as second argument the unique reference to the new I/O state. In the example this new I/O state is reached in two stages, with the intermediate stage `S1` reflecting the I/O state after "Hello, " has been written but "world" has not.

Our main concern here is with the termination analysis of ordinary programs that read all of their input and write output. Although one can force such a program to run forever by giving it an infinite amount of input, this is not the way programs are normally used. It is therefore

much more useful for termination analysis to say "yes, this program always terminates provided its input streams are finite" than to say "this program may run forever" for such programs. We have therefore added an implicit condition to the meaning of the output of our termination analysis algorithm. Whenever it returns "yes", this does not mean "this procedure always terminates"; it means "this procedure always terminates provided the program's input streams are finite".

Given this implicit provision, we should be able to associate negative change constants with predicates that read input, saying that their output I/O state is strictly smaller than their input I/O state. The consequence would be that the analysis will be able to prove the conditional termination of programs that read all their input and perform some provably terminating processing on each input item.

Associating a negative change constant with a predicate that reads input makes intuitive sense if one considers that reading an item removes that item from the input stream. Unfortunately, this is not quite true; if the program has reached the end of a file, the I/O systems of most languages (including Mercury) will on request return a potentially unbounded number of end-of-file and/or error items. However, we can *make* it true by modifying the implementation of the input predicates to allow them to return at most a bounded number of end-of-file and error indications on each file, and to abort the program when one of these limits is about to be exceeded. This safeguard should not be triggered by properly written programs, since predicates that read items in a loop should exit the loop (i.e. do not recurse) when the read predicate returns an end-of-file or error indication.

Real programs of course write output as well as read input. Since output predicates add items to the output streams of the program, one might suppose that one should assign them a positive change constant indicating that their output I/O state is strictly bigger than their input I/O state. However, such a decision would have undesirable consequences. The reason is that input predicates and output predicates use successive versions of the same I/O state. (The order in which I/O operations are carried out is governed by the data dependencies among these successive versions.) If read operations reduced the size of the I/O state but write operations increased it, the analysis would not be able to prove the conditional termination of many programs that mix input and output.

The correct solution is to assign to output predicates the change constant zero, saying that their output I/O state is the same size as their input I/O state. (Assigning a negative constant would be incorrect, since it would allow termination analysis to "prove" the termination of a program that outputs data in an infinite loop.) This can also be justified intuitively, on the grounds that output operations do not increase the amount of data that has to be processed by this program. The only situation in which an output operation does increase the amount of data to be processed by this program is when (some of) the output streams of the program are connected by external plumbing (for example pipes) to the program's own inputs. Given an otherwise terminating program that has been abused in this manner and which as a result runs forever, termination analysis would still be justified in saying "the program always terminates provided its input streams are finite". In this case, the program's input streams are not finite.

5 Experimental evaluation

5.1 Small test programs

For their recent paper [13], Lindenstrauss and Sagiv have gathered together a large set of small Prolog programs that have previously been used by other researchers [1, 7, 8, 16] to test the perfor-

Suite	Program	Query	Time	Us	LS	M/P
apt	append	append(out,out,in)	0.04s	T	T	
apt	fold	fold(in,in,out)	0.03s	T	T	
apt	list	list(in)	0.03s	T	T	
apt	lte	goal	0.04s	T	T	
apt	map	map(in,out)	0.04s	T	T	
apt	member	member(out,in)	0.01s	T	T	
apt	mergesort	mergesort(in,out)	0.04s	N-	N-	
apt	mergesort_ap	mergesort(in,out)	0.07s	T	T	
apt	naive_rev	reverse(in,out)	0.02s	T	T	
apt	ordered	ordered(in)	0.02s	T	T	
apt	overlap	overlap(in,in)	0.03s	T	T	
apt	permutation	perm(in,out)	0.07s	T	T	
apt	quicksort	qs(in,out)	0.05s	T	T	
apt	select	select(out,in,out)	0.01s	T	T	
apt	subset	subset(in,in)	0.06s	T	T	
apt	subset	subset(out,in)		N+	N+	
apt	sum	sum(out,out,in)	0.02s	T	T	
bgh	mmatrix	mmultiply(in,in,out)	0.05s	T	T	
bgh	money	solve(out)	0.30s	T	T	
bgh	occur	occurall(in,in,out)	0.04s	T	T	
dds	dds1_2	permute(in,out)	0.04s	T	T	
dds	dds3_13	duplicate(in,out)	0.02s	T	T	
dds	dds3_14	sum(in,in,out)	0.01s	T	T	
dds	dds3_15	merge(in,in,out)	0.03s	T	T	
dds	dds3_17	dis(in)	0.02s	T	T	
dds	dds3_17	con(in)		T	T	
dds	dds3_8	reverse(in,out,in)	0.04s	T	T	

Table 1: Results of analysing the test suites

mance of termination analysis algorithms. We have translated most of these programs to Mercury in order to try out our implementation of termination analysis. We could not translate them all. The ones we could not translate either depend on non-logical features of Prolog (which are not present in a pure language such as Mercury), or are not well typed and/or well moded according to Mercury's rules.

Tables 1 and 2 show our results. The first three columns specify the benchmark set (each set corresponds to one of the tables in [13]), the program within the benchmark set, and the query form. The fourth column shows the time taken by our termination analysis algorithm for 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 fifth column gives the result of our termination analysis algorithm. If the algorithm can prove that the procedure corresponding to the query form always terminates, the column contains T; if it cannot do so, the column contains N, with the suffix + indicating that the procedure can really loop forever, while a suffix - indicates that the procedure always terminates but the algorithm cannot prove this. The sixth column uses the same notation to give the results achieved by Lindenstrauss and Sagiv's algorithm. The seventh column indicates the

Suite	Program	Query	Time	Us	LS	M/P
plu	mergesort_t	mergesort(in,out)	0.06s	N-	T	
plu	pl1_1	append(in,in,out)	0.03s	T	T	
plu	pl1_1	append(out,out,in)		T	T	
plu	pl1_2	perm(in,out)	0.08s	T	N-	
plu	pl2_3_1	p(in,out)	0.03s	N+	N+	
plu	pl3_5_6	p(out)	0.04s	N+	N+	
plu	pl3_5_6a	p(out)	0.05s	N+	T	MN/PT
plu	pl4_01	append3(in,in,in,out)	0.07s	T	T	
plu	pl4_01	append3(out,out,out,in)		T	N+	MT/PN
plu	pl4_4_3	merge(in,in,out)	0.04s	T	T	
plu	pl4_4_6a	perm(in,out)	0.06s	T	T	
plu	pl4_5_2	s(in,out)	0.03s	N+	N+	
plu	pl4_5_3a	p(in)	0.04s	N-	N+	MT/PN
plu	pl5_2_2	turing(in,in,in,out)	0.07s	N+	N+	
plu	pl6_1_1	qsort(in,out)	0.07s	T	T	
plu	pl7_2_9	mult(in,in,out)	0.03s	T	T	
plu	pl7_6_2a	reach(in,in,in)	0.03s	N+	N+	
plu	pl7_6_2b	reach(in,in,in,in)	0.18s	N-	N-	
plu	pl7_6_2c	reach(in,in,in,in)	0.03s	T	T	
plu	pl8_2_1	mergesort(in,out)	0.06s	N-	N-	
plu	pl8_2_1a	mergesort(in,out)	0.06s	T	T	
plu	pl8_3_1	minsort(in,out)	0.04s	N-	N-	
plu	pl8_3_1a	minsort(in,out)	0.06s	T	T	
plu	pl8_4_1	even(in)	0.03s	T	T	
plu	pl8_4_1	odd(in)		T	T	
plu	pl8_4_2	e(in,out)	0.18s	T	T	

Table 2: Results of analysing the test suites

queries for which the translation from Prolog to Mercury changes the actual termination properties of the program, MT/PN indicating that the Mercury version terminates while the Prolog one does not while MN/PT indicates the converse.

For three of the benchmark sets (the ones from [1, 7, 8], indicated by apt, bgh and dds respectively) our termination analysis gives answers which are identical to the ones given by the algorithms of Lindenstrauss and Sagiv. For the fourth benchmark set (the one from [16], indicated by plu) 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 machine we used for these test runs was a Sun SPARCserver 1000 with four 50 MHz SuperSPARC CPUs and adequate memory. For programs that use only one CPU (as our tests do), the performance of this machine is very similar to the performance of the SPARCstation 51 used by Lindenstrauss and Sagiv [13] for their benchmarking. This permits direct comparisons between their timing results and ours. The main difference is that while their algorithm is usually reasonably fast (taking 0.1 to 0.2 seconds), 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. It

	library	compiler
number of modules	45	131
total lines of code	24,337	106,084
total bytes of code	744,753	3,748,577
total number of procedures	1,584	5,426
total number of procedures inferred to be terminating	1,141	2,992
percentage of procedures inferred to be terminating	72.03%	55.14%
time taken by compilation of all modules	4m58s	32m44s
time taken by termination analysis of all modules	3.59s	36.69s
percentage of compilation time spent in termination analysis	1.20%	1.86%
average time taken by termination analysis of a module	0.08s	0.28s
maximum time taken by termination analysis of a module	0.42s	0.92s

Table 3: Results of analysing the Mercury compiler and standard library

took only 0.05 seconds on average; this is significantly less than the random variation between two compilations of the same source file. (The overall compilation times were in the range of 1 to 5 seconds.) In the worst case, our termination analysis algorithm took 0.30 seconds.

5.2 A large real program

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. In fact, the compiler alone is bigger than all but ten of the 500 programs listed on the Prolog 1000 list of real Prolog applications.

We ran our tests on one of our usual development machines, a Sun SPARCstation Ultra 1/170 with one 167 MHz UltraSPARC CPU and adequate memory. The results are summarized in table 3.

Our timing results show that our analyzer is very fast and has excellent worst-case behavior. 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.

As far as we know, neither the library nor the compiler contain procedures that can possibly loop forever. (If they do, it is a bug, and we have not seen such bugs in the version of the library and compiler we have tested in more than a year.) We can therefore assume that if termination 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 and more than half of the procedures in the compiler. There are several reasons for the difference. First, whereas most modules in the library are self-contained, the dependencies among the modules in the compiler are quite complex; about half the modules in the compiler belong to one SCC in the

dependency graph.[2] Second, the modules in the compiler depend on those in the library, while those in the library do not depend on those in the compiler. Third, 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 a very good result. For example, Lindenstrauss and Sagiv state [13] that the largest Prolog program they have processed successfully has 57 clauses; the compiler is two orders of magnitude larger than that. We believe there are good reasons why duplicating our level of accuracy for such large programs with a termination analyzer written for Prolog would be very difficult. The most important is that mode inference is a necessary first step in any termination analysis algorithm for an unmoded logic programming language. The accuracy of mode inference provides an upper bound on the accuracy of the termination analysis algorithm itself. Given the paucity of mode inference algorithms that have even been tested on large programs (10,000 lines or more), this presents a significant obstacle to the achievement of accurate termination analysis algorithms for such languages.

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 [13] 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, in particular by setting 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, as well as an instructive taxonomy, we refer the reader to the survey paper by De Schreye and Decorte [8]. The approach that we have used in our implementation has its roots in the seminal work by Ullman and Van Gelder [20], and in the development it spurred, including [3, 5, 6, 18]. The immediate inspiration was the work by Plümer [16], and Gröger and Plümer's handling of mutual recursion [10].

We have obtained greatly improved results by enhancing the basic method with two kinds of "usage analysis", a process somewhat similar to the demand analysis of Falaschi, Hicks and Winsborough [9]. In Stage 1 we track input variables whose sizes contribute to the sizes 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 [13]. Our results show that our analyzer has comparable accuracy and better performance in the average case. They also show that our analyzer is not subject to the dramatic slowdowns that sometimes affect Lindenstrauss and Sagiv's prototype analyzer.

[2] We discovered this fact in the course of this research. We intend to fix it shortly.

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.

We have some anecdotal evidence that a powerful termination analyzer can serve as a debugging tool for programs using complex data structures. If the programmer's intuition says that a procedure should always terminate, then the analyzer's inability 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. Our analyzer emits detailed explanations pinpointing the troublesome program point and the reason for its inability to prove termination.

In one case this allowed us to identify an error in a library module that is used heavily in the Mercury compiler. The compiler uses 2–3–4 trees for various kinds of "dictionaries". Although intuitively terminating universally, the predicate for insertion into 2–3–4 trees was flagged as possibly non-terminating. Closer inspection of the offending recursive call revealed that node splitting was handled incorrectly, so that the resulting trees were not necessarily kept balanced. Correcting this yielded provably terminating code and sped up the compiler by 35% (see the appendix for details).

Acknowledgements

We thank Fergus Henderson for our stimulating discussions, Naomi Lindenstrauss for responding so quickly to questions and for making benchmark programs available, and the Australian Research Council for their support.

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 version 2.0. Available from 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*, Lecture Notes in Computer Science 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*, Lecture Notes in Computer Science 582, pages 73–92. Springer-Verlag, 1992.

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

[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. Int. Logic Programming Symp.*, 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] Moreno Falaschi, Patrick Hicks, and William Winsborough. Demand transformation analysis for concurrent constraint programs. In M. Maher, editor, *Proc. Joint Int. Conf. Symp. Logic Programming*, pages 333—347. MIT Press, 1996.

[10] 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.

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

[12] David B. Kemp, Kotagiri Ramamohanarao, Isaac Balbin, and Krishnamurthy Meenakshi. Propagating constraints in recursive deductive databases. In *Proceedings of the First North American Conference on Logic Programming*, pages 981–998. MIT Press, 1989.

[13] 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, 1997.

[14] The Mercury group. Mercury 0.6. Available from http://www.cs.mu.oz.au/mercury.

[15] John K. Ousterhout. *Tcl and the tk Toolkit*. Addison-Wesley, 1994.

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

[17] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

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

[19] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[20] 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.

Appendix: Termination warnings as performance diagnostics

When we first tried out our termination analysis algorithm on the Mercury standard library, we had not yet implemented .trans_opt files. Therefore for many predicates, the analysis reported that it could not prove their termination because it depended on predicates in other modules. For a few predicates, however, it reported that it could not prove their termination even though it had access to all the relevant information.

Two of these predicates were in the module implementing 2–3–4 trees. These two predicates had almost identical code; they both inserted key/value pairs into 2–3–4 trees, the only difference being their action when the key was already present (one failed, the other just updated the associated value). The problem with both predicates was code such as

```
insert(three(T0, K0, T1, K1, T2), K, TreeOut) :-
   ( K < K0 ->
      ( T0 = four(T00, K00, T01, K01, T02, K02, T03) ->
         Left = two(T00, K00, T01),
         Right = two(T02, K02, T03),
         NewTop = four(Left, K01, Right, K0, T1, K1, T2),
         insert(NewTop, K, TreeOut)
      ;
         ...
      )
   ;
      ...
   ).
```

(For simplicity and clarity, we have omitted the values associated with keys and changed the order of the fields in the nodes; in the original source [14], the first few fields contain the keys and values while the remaining ones contain the subtrees.)

The three lines of this code fragment containing the unifications with `Left`, `Right` and `NewTop` carry out a split, a structural modification of the tree that leaves its logical contents unaltered. The purpose of the split is to ensure that when the traversal reaches a leaf node, that node will not be a four-node, since one cannot insert into a four-node without splitting it and thus destroying the balance of the tree (see [17]).

The fourth line of this code fragment contains the recursive call to `insert` into `NewTop`. The first argument of the recursive call is in fact bigger than the first argument of the head: a four-node and two two-nodes versus a three-node and a four-node (all the variables representing the various keys and other subtrees are present exactly once in both arguments). Therefore termination analysis is quite right in pointing out that this code poses a risk of nontermination.

To prove the termination of the predicate, one has to employ more sophisticated reasoning than our termination analysis is capable of. One has to argue that the recursive call will invoke a different clause of the insert predicate, that this clause will split the four-node into three two-nodes (with the higher level two-node having the two lower-level two-nodes as its children), and that this will lead to a recursive call to insert `K` into one of the lower-level two-nodes. Since the call whose first argument has a three-node and a four-node must indirectly lead to a recursive call whose first argument is a two-node, and since similar arguments can be made about all the other cases in which the recursive call may have input arguments at least as big as those in the head, the predicate does in fact terminate.

However, the recursive call does have two problems, one superficial and one very important. The superficial problem is that the recursive invocation of `insert` will start by comparing `K` to the middle key in the top-level four-node, this middle key being `K0`, even though we already know that `K` < `K0`. Therefore the second comparison is redundant. (Given that the `insert` predicate is polymorphic, and is able to deal with keys of any type, the comparison involves a procedure call; it is significantly more expensive than an integer comparison.)

The important problem is that when the top-level node of the input tree is a four-node, the `insert` predicate splits it into three two-nodes, thus increasing the height of the tree by one. This is supposed to happen only at the root of the whole tree. However, calling `insert` on `NewTop` causes this to happen for non-root nodes as well. Since the insertion into `NewTop` will cause it to grow in height while its siblings do not, the tree will become unbalanced. In fact, many of the 2–3–4 trees used in the Mercury compiler have sequentially allocated small integers as the keys. As a results, the 2–3–4 trees, instead of being balanced, often became sticks.

The fix is not to insert into `NewTop`, but to find out whether whether `K` is smaller or greater than `K01`, and to insert into `Left` or `Right` depending on the result. The new code looks like this:

```
( K < K01 ->
   Left = two(T00, K00, T01),
   Right = two(T02, K02, T03),
   insert(Left, K, NewLeft),
   TreeOut = four(NewLeft, K01, Right, K0, T1, K1, T2),
;
   ...
)
```

This fix has to be applied in several places in the predicate, which increases the size of the predicate. However, the new code avoids the redundant comparison, keeps the tree balanced, and since the first arguments of the recursive calls are now smaller than the first argument of the head, our analysis is now able to prove that the predicate always terminates.

The fix turned out to have a large impact on the speed of the Mercury compiler, which runs on average about 35% faster with the fix than without. In some cases, the benefit was even greater. In one experimental version of the compiler, the fix caused the compiler to terminate normally instead of running out of stack space. The reason is the heavy use made by the compiler of dictionary-type data structures, which are implemented as 2–3–4 trees.

We do not claim that when termination analysis points to a predicate as a potential trouble spot, fixing the predicate to make its termination apparent to the analysis will always have a similar impact. However, we think there is good reason to believe that a significant fraction of the predicates whose termination cannot be proven by our analysis do in fact have performance problems, actual non-termination being of course the worst possible performance problem. The reason is that a recursive call that does not decrease the size of the input arguments is a potential source of redundant computation even if it terminates. (This reason of course does not apply to predicates whose termination cannot be proven because of the absence of termination information about called predicates, either because of module boundaries or because of calls via higher-order constructs.)