Determinism analysis in the Mercury compiler

Fergus Henderson, Zoltan Somogyi and Thomas Conway
Department of Computer Science
University of Melbourne
Parkville, 3052 Victoria, Australia
{fjh,zs,conway}@cs.mu.OZ.AU

Abstract

Mercury is a new purely declarative logic programming language. The Mercury determinism system allows programmers to specify which predicates never fail and which predicates succeed at most once. This information allows the compiler to check the program for errors, pinpointing predicates that do not satisfy their declarations. This makes programmers significantly more productive and enhances the reliability of their programs. The Mercury compiler also uses determinism information to generate much faster code than any other logic programming system. This paper presents the algorithms used by the Mercury compiler to infer determinism information.

Keywords Programming language implementation, program analysis, logic programming, determinism, Mercury.

1 Introduction

One of the fundamental differences between logic programming and other programming paradigms is the presence of so-called "don't-know" nondeterminism, whereby a predicate may return more than one solution, and the associated backtracking mechanism used by the search engine to find answers to queries. Nondeterminism has both advantages and disadvantages. It provides a very declarative, high-level way of expressing certain problems and is one of the main reasons for the popularity of Prolog. On the other hand, nondeterminism can make the run-time behaviour of a program much more complex, making the program harder to debug and imposing significant efficiency overheads.

Ideally, of course, we would like to obtain the benefits of nondeterminism without the drawbacks. The approach we have taken in the design of the logic programming language Mercury is to ask programmers to give a determinism declaration for each predicate. This declaration specifies whether or not the predicate can succeed more than once and whether or not it can fail before producing its first solution. The Mercury compiler checks the definitions of the predicates in the program to make sure they are consistent with their declarations; if it cannot prove that they are consistent, it prints some error messages and rejects the program.

Proceedings of the Nineteenth Australasian Computer Science Conference, Melbourne, Australia, January 31–February 2 1996.

Sometimes the Mercury compiler rejects a program in which the declarations and definitions are consistent simply because it cannot prove that they are consistent. This is unavoidable, because proving that a predicate can have e.g. at most one solution is an undecidable problem [6]. It turns out that the simple algorithms employed by the Mercury compiler are sufficient to discover the true determinism of the vast majority of predicates in real programs. The few predicates falsely rejected can be easily modified to make them acceptable to the compiler. Making the determinism more apparent to the compiler makes it more apparent to human readers as well, and in our experience the new versions of such predicates are almost always more readable and maintainable.

The Mercury determinism system has several important benefits. First, by checking determinism declarations the compiler catches many bugs that would otherwise have required tedious manual debugging to track down. Second, the determinism declarations provide good documentation of the intentions of the original programmer, and since the compiler proves their correctness, they are more reliable than comments would be. Third, the compiler can exploit determinism information to generate specialized code for deterministic predicates, avoiding the overheads of the more complicated mechanisms required for nondeterministic predicates. The overheads of these mechanisms are therefore paid only by the predicates that need them.

The Mercury compiler is written Mercury itself. Several of the program fragments we use in this paper to motivate our algorithms are from the compiler.

The rest of this paper is organized as follows. Section 2 is a brief overview of Mercury: it describes Mercury's type, mode and module systems, and introduces Mercury's notion of determinism. Sections 3 and 4 describe the Mercury compiler's switch detection and common subexpression elimination algorithms. These two algorithms transform the internal form of the program to prepare it for the determinism analysis algorithm, which is the subject of section 5. Section 6 considers of the impact of determinism information on programmer productivity.

We assume that the reader is familiar with Prolog syntax.

2 The Mercury language

Syntactically, Mercury is similar to Prolog with additional declarations. Semantically, however, it is very different. Mercury is a pure logic programming language with a well-defined declarative semantics. Like Gödel [3], Mercury provides declarative replacements for Prolog's non-logical features. Unlike Gödel, Mercury provides replacements for *all* such features, including I/O.

The rest of this section is necessarily brief, and omits many details of the language. For information on those details, see our World Wide Web page at `http://www.cs.mu.oz.au/~zs/mercury.html`.

2.1 Types

Mercury's type system is based on a polymorphic many-sorted logic. It is essentially equivalent to the Mycroft-O'Keefe type system [4], and to the type system of Gödel [3]. We borrow our syntax from the NU-Prolog type checkers.

The basic method of defining types is with declarations such as

```
:- type determinism ---> det ; semidet ;
                         nondet ; multi ;
                         erroneous ; failure.
:- type can_fail    ---> yes ; no.
:- type max_soln    ---> zero ; one ; many.
```

A type declaration introduces a new type and lists the one or more function symbols (in the above cases six, two and three respectively) that can be used to construct terms of that type. These types are effectively enumerations, i.e. none of the function symbols has any arguments. It is of course possible to declare function symbols with arguments, as in

```
:- type tree(T) ---> empty ;
                     node(T, tree(T), tree(T)).
```

The arguments of the function symbols in the declaration are types, giving the types of the arguments of the function symbols. These types can be defined anywhere in the program; the language allows forward references as well as (mutually) recursive types. Mercury supports parametric polymorphism. Type declarations may contain references to type variables, e.g. T is a type variable in tree(T).

In principle all types can be defined just like this, but in practice types such as `int` must be built into the system.

Since we need a strong type system as a foundation for our strong mode system, the compiler must be able to determine the type of every variable. The language requires that the programmer declare types of the arguments of every predicate in the program, like this:

```
:- pred detism_parts(determinism,
   can_fail, max_soln).
:- pred append(list(T), list(T), list(T)).
```

The compiler then automatically infers a unique most general assignment of parametric polymorphic types to all the variables and function symbols in the program; it rejects the program if there is no such assignment.

2.2 Modes

Mercury requires programmers to declare the modes of their predicates. In their simplest form, mode declarations specify which arguments to a predicate are input (ground on entry to the predicate), and which arguments are output (free on entry to the predicate and ground on exit). A predicate may have more than one mode:

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

We call each mode of a predicate a *procedure*.

Of course, one of the differences between logic programming languages and traditional imperative languages is that in logic programming, variables can be partially instantiated and procedure arguments can be partially input and partially output, and so Mercury's mode system allows the programmer to express much more complex dataflow patterns than just '`in`' and '`out`'. However, space limitations prevent us from going into more detail here.

2.3 Determinism

For each mode of a predicate, the programmer must categorise that procedure according to the maximum number of solutions it can produce (zero, one, or more than one) and whether or not it can fail before producing its first solution. There are six possible combinations, as shown by the definition of the predicate `detism_parts`:

```
:- pred detism_parts(determinism,
   can_fail, max_soln).
:- mode detism_parts(in, out, out) is det.
:- mode detism_parts(out, in, in) is det.

detism_parts(det,       no,  one).
detism_parts(semidet,   yes, one).
detism_parts(multi,     no,  many).
detism_parts(nondet,    yes, many).
detism_parts(erroneous, no,  zero).
detism_parts(failure,   yes, zero).
```

If a procedure has exactly one solution, then it is deterministic and its mode declaration should specify the determinism `det`. If it has at most one solution, but may not have any then it is semideterministic and should be declared `semidet`. If it has one or more solutions then it should be declared `multi`. If it may have multiple solutions or none, then it is nondeterministic and should be declared `nondet`.

The four determinisms det, semidet, multi and nondet suffice for the vast majority of procedures but there are a very small number of procedures which require another determinism because they never produce a solution. The determinism `failure` is used for procedures which always fail. Since computations that always fail cannot export any bindings, there is no practical reason for writing such predicates, but the determinism is required to describe the builtin goal `fail`, which is occasionally useful. The determinism `erroneous` is used for procedures that from the program's point of view neither succeed nor fail, i.e. they loop forever or abort execution. Programs can abort execution by calling the builtin predicate `error`. Programs call this predicate when they detect an internal inconsistency, for example when the precondition of a predicate is not satisfied, which would correspond to an assertion failure in imperative languages. Mode analysis does not require conjunctions containing calls to `error` (or to other predicates with a determinism of erroneous or failure) to produce all the variables they would otherwise be required to produce; since control will never reach the end of the conjunction, this is acceptable. The rest of the compiler mostly treats failure procedures as if they were semidet and erroneous procedure as if they were det; it also treats multi procedures the same way it treats nondet procedures most of the time.

Determinism declarations are attached to mode declarations, as shown by the above declaration of `detism_parts`. Both modes of that predicate are deterministic, but different modes of a predicate may have different determinisms. For example, concatenating two lists can only be done one way, but an existing list can be split up into two components in several ways:

```
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
```

The compiler analyses the bodies of procedures to check that their determinism declarations are correct. The algorithm that the compiler uses for this is the subject of this paper.

2.4 Input and output

Since Mercury is a declarative language, it must provide a declarative way to I/O. Our approach is to view predicates that perform I/O as representing relations between an initial state and a final state of the world. Since the implementations of these predicates must update the state of the world destructively, we must be sure that the old state of the world will never be referred to again. This requires that the old state of the world passed to an I/O predicate be a unique reference, and since the new state returned by the I/O predicate will probably be passed to another I/O predicate later on, such predicates must guarantee that the return a unique reference to the new state of the world. To describe arguments representing the state of the world (and other data structures that must be updated destructively), Mercury allows programmers to declare modes that combine dataflow information with uniqueness information. The two usual such modes are `destructive_input` and `unique_output` (`di`/`uo` for short); these are used to describe the state of the world arguments of all I/O predicates. For example, here is a contrived version of the hello-world program in Mercury:

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

main(State0, State) :-
    write_string("hello ", State0, State1),
    write_string("world\n", State1, State).
```

Usually we use DCG notation as a way of hiding the extra state of the world arguments in predicates that do more than trivial amounts of I/O.

The Mercury compiler must reject the program if it finds that a predicate that requires one its arguments to be unique is called with a nonunique reference to a data structure. Determinism analysis plays a significant part of this check. The state of the world must not only be unique during forward execution, we must also be sure that backtracking cannot cause an old state of the world to be reused (for obvious reasons, most components of the external world — such as printers — cannot be made to backtrack). For this reason all the I/O predicates must be deterministic, and they can only be called from a deterministic context (not e.g. from the condition of an if-then-else).

2.5 Modules

The Mercury module system is simple and straightforward. Each module must start with a `module` declaration, specifying the name of the module. An `interface` declaration specifies the start of the module's interface section: this section contains declarations for the types, instantiation states, modes, and predicates exported by this module. Mercury provides support for abstract data types, since the definition of a type may be kept hidden, with only the type name being exported. An `implementation` declaration specifies the start of the module's implementation section. Any entities declared in this section are local to the module and cannot be used by other modules. The implementation section must of course contain definitions for all abstract data types and predicates exported by the module, as well for all local types and predicates. If a module wishes to make use of entities exported by other modules, then it must explicitly import those modules using one or more `import_module` declarations.

3 Switch detection

Consider the predicate `partition` in figure 1. The definition of this predicate has two disjunctions. The first disjunction is implicit in the two clauses of the definition, the second is explicitly written by the programmer. Nevertheless, `partition` is deterministic. Not only are all the procedures called in the definition declared deterministic with the exception of the semidet procedure called in the condition of the if-then-else, but also both disjunctions have the property that at most one disjunct can succeed for any given combination of input values. The reason is that for each disjunction, there is a variable that is (a) already bound on entry to the disjunction and (b) is unified with different function symbols in different arms of the disjunction. For the outer disjunction, this is the first argument, with the two function symbols being nil/0 and cons/2; for the inner disjunction, this is MaybeFunctor, with the two function symbols being yes/1 and no/0. We call such disjunctions *switches*, since they superficially resemble switches in the C programming language.

Finding out which disjunctions are switches is an essential component of the Mercury determinism system. Having two kinds of disjunctions, implicit and explicit, would complicate this algorithm (and several others), so the compiler makes all disjunctions explicit as soon as it has finished reading in the program. This transformation also introduces new variables to represent the various arguments of the predicate, and unifications between these variables and the arguments of clause heads. The transformed version of the `detism_parts` predicate from the previous section is:

```
detism_parts(Headvar1, Headvar2, Headvar3) :-
   ( Headvar1 = det,
     Headvar2 = no,  Headvar3 = one
   ; Headvar1 = semidet,
     Headvar2 = yes, Headvar3 = one
   ; Headvar1 = multi,
     Headvar2 = no,  Headvar3 = many
   ; Headvar1 = nondet,
     Headvar2 = yes, Headvar3 = many
   ; Headvar1 = erroneous,
     Headvar2 = no,  Headvar3 = zero
   ; Headvar1 = failure,
     Headvar2 = yes, Headvar3 = zero
   ).
```

For each variable that is input to the disjunction, our switch detection algorithm tests whether the disjunction is a switch on that variable. The predicate `partition` (figure 1) is a simplified version of the predicate in the compiler that implements most of this test. This predicate takes as inputs a list of goals (the disjuncts) and the variable currently being considered. Its outputs are a table mapping function symbols to a list of goals (the disjuncts that test the variable against that function symbol) and a list of goals left out of the table (the disjuncts that do not test the variable against any function symbol). For an empty list of disjuncts, the predicate returns an empty table and an empty list. For a non-empty list of disjuncts, the predicate invokes itself recursively on the tail of the list, and then considers how to classify the disjunct at the head of the list.

```
:- pred partition(list(goal), var,
   map(functor, list(goal), list(goal)).
:- mode partition(in, in, out, out) is det.

partition([], _Var, Cases, []) :-
   map_init(Cases).
partition([Goal0 | Goals], Var, Cases, Left) :-
   partition(Goals, Var, Cases1, Left1),
   goal_to_conj_list(Goal0, ConjList0),
   map_init(Substitution),
   find_switch_test(ConjList0, Substitution,
      Var, ConjList, MaybeFunctor),
   (
      MaybeFunctor = yes(Functor),
      conj_list_to_goal(ConjList, Goal),
      ( map_search(Cases1, Functor, Disjs1) ->
         Disjs = [Goal | Disjs1]
      ;
         Disjs = [Goal]
      ),
      map_set(Cases1, Functor, Disjs, Cases),
      Left = Left1,
   ;
      MaybeFunctor = no,
      Cases = Cases1,
      Left = [Goal0 | Left1]
   ).
```

Figure 1: The predicate `partition`

The first step in this process is converting the disjunct to a conjunction of goals. Almost always, the disjunct is a conjunction already, and `goal_to_conj_list` merely returns the conjuncts. If the disjuct is some other type of goal, `goal_to_conj_list` returns a list with that goal as its only element. The predicate `find_switch_test` processes the unifications among the conjuncts, building up a current substitution on the way (`map__init` creates an empty substitution). At each unification `find_switch_test` checks whether Var has become bound to a known term. If it has, it wraps yes/1 around the function symbol of that term and returns the result as MaybeFunctor. If it hasn't, it processes the rest of the conjunction; if no more conjuncts remain, it returns no as MaybeFunctor.

If the current disjunct does not test the variable, the algorithm simply puts it in the list of such disjuncts. If the current disjunct does test the variable, it finds out what other disjuncts (if any) test the variable against the same function symbol, puts this disjunct in the list, and modifies the table to say that this function symbol is mapped to this new list of disjuncts.

The main switch detection algorithm traverses the internal form of each procedure definition. Whenever it finds a disjunction, it looks up the list of variables used within the disjunction and finds out which ones are bound on entry to the disjunction. For each such variable, the algorithm calls `partition`. If `partition` discovers that the variable is tested against at least two different function symbols in the various disjuncts, then one can use this variable to create a switch. If all disjuncts test the value of the variable (none are left out), the algorithm converts the disjunction into a switch immediately. The switch records the name of the variable being switched on, whether the switch covers all the function symbols the variable can be bound to, and the list of cases, where each case pairs a function symbol with the goal representing that arm of the switch. If Cases maps that function symbol to just one disjunct, then the corresponding arm of the switch will be that disjunct; if Cases maps that function symbol to more than one disjunct, the corresponding arm of the switch will be a disjunction containing those disjuncts.

If `partition` finds that there are some disjuncts that do not test the variable, then the value of the variable discriminates only among *some* disjuncts. Since we prefer switches that discriminate among all disjuncts, our algorithm delays acting on such *partial* switches until it has tested all the input variables of the disjunction and found no full switches. It then creates a disjunction with two arms, one being the partial switch and the other being a disjunction of all the original disjuncts left out of the partial switch. Note that if the algorithm has recorded more than one partial switch, it must choose one using some criterion such as maximizing the number of function symbols that variable is tested against or minimizing the number of disjuncts left out of the switch. Since the final goal must contain a disjunction, the choice cannot affect the determinism of the computation, only its speed. However, in the rare situations when the algorithm must choose between two or more partial switches, each choice tends to yield code of equivalent speed, so the nature of the heuristic used does not matter much in practice.

Some of the disjuncts of the original disjunctions may have other disjunctions nested within them. Therefore before it returns, the switch detection algorithm will call itself recursively on the arms of the construct (disjunction or switch) it ended up with.

When this algorithm processes the predicate `detism_parts` in its forward mode, the only variable whose value is known on entry to the disjunction is Headvar1. The call to `partition` discovers that all disjuncts test Headvar1, so the algorithm turns the disjunction into a six-way switch on Headvar1. Each arm of the switch consists of one of the original disjuncts, with one difference: the test against Headvar1 is now marked to say that it always succeeds. (This is why `find_switch_test` returns a new conjunction list.) This reflects the fact that the switch tests the value of Headvar1, so each arm of the switch is entered only if Headvar1 has the appropriate value.

When our algorithm processes `detism_parts` in its backward mode, the values of Headvar2 and Headvar3 are both known on entry to the disjunction. If Headvar2 appears first, the algorithm will find that the disjunction is a two-way switch on Headvar2, with each arm of the switch containing a disjunction with three of the original disjuncts. It then performs switch detection on these disjunctions, and finds that both disjunctions are in fact three-way switches on Headvar3. Had Headvar3 appeared first, the resulting goal would have the switches nested the other way (two-way switches on Headvar2 nested inside a three-way switch on Headvar3). The two alternatives have the same determinism and the same speed.

4 Common subexpressions

The algorithm we have presented in section 3 is sufficient to find all switches in the majority of Mercury programs. However, there are some cases it does not handle as well as we would like. Consider this predicate definition:

```
count_elements([], zero).
count_elements([_], one).
count_elements([_,_|_], many).
```

The internal form of this predicate not only converts the clauses into a disjunction but also breaks down unifications into their simplest forms (X = Y or X = f(Y_1, ..., Y_n)):

```
count_elements(Headvar1, Headvar2) :-
   ( Headvar1 = [],
     Headvar2 = zero
   ; Headvar1 = [H1 | T1], T1 = [],
     Headvar2 = one
   ; Headvar1 = [H2 | T2], T2 = [H3 | T3],
     Headvar2 = many
   ).
```

The switch detection algorithm turns this disjunction into a two-way on Headvar1, with the "cons" arm of the switch being a two-way disjunction. The reason why the algorithm cannot turn this disjunction into a switch is that the entity being switched on, the tail of Headvar1, is not an input to the disjunction. The fix is to make it an input to the disjunction. This requires noticing that both disjuncts have an unification to extract the head and tail of Headvar1's cons cell, and to move this unification outside the disjunction.

The algorithm we use for this common subexpression elimination is very similar to the switch detection algorithm in several respects. It also traverses goals looking for disjunctions, it also loops through the various nonlocal variables of a disjunction, it also converts the body of each disjunct into a lists of conjuncts, and it also looks at the unifications among the conjuncts. The difference is that once it has found that the first disjunct tests the variable under consideration against a given function symbol, it insists on all the other disjuncts testing it against the same function symbol. If some disjuncts do not have such a test, or if the first disjunct contains no tests at all of the variable, it returns the original goal unchanged. (We don't pull a unification out of a disjunction if it appears in only some of the disjuncts. Doing so is not necessary for accurate determinism analysis, and in some cases such action would increase the number of stack slots required by the procedure.)

When the algorithm does find that all the disjuncts match a variable against the same function symbol, it creates a new unification between the variable and the function symbol with fresh variables as its arguments, and puts this unification just before the disjunction. It then replaces the equivalent unifications inside all the disjuncts with code that for each argument of the function symbol, unifies the fresh variable representing that argument with the variable representing that argument in the unification that was in the disjunct. The result is to turn a disjunction like this:

```
( Headvar1 = [H1 | T1], T1 = [],
  Headvar2 = one
; Headvar1 = [H2 | T2], T2 = [H3 | T3],
  Headvar2 = many
)
```

into a goal like this:

```
Headvar1 = [H0 | T0],
( H0 = H1, T0 = T1, T1 = [],
  Headvar2 = one
; H0 = H2, T0 = T2, T2 = [H3 | T3],
  Headvar2 = many
)
```

If the common subexpression elimination algorithm makes any changes in the internal form of a procedure in the process of examining all its disjunctions, it reruns several phases of the compiler on the procedure. First, it recalculates the scopes of variables, since these may have changed. Second, since the order of execution has changed as well, a variable may now be produced by a different goal than before, so it reruns mode analysis to recompute the mode annotations on goals. Third, it then reruns switch detection in the hope that it can now find switches it couldn't find before. Switch detection will indeed find the disjunction above to be a switch on T0. (After the goal `T0 = T1`, the current substitution records that T0 and T1 are effectively the same variable, so a test of T1 against the empty list is effectively a test of T0 as well. Similarly, after the goal `T0 = T2`, a test of T2 against a cons cell is a test of T0 as well.) And fourth, switch detection may have broken down larger disjunctions into several smaller ones and hence exposed further opportunities for common subexpression elimination, and therefore the common subexpression elimination algorithm is rerun as well. It is possible for this algorithm to be invoked several times, but the compiler cannot go into an infinite loop, because each iteration reduces the number of function symbols in the code.

The reason why we introduce unifications such as `T0 = T1` instead of just substituting T0 for T1 in the relevant disjunct is that substituting a compiler-invented variable for a variable named by the programmer would make make any error messages generated by determinism analysis much less meaningful to the programmer. A later phase of the compiler does perform this substitution.

5 Determinism analysis

The determinism analysis phase of the Mercury compiler runs after the type analysis, mode analysis, switch detection and common subexpression elimination phases. It has access to the definitions of the predicates of the module being compiled as modified and annotated by the earlier phases, and to the declarations of all the predicates that can be called from the module, whether they are defined in the module or imported from another module. Its job is to infer the determinism of procedures that do not have a declared determinism, to check for violations of determinism declarations, and to annotate every goal and subgoal in the program with its determinism for use by the later phases of the compiler (various optimizations and the code generator).

The determinism analysis phase of the compiler uses two main algorithms, which are the subjects of the following two subsections.

5.1 Analysing a goal

The first main algorithm of the determinism analysis phase is the algorithm that infers the determinism of a goal. The predicate `infer_goal`, which implements this algorithm, is essentially a case analysis of the various types of goals, of which there are eight: calls, unifications, conjunctions, disjunctions, switches, if-then-elses, negations, and existential quantifications. (Universal quantifications are transformed into a combination of negation and existential quantification soon after parsing.)

Inferring the determinism of a call is trivial if the called mode of the called predicate has a determinism declaration. If it doesn't, we initially infer a determinism of `erroneous`. If this assumption later turns out to be incorrect, the compiler repeats parts of the inference process; we will discuss this in section 5.2.

The mode analysis phase of the compiler classifies unifications into five types. Unifications of the form X = Y with mode (in,out) or (out,in) are *assignments*, and are always deterministic. Unifications of the form $X = f(Y_1, ..., Y_n)$, $n >= 0$, where X is output and the Y_i are distinct variables are *constructions*, and are also deterministic.

Unifications of the form X = f(Y_1, ..., Y_n), $n >= 0$, where X is input and the Y_i are distinct variables are *deconstructions*. Such unifications are usually semidet, but in two cases the compiler knows that they cannot fail and therefore infers them to be deterministic. The two cases occur when f is the only functor to which a variable of X's type can be bound to, and when control reaches the deconstruction at runtime only after X's functor has been tested and found to be f, which can happen e.g. if X is the control variable of a switch.

Unifications of the form X = Y with mode (in,in) where the shared type of X and Y is atomic are *simple tests*. These are always semidet, unless X and Y are already known to be bound to the same function symbol. Unifications of the form X = Y which are not simple tests are implemented as calls to automatically generated type-specific unification procedures, such as this one for unifying two lists of integers:

```
unify_list_int(V_1, V_2) :-
   V_1 = [],
   V_2 = [].
unify_list_int(V_1, V_2) :-
   V_1 = [V_3 | V_4],
   V_2 = [V_5 | V_6],
   unify_int(V_3, V_5),
   unify_list_int(V_4, V_6).
```

The mode of the call depends on the initial instantiation states of X and Y. The determinism inferred for the unification is the determinism of the called mode of the unification procedure. Usually this will be semidet, but it can be det if X and Y are known to have the same shape and complementary instantiations, such as f(a, B) and f(A, b).

The following is a slightly simplified version of the predicate that the compiler uses to infer the determinism of a conjunction of goals:

```
:- pred infer_conj(list(hlds_goal), instmap,
   misc_info, list(hlds_goal), determinism).
:- mode infer_conj(in, in, in, out, out) is det.

infer_conj([], _InstMap0, _MiscInfo, [], det).
infer_conj([Goal0 | Goals0], InstMap0,
      MiscInfo, [Goal | Goals], Detism) :-
   infer_goal(Goal0, InstMap0, MiscInfo,
      Goal, InstMap1, Detism1),
   infer_conj(Goals0, InstMap1, MiscInfo,
      Goals, Detism2),
   detism_parts(Detism1, CanFail1, MaxSoln1),
   detism_parts(Detism2, CanFail2, MaxSoln2),
   conj_canfail(CanFail1, CanFail2, CanFail),
   conj_maxsoln(MaxSoln1, MaxSoln2, MaxSoln),
   detism_parts(Detism, CanFail, MaxSoln).
```

The predicate takes as inputs the list of goals in the conjunction, and two variables (InstMap0 and MiscInfo) that together describe the state of instantiation of the variables in the conjunction (the instantiation information is needed to find

Left	Right	Conj	Disj	Switch
yes	yes	yes	yes	yes
yes	no	yes	no	yes
no	yes	yes	no	yes
no	no	no	no	no

Table 1: Can_fail combinations

Left	Right	Conj	Disj	Switch
zero	zero	zero	zero	zero
zero	one	zero	one	one
zero	many	zero	many	many
one	zero	zero	one	one
one	one	one	many	one
one	many	many	many	many
many	zero	zero	many	many
many	one	many	many	many
many	many	many	many	many

Table 2: Max_soln combinations

goals that generate no outputs; see the end of this section). The predicate has two outputs: new versions of the goals in the initial list, now annotated with their determinisms, and the determinism inferred for the conjunction.

The first clause states that the empty conjunction (which is equivalent to the goal `true`) is deterministic. The recursive clause finds the determinism of the first goal of the conjunction on the one hand and the determinism of the remaining part of the conjunction on the other hand. It then breaks down these two determinisms into their components, combines the two canfail components and the two maxsoln components, and builds up a determinism from the resulting canfail and maxsoln components. The third column of table 1 gives the definition of `conj_canfail`, while the third column of table 2 gives the definition of `conj_maxsoln`. Note that the operations implemented by these predicates are symmetric and associative, so the order of the goals in the conjunction does not matter, and that the components of det (no and one) are the identities of the respective operations (which form two *monoids*, algebraic structures with associativity and identity). Essentially a conjunction can fail if any one of its goals can fail. It has at most zero solutions if any one of its goals has at most zero solutions. If all goals can have solutions, then the conjunction may have many solutions if any goal can have many solutions, and the conjunction is deterministic only if all the goals in it are deterministic.

The algorithm for inferring the determinism of a disjunction is very similar to the algorithm for inferring the determinism of a conjunction, the only difference being the use of different tables for combining determinism components; these are shown in the fourth columns of tables 1 and 2. The identities of these two operations are yes and zero, so the determinism of an empty disjunction (which is equivalent to the goal `fail`) is failure. Basically a disjunction can fail only if all its goals can fail. It can succeed many times if any of its goals can succeed many times or if at least two of its goals can succeed once. It is deterministic only if one of its goals is deterministic and all the others succeed at most zero times. Programmers tend not to put goals without solutions into disjunctions, so disjunctions are virtually never deterministic.

The algorithm for inferring the determinism of a switch is very similar to the algorithm for inferring the determinism of disjunctions, but it uses the tables shown in the fifth columns of tables 1 and 2. The identities of these two operations are no and zero, so the determinism of an empty list of cases (a construct that exists only in the compiler, not in programs) is erroneous. The differences between the tables for disjunctions and switches arise because at most one arm of a switch can produce solutions. Therefore if all arms are deterministic the switch is deterministic as well, and if any arm can fail the switch can fail as well. Regardless of the determinism of the goals forming the arms of the switch, the switch can fail without producing solutions if the switch does not cover all the function symbols that the variable being switched on can be bound to; this is discovered during switch detection.

The Mercury code for inferring the determinism of an if-then-else calls `infer_goal` on the condition, the then-part and the else-part, breaks down the returned determinisms into their components, and then combines these components using this code:

```
conj_maxsoln(CondMS, ThenMS, AllThenMS),
switch_maxsoln(AllThenMS, ElseMS, MS),
switch_canfail(ThenCF, ElseCF, CF),
detism_parts(Detism, CF, MS).
```

The if-then-else is basically a switch between the "then" part and the "else" part. However, if the condition succeeds several times, then the "then" part may succeed several times as well even if it is deterministic, since it will produce one solution for each set of inputs given to it by the condition. This is why the call to `conj_maxsoln` is needed.

Almost all negated goals are semidet, in which case the negation itself is semidet. In rare cases the negated goal may be deterministic, in which case the negation itself has a determinism of failure, or the negated goal may be failure, in which case the negation itself is deterministic. If the negated goal has a determinism of erroneous, meaning the goal will cause a runtime abort, the negation itself also has a determinism of erroneous. To preserve soundness, a negation can never produce any output. Therefore negated goals can never succeed more than once. If a negated computation does generate values for some variables, the negated goal will need to quantify those variables within the scope of the of the negation.

The determinism of a goal that is an existential quantification is the same as the determinism of the goal being quantified over, except possibly in cases where the quantification as a whole has no output, in which case the quantification goal may not have more than one solution. Consider the goal `some [Item] (member(Item, List), test(Item))` where the value of List is already known. The `member` predicate will succeed as many times as there are elements in List, so the conjunction being quantified over may have many solutions. However, the quantification as a whole has no output. Therefore the surrounding goal only cares about whether any solutions exist; all solutions to the quantified goal are equivalent from its point of view. Therefore there is no point in finding more than one solution to the conjunction, and the code generator will emit code to stop backtracking into `member` after `test` succeeds for the first time. (This code is effectively a commit, but it is invisible from the user level and it does not affect the semantics of the program.)

Explicit quantification is rare in Mercury since the language specifies that variables are implicitly existentially quantified over their smallest enclosing scope, and this is usually what programmers want. Therefore after it has performed its case analysis to find the natural determinism of the goal under consideration, `infer_goal` checks whether the goal has any outputs. If the goal has no outputs yet its inferred determinism indicates that it can succeed more than once, `infer_goal` wraps the implied existential quantification around the goal.

5.2 Analysing a module

(Reminder: we call each mode of a predicate a *procedure*. Most predicates have one mode, but a few, e.g. `append` and `detism_parts`, have more.)

Mercury requires determinism declarations for every mode of every predicate that is exported from its module. However, predicates that are local to a module need not have determinism declarations. Since these declarations are useful documentation, the compiler issues a warning if a determinism declaration is omitted, but this warning can be switched off. The determinism analysis pass of the compiler therefore works in two stages. The first stage finds all the procedures in the module that do not have determinism declarations, and infers a determinism for them. The second stage looks at all the procedures in the module that do have determinism declarations, and checks the correctness of those declarations.

The first stage assigns erroneous as the initial inferred determinism of the procedures it works with. It then enters a loop, each iteration of which invokes the algorithm of the previous section to infer the determinism of each of these procedures. The loop stops when for every procedure, the new inferred determinism is the same as the old one. Otherwise the loop continues with the new inferred determinisms (but see below).

```
det_analysis.m:508:  In `infer_conj(in, in, in, out, out)':
det_analysis.m:508:  Error:  determinism declaration not satisfied.
det_analysis.m:508:  Declared `det', inferred `semidet'.
det_analysis.m:510:  in argument 1 of clause head:
det_analysis.m:510:  unification of `HeadVar1' and `[Goal0 | Goals0]' can fail.

det_analysis.m:580:  in argument 1 of clause head:
det_analysis.m:580:  in argument 5 of functor `deconstruct/5':
det_analysis.m:580:  unification with `yes' can fail.

det_analysis.m:570:  The switch on HeadVar1 does not cover construct/5.
```

Figure 2: Examples of determinism error messages

If the new inferred determinism for `p` is semidet, and that of `q` is det, the inferred determinism of `q` may change to semidet on the next iteration if `q` calls `p`. This is why we recompute the inferred determinism of all undeclared procedures if any one of them changes. It would be sufficient to recompute the inferred determinisms of the procedures that call `p` directly or indirectly, but in our experience very few procedures lack determinism declarations (at least partially due to the warnings), and most of those are deterministic anyway, so the cost of computing the call tree would usually be greater than the cost of recomputing all the inferred determinisms.

A predicate with the definition `p :- not(p)` could cause the above algorithm to go into an infinite loop if the initial assumed determinism for `p` were det. The first iteration would modify this to failure, and the second iteration would change it back to det. It is possible that more complicated definitions involving recursion through negation might go into similar loops even with erroneous as their initial assumed determinism. To avoid such loops, our algorithm combines the new inferred determinism of a procedure with the old one before recording it for use by later iterations. This combination operation is designed to make monotonic the sequence of determinisms recorded for a given procedure. The can_fail component of the recorded determinism can change from no to yes but not back; the max_soln component can change from zero to one to many, but not back. Since the total number of possible combinations of determinisms of the procedures under consideration is bounded, this monotonicity guarantees that our algorithm terminates. The determinisms it infers are also guaranteed to be safe approximations of the intended determinisms.

The second stage of the determinism analysis pass needs no iteration. It simply visits each procedure with a determinism declaration, infers its determinism, and compares the inferred and declared determinisms. If they are the same, no action is necessary. If the inferred determinism is tighter than the declared one (e.g. declared multi but inferred det), the compiler emits a warning. If the inferred determinism is not as tight as the declared one (e.g. declared det but inferred semidet), the compiler prints an error message, prints its diagnosis of the cause(s) of the problem, and stops the compiler from proceeding to code generation.

6 Impact on productivity

One of the most frustrating experiences of a Prolog programmer occurs when a large computation that was intended to succeed fails instead. The bug could be *anywhere* in the computation, and the language offers no help in tracking it down except the usual facilities for tracing the execution of the program. One cannot even just watch out for the failure of goals, because even a computation intended to be deterministic will have failed goals in the conditions of if-then-elses. When tracing through the program, the programmer must know which goal failures are expected and which represent bugs. In Prolog, this is frequently difficult for anyone but the original author of the predicate concerned.

The Mercury determinism systems detects such problems at compile time, and in our implementation of the Mercury compiler we expended considerable effort to make the resulting error messages as useful to the programmer as possible (see figure 2 for some examples). This support significantly reduces the time programmers spend debugging, and since debugging times are notoriously difficult to predict, makes projects easier to schedule. The cost is the up-front investment of some time writing down declarations. Since programmers should certainly know the intended behavior of the predicates they are writing, this cost is quite small, and it is amply justified by the resulting productivity and reliability benefits.

But perhaps the biggest benefit of the determinism system is that it makes programs significantly easier to maintain. The Mercury compiler is 82,000 lines of Mercury code, and by now ten people have contributed to it. If it weren't for the determinism system, we would be very reluctant to add alternatives to any widely used types, because such changes require that all predicates that process items of the affected type be located, inspected and fixed. Considering that such changes often break code that the person making the change didn't even know existed, this task is very tedious and error prone. The determinism system automates this task very well. When we make such changes, we normally change just the type definition before invoking make to recompile the affected source files. This recompilation generates error messages pinpointing each location affected by the change; we give these messages to the utility "error" which makes it significantly easier to visit each location to be fixed. After we have adapted a programming style to exploit this capability, we have found this approach to be so reliable in locating all the predicates that must be fixed that we have no aversion at all to making such changes on a regular basis.

7 Conclusion

The concept of determinism in logic programs has been around for a long time. Several researchers [1, 2, 5] have proposed global analysis algorithms for finding out which predicates in a Prolog program can succeed at most once with the intention of optimizing the compilation of such predicates. Smolka [7, 8] has proposed a system that requires determinism declarations somewhat similar to ours but in which these declarations are not checked by the compiler. Neither of these approaches is as effective as the Mercury determinism system in improving programmer productivity and program reliability. While writing about 100,000 lines of Mercury code (including the Mercury compiler and the Mercury standard library) we have found the determinism system to be invaluable in pinpointing errors. We have found that once a new module gets past the compiler, it often works correctly the first time.

The Mercury determinism system also contributes to the efficiency of the Mercury system. First, procedures that cannot succeed more than once (which is most procedures in most Mercury programs) can manage their stack frames using the techniques used by imperative languages. Second, switches make indexing very fast, sometimes even faster than in C, since (due to the Mercury type system) there is often no need to check whether the switched-on variable is in range. Benchmarks show Mercury to be almost twice as fast as Aquarius Prolog, about five times as fast as SICStus Prolog, and about ten times as fast as Quintus Prolog [9].

In some ways, the most important contribution of the Mercury determinism system is that it provides a necessary basis for safe declarative I/O. Other logic programming languages (e.g. NU-Prolog and Gödel) have provided replacements for most of Prolog's nonlogical constructs, but Mercury is the first *pure* logic programming language (apart from toy languages without any I/O). Making the language purely declarative has several enormous benefits. It allows optimizers to automatically transform programs into more efficient forms, it allows compilers to emit parallel code without any intervention by the programmer, and it makes it possible to partially automate the process of debugging via the technique known as declarative debugging.

Mercury has been available under the General Public License since July 1995. Further information on the project (including previous papers and directions for obtaining the latest version) are available through the project's home page on the World Wide Web, at URL `http://www.cs.mu.oz.au/~zs/mercury.html`.

We would like to thank the Australian Research Council, the Key Centre for Knowledge Based Systems, and the Centre for Intelligent Decision Systems for their support.

References

[1] S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan and R.C. Sekar. Extracting determinacy in logic programs. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 424–438, Budapest, Hungary, June 1993.

[2] Saumya K. Debray and David S. Warren. Detection and optimisation of functional computations in Prolog. In *Proceedings of the Third International Conference on Logic Programming*, pages 490–504, London, England, June 1986.

[3] Pat M. Hill and John W. Lloyd. *The Gödel programming language*. MIT Press, 1994.

[4] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, Volume 23, pages 295–307, 1984.

[5] Katsuhiko Nakamura. Control of logic program execution based on the functional relations. In *Proceedings of the Third International Conference on Logic Programming*, pages 505–512, London, England, June 1986.

[6] Hajime Sawamura and Taku Takeshima. Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to Prolog optimization. In *Proceedings of the Second International Conference on Logic Programming*, pages 200–207, Boston, Massachusetts, June 1985.

[7] Gert Smolka. TEL (version 0.9) report and user manual. Technical Report SEKI Report SR-87-11, Department of Computer Science, University of Kaiserslautern, Germany, February 1988.

[8] Gert Smolka. Making control and data flow in logic programs explicit. In *Conference Record of the ACM Symposium on LISP and Functional Programming*, pages 311–322, Austin, Texas, July 1984.

[9] Zoltan Somogyi, Fergus Henderson and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, to appear.