Determinism analysis in the Mercury compiler

Fergus Henderson, Zoltan Somogyi and Thomas Conway
{fjh,zs,conway}@cs.mu.OZ.AU
Fax: +61 3 348 1184, Phone: +61 3 282 2401
Department of Computer Science, University of Melbourne
Parkville, 3052 Victoria, Australia

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 and to detect determinism errors.

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

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. On the positive side, 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 negative side, 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 rejects the program.

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 [8]. 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 easily be 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 overhead 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. Most of the code samples we show in this paper are from the compiler, and illustrate both a problem and (part of) its solution.

The rest of this paper is organized as follows. Section 2 describes Mercury's type and mode systems, the foundations on which the determinism system is built, 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, programming style, and the efficiency of the generated code.

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 [4], 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 `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 [5], and to the type system of Gödel [4]. 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 ; multidet ; erroneous ; failure.
:- type can_fail    ---> can_fail ; cannot_fail.
:- type soln_count  ---> at_most_zero ; at_most_one ; at_most_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 determinism_components(determinism, can_fail, soln_count).
:- 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 determinism_components(in, out, out).
:- mode determinism_components(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 data-flow 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 determinism_components:

```
:- pred determinism_components(determinism, can_fail, soln_count).
:- mode determinism_components(in, out, out) is det.
:- mode determinism_components(out, in, in) is det.

determinism_components(det,       cannot_fail, at_most_one).
determinism_components(semidet,   can_fail,    at_most_one).
determinism_components(multidet,  cannot_fail, at_most_many).
determinism_components(nondet,    can_fail,    at_most_many).
determinism_components(erroneous, cannot_fail, at_most_zero).
determinism_components(failure,   can_fail,    at_most_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.

Computations that always fail may not export any bindings, and can be replaced by the builtin goal `fail`. The determinism `erroneous` is used for procedures that operationally correspond to aborting execution; from the program's point of view they neither succeed nor fail. (The logical semantics for erroneous procedures is that they loop, i.e. evaluate to undefined.) The builtin predicate `error` has the determinism erroneous. Programs call it 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 erroneous calls 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 `determinism_components`, 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 it is one of our design goals that Mercury be a pure language, we must find a declarative framework within which input and output (which in some sense inherently require side effects) can be done. To do this we introduce a concept of *uniqueness* where we say that an object is unique if there is only one live reference to it at any point in the computation. Two special modes that are used with unique modes are `destructive_input` or `di` and `unique_out` or `uo`. The former indicates that the initial instantiation of the variable is ground and unique, and the final state is dead; the latter indicates an initial instantiation of free and a final state of ground and unique. We represent the external state of the world as a unique object, and predicates that perform I/O are relations have two extra arguments that specify the state of the world before and after the I/O operation. The main predicate of a program is a relation between the initial state of the world and the final state.

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 programs that do more than trivial amounts of I/O.

An extra constraint on the state of the world is that it must not only be unique during forward execution, but 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 deterministic code.

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.

Mercury has a standard library which includes modules for lists, stacks, queues, priority queues, sets, bags (multi-sets), maps (dictionaries), random number generation, input/output, and filename and directory handling.

3 Switch detection

Consider the predicate `partition_disj` 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_disj` is deterministic. Not only are all procedures called in the definition declared deterministic with the exception of the semidet procedure called in the condition of the if-then-else (you have to trust us on this), 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 already bound on entry to the disjunction that 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 `determinism_components` predicate from the previous section is:

```
determinism_components(Headvar1, Headvar2, Headvar3) :-
        ( Headvar1 = det,       Headvar2 = cannot_fail, Headvar3 = at_most_one
        ; Headvar1 = semidet,   Headvar2 = can_fail,    Headvar3 = at_most_one
        ; Headvar1 = multidet,  Headvar2 = cannot_fail, Headvar3 = at_most_many
        ; Headvar1 = nondet,    Headvar2 = can_fail,    Headvar3 = at_most_many
        ; Headvar1 = erroneous, Headvar2 = cannot_fail, Headvar3 = at_most_zero
        ; Headvar1 = failure,   Headvar2 = can_fail,    Headvar3 = at_most_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_disj` (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

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

partition_disj([], _Var, Cases, []) :-
        map__init(Cases).
partition_disj([Goal0 | Goals], Var, Cases, Left) :-
        partition_disj(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, DisjList1) ->
                        DisjList = [Goal | DisjList1]
                ;
                        DisjList = [Goal]
                ),
                map__set(Cases1, Functor, DisjList, Cases),
                Left = Left1,
        ;
                MaybeFunctor = no,
                Cases = Cases1,
                Left = [Goal0 | Left1]
        ).
```

Figure 1: The predicate partition_disj

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.

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 disjuncts with this property. 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 only caller of `partition_disj` is the predicate `detect_switches_in_disj`, shown in figure 2. The inputs to this predicate are (two copies of) the list of nonlocal variables (variables that occur both inside and outside the disjunction), the list of disjuncts, information about the instantiation states and the types of variables, and a list of partial switches found so far (initially empty).

```
:- pred detect_switches_in_disj(list(var), list(var), list(goal),
        map(var, inst), map(var, type), list(partial), goal).
:- mode detect_switches_in_disj(in, in, in, in, in, in, out) is det.

detect_switches_in_disj([Var | Vars], AllVars, Goals0, InstMap, TypeMap,
                Partials0, Goal) :-
        (
                is_var_bound(Var, InstMap),
                partition_disj(Goals0, Var, Cases, Left),
                map__to_assoc_list(Cases, CasesList),
                CasesList = [_,_|_]
        ->
                ( Left = [] ->
                        cases_to_switch(CasesList, Var, InstMap, TypeMap, Goal)
                ;
                        Partials1 = [partial(Var, Left, CasesList) | Partials0],
                        detect_switches_in_disj(Vars, AllVars, Goals0, InstMap, TypeMap,
                                Partials1, Goal)
                )
        ;
                detect_switches_in_disj(Vars, AllVars, Goals0, InstMap, TypeMap,
                        Partials0, Goal)
        ).
detect_switches_in_disj([], AllVars, Goals0, InstMap, TypeMap,
                Partials0, disj(Goals)) :-
        (
                Partials0 = [],
                detect_sub_switches_in_disj(Goals0, InstMap, TypeMap, Goals)
        ;
                Partials0 = [Partial0 | Partials1],
                select_best_partial_switch(Partial0, Partials1, BestPartial),
                BestPartial = partial(Var, Left0, CasesList),
                cases_to_switch(CasesList, Var, InstMap, TypeMap, SwitchGoal),
                detect_switches_in_disj(AllVars, AllVars, Left0, InstMap, TypeMap,
                        [], Left),
                goal_to_disj_list(Left, LeftList),
                Goals = [SwitchGoal | LeftList]
        ).
```

Figure 2: The predicate detect_switches_in_disj

The predicate looks at the nonlocal variables one by one. If a nonlocal variable is bound on entry to the disjunction, and if `partition_disj` discovers that it is tested against at least two different function symbols in the various disjuncts, then we can use this variable to create a switch. If all disjuncts test the value of the variable (none are left out) then we convert the disjunction into a switch immediately. `cases_to_switch` does this: it sets a flag on the switch if the various arms of the switch cover all the function symbols in the type of the variable (this flag is used in the algorithm described in section 5), and it looks for disjunctions inside the new switch and tries to turn them into switches as well. If there are some disjuncts that do not test the variable, then the the value of the variable discriminates only among *some* disjuncts, and a switch on the variable would be only partial. Since some other variable may yet provide a total switch, we record the details of the partial switch but do not act on it immediately.

Control reaches the base case of `detect_switches_in_disj` only if no nonlocal variable yields a switch covering all disjuncts. If the algorithm has recorded no partial switches either, this

disjunction cannot be turned into any kind of switch, and the most that can be done is to see if any disjunctions nested inside can be turned into switches. If the algorithm has recorded at least one partial switch, we must choose one from the list 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; the nature of the heuristic used does not seem to matter much in practice. After we create the selected partial switch, we put the disjuncts left out of the partial switch through `detect_switches_in_disj` again, since some other variable may yield a switch on these disjuncts. Whether or not it does, we return a disjunction containing our partial switch and one or more other goals derived from the disjuncts left out of our partial switch.

When this algorithm processes the predicate `determinism_components` in its forward mode, the only variable whose value is known on entry to the disjunction is Headvar1. The call to `partition_disj` 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 `determinism_components` 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.

Consider the following definition of the predicate merge, which does a nondeterministic merge of two lists:

```
:- pred merge(list(T), list(T), list(T)).
:- mode merge(in, in, out) is multi.

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

When our algorithm processes this predicate, it finds two partial switches, one on Headvar1 and one on Headvar2. Both switches have two arms and leave out two disjuncts, so the choice between them is arbitrary. Regardless of which one is chosen, the recursive invocation of the algorithm on the two disjuncts left out will find the other switch, so the final goal will be a disjunction containing the two two-way switches.

4 Common subexpression elimination

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'd 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)) and gives names to the anonymous variables, though not as expressive names as the ones we use in this paper:

```
count_elements(Headvar1, Headvar2) :-
        ( Headvar1 = [], Headvar2 = zero
        ; Headvar1 = [Head1 | Tail1], Tail1 = [], Headvar2 = one
        ; Headvar1 = [Head2 | Tail2], Tail2 = [Head3 | Tail3], 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 turns 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 they don't, or if the first disjunct contains no tests 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, since it would not provide any additional opportunities for switch detection, 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 removes the unification that does this from the first disjunct, puts it outside the disjunction, and replaces the equivalent unifications inside the other disjuncts with code that for each argument of the function symbol, unifies the two variables representing that argument in the first disjunct and the disjunct under consideration. The result is to turn a disjunction like this:

```
( Headvar1 = [Head1 | Tail1], Tail1 = [], Headvar2 = one
; Headvar1 = [Head2 | Tail2], Tail2 = [Head3 | Tail3], Headvar2 = many
)
```

into a goal like this:

```
Headvar1 = [Head1 | Tail1],
( Tail1 = [], Headvar2 = one
; Head1 = Head2, Tail1 = Tail2, Tail2 = [Head3 | Tail3], 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 Tail1. (After the goal `Tail1 = Tail2`, the current substitution records that Tail1 and Tail2 are effectively the same variable, so a test of Tail2 against a cons cell is effectively a test of Tail1

Left	Right	Conj	Disj	Switch
can_fail	can_fail	can_fail	can_fail	can_fail
can_fail	cannot_fail	can_fail	cannot_fail	can_fail
cannot_fail	can_fail	can_fail	cannot_fail	can_fail
cannot_fail	cannot_fail	cannot_fail	cannot_fail	cannot_fail

Table 1: Can_fail combinations

Left	Right	Conj	Disj	Switch
at_most_zero	at_most_zero	at_most_zero	at_most_zero	at_most_zero
at_most_zero	at_most_one	at_most_zero	at_most_one	at_most_one
at_most_zero	at_most_many	at_most_zero	at_most_many	at_most_many
at_most_one	at_most_zero	at_most_zero	at_most_one	at_most_one
at_most_one	at_most_one	at_most_one	at_most_many	at_most_one
at_most_one	at_most_many	at_most_many	at_most_many	at_most_many
at_most_many	at_most_zero	at_most_zero	at_most_many	at_most_many
at_most_many	at_most_one	at_most_many	at_most_many	at_most_many
at_most_many	at_most_many	at_most_many	at_most_many	at_most_many

Table 2: Max_soln combinations

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 infinite loop, since each iteration reduces the number of function symbols.

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 three main algorithms, which are the subjects of the following three 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 at an early stage of the compiler into a combination of negation and existential quantification.)

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 = Y with mode (in,in) where the shared type of X and Y is atomic are *simple tests*, and are always semidet. 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 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),
        determinism_components(Detism1, CanFail1, MaxSolns1),
        determinism_components(Detism2, CanFail2, MaxSolns2),
        conjunction_canfail(CanFail1, CanFail2, CanFail),
        conjunction_maxsoln(MaxSolns1, MaxSolns2, MaxSolns),
        determinism_components(Detism, CanFail, MaxSolns).
```

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 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 `conjunction_canfail`, while the third column of table 2 gives the definition of `conjunction_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 (cannot_fail and at_most_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 can_fail and at_most_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 do not 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 cannot_fail and at_most_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 in the type of the variable being switched on; 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:

```
conjunction_maxsoln(CondMaxSolns, ThenMaxSolns, AllThenMaxSolns),
switch_maxsoln(AllThenMaxSolns, ElseMaxSolns, MaxSolns),
switch_canfail(ThenCanFail, ElseCanFail, CanFail),
determinism_components(Detism, CanFail, MaxSolns).
```

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 `conjunction_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. If the 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. Therefore the negated goal can never succeed more than once.

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`, `determinism_components`, 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 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).

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. One component of the recorded determinism can change from cannot_fail to can_fail, but not back; the other can change from at_most_zero to at_most_one to at_most_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 multidet 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.

5.3 Determinism error diagnosis

Determinism analysis is a bottom-up process; it combines the determinisms of primitive goals to find the determinism of compound goals. Determinism error diagnosis is a top-down process. Given the desired determinism of a compound goal (initially the procedure definition), it finds the loosest possible determinism of each of the subgoals, and looks for subgoals that go outside their boundaries.

The rules for computing the loosest possible determinisms of subgoals given a desired determinism for a compound goal are effectively the same as the rules we described in section 5.1, they are just viewed from a different perspective. Therefore there wouldn't be much point in a detailed description of these rules. We think it is much more important to give some examples of the kinds of error messages generated by the Mercury compiler for determinism errors. We expended considerable effort to make these messages as useful to the programmer as possible (although there is still room for improvement).

The first step in this effort was to ensure that all lines in all error messages and warnings generated by the Mercury compiler start with the file name and the line number of the location of the error in a standard format that is parsable by the Unix utility "error" or by editors such as emacs, elvis, vim, etc. The "error" utility takes error messages with location information and inserts each one them into the named file at the named line. This makes it considerably easier to fix several errors in a row, because even if a fix to one error adds or deletes lines, the messages describing other errors are already attached to their locations, and thus it no longer matters that the line numbers of the program diverge from those in the error messages. Error can even invoke the programmer's designated editor on the affected files, with the cursor positioned on the first error message.

Consider the `infer_conj` predicate from section 5.1. If the programmer forgets its base clause, the compiler prints these messages:

```
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.
```

The first three lines report the location of the unsatisfied determinism declaration, the name of the predicate affected, the affected mode, and the declared and inferred determinisms. The next two lines indicate the subgoal that the compiler believes to be the underlying cause of the determinism error. For unifications that involve variables introduced by the compiler to denote subterms of other terms, the error message indicates the path to the relevant subterm. As an

example, consider the predicate for inferring the determinism of unifications. One of its clauses computes the determinism of deconstructions:

```
infer_unify(deconstruct(_, _, _, _, CanFail), _MiscInfo, Detism) :-
        determinism_components(Detism, CanFail, at_most_one).
```

(The CanFail field of the deconstruct structure is filled by mode analysis. Switch detection may replace the structure with another one containing a different value of the field (cannot_fail) if the deconstruction is in the correct arm of a switch.) If this clause had been written like this,

```
infer_unify(deconstruct(_, _, _, _, can_fail), _MiscInfo, Detism) :-
        determinism_components(Detism, can_fail, at_most_one).
```

i.e. had the programmer insisted on the value of the field being can_fail, the compiler would have issued this diagnosis:

```
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 `can_fail' can fail.
```

If the programmer had omitted the entire clause instead, the compiler would have issued this diagnosis:

```
det_analysis.m:580:   The switch on HeadVar1 does not cover deconstruct/5.
```

In this case, the other four clauses tell the compiler that `infer_unify` is a switch, and therefore it can diagnose the problem accurately. In the example above involving `infer_conj`, the programmer was almost certainly trying to write a switch, but the compiler did not see any switch and therefore couldn't infer the programmer's intention.

6 Impact of determinism information

6.1 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 fully deterministic computation 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 points out all such problems at compile time. This 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 benefits.

But perhaps the biggest benefit of the determinism system is that it makes programs significantly easier to maintain. The Mercury compiler is 75,000 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. In our experience this approach is 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. According to CVS, our configuration management system, the files defining the main data structures of the front and back ends of the Mercury compiler have been modified 172 and 147 times respectively since their creation. Before the compiler's type, mode and determinism analysis modules started working, we executed the Mercury compiler through a Prolog system. At that time, this safety net did not exist, and as a consequence we were quite reluctant to make such changes, even though the system was much smaller and fewer programmers were involved.

6.2 Programming style

The determinism system of Mercury impacts upon programming style in a number of ways. Most Prolog systems (including Quintus and SICStus) index only on the top level functor of the first argument of clause heads, and do not index on disjunctions at all. (Aquarius Prolog has a similar scheme to ours, but very few people use Aquarius.) This impacts significantly on programming style since to get acceptable efficiency from a portable Prolog program, the programmer has to introduce numerous auxiliary predicates in order to get the benefit of indexing. (Consider the predicate in figure 1.) In Mercury, such predicates are not needed.

Sometimes a program needs to classify a value as being in or out of a particular set. The obvious way of writing this is to have a clause for each alternative of the type that is in the set. Once again, there is information implicit in the missing clauses, and if the definition of the type changes, it is easy to forget that new clauses for such predicates may be necessary. A more robust way of writing such predicates that uses Mercury's determinism system effectively is to make the predicate deterministic and have it map the alternatives of the type to a boolean value. In this style, there is a clause for every alternative of the type, and any change to the type definition without a corresponding change to the predicate will cause the Mercury compiler to report an error.

Some Prolog programs use a data representation which distinguish only some elements of a type with a function symbol. For example, when a Prolog metainterpreter looks at goal, it looks for certain function symbols (e.g. comma, semicolon etc), but it considers any other function symbol to represent a call. The code that works with such a "defaulty" representation usually has a last clause that matches any input, which means the predicate cannot be deterministic without cuts. This is one reason why Richard O'Keefe argues against this technique [7]. His suggested alternative (which is the only alternative in Mercury) is a representation that fully discriminates among all the alternatives, which naturally leads to code that Mercury can turn into a switch.

When writing a predicate that is a case analysis on one of its arguments, the programmer sometimes knows that the argument cannot have a certain value. A typical Prolog programmer would simply omit that value from the case analysis. To make the predicate deterministic, a Mercury programmer would have to include a case for the value, and would either make the predicate return an indication or call the built-in predicate "error" if that value was actually encountered. (The declarative semantics of error is that it loops forever. Operationally it prints an error message and aborts program execution. We are considering a more general exception handling system for Mercury.) The second technique is significantly more robust, because what is a value that is illegal today may be produced tomorrow, perhaps because of a logical bug

somewhere else in the program. It also provides much better documentation for maintainers, since the cases that a predicate cannot handle are explicitly stated rather than being implicit in missing clauses.

Deterministic and nondeterministic regions of the computation tend to be better separated in Mercury than in Prolog, which makes programs easier to maintain. There are two ways that code with at most one solution can call code with potentially more than one solution. If code that may have more than one solution has all its output variables existentially quantified, then from outside the quantifier, all the possible solutions are indistinguishable (not even side-effects can distinguish them, since Mercury is a pure language). Therefore the containing goal need only find whether or not a solution exists. In this situation, the compiler generates a commit across the nondeterministic goal to prevent backtracking finding any further solutions, since they would be redundant. The other way in which code with at most one solution can call code with potentially more than one solution is by using the all solutions predicate `solutions/2`. This predicate takes a goal with a single output variable as its first argument and produces as its second argument a sorted list of all the solutions to the goal with duplicates removed. (Note that this is a logical all solutions predicate, unlike those found in most Prolog systems.)

6.3 Code generation

One of the key differences between logic programming languages on the one hand and imperative and functional programming languages on the other hand is that a call in an imperative or functional language will return once, while a call in a logic language may return several times. This imposes substantial costs on the implementation. Consider the stack frame of a call to a predicate that can return multiple solutions. This frame cannot be deallocated when the call succeeds, because the failure of a later goal may cause backtracking back into the call, asking it for another solution. The production of this solution may require access to the values of the input arguments of the call. The code of a deterministic procedure is free to reuse the registers and/or stack slots occupied by the input arguments (and other variables) once execution has passed the point of their last use in the production of the first (only) solution, but the code of a procedure that can succeed more than once must keep many of these values around in case they are needed for the computation of later solutions. Therefore the stack frames of nondet and multi procedures are both larger and much longer lived than the stack frames of other procedures [1].

Functional and logic languages do not allow programmers to express destructive update operations directly; programmers must make a slightly modified copy of the data structure instead. This is why their implementations have traditionally been very allocation-intensive which usually leads to bad locality. Systems using compile-time garbage collection address this problem by arranging for the compiler to find out which are structures are dead (will not be referenced again) and to reuse their storage. Unfortunately, the presence of calls that may succeed more than once severely limits the applicability of this important technique, because backtracking may cause accesses to many structures that would otherwise be dead.

Mercury's determinism system enables the compiler to optimize the implementations of predicates that have at most one solution. The compiler arranges for the stack frames of such predicates to be popped on success, leading to stack behavior similar to that seen in conventional languages. We are currently implementing structure reuse which also relies on predicates having only a single solution. The code emitted by the compiler thus pays the penalties of multiple solutions only when required. This is a very important improvement. Over 90% of the procedures in the Mercury compiler are det or semidet. This figure may not be truly representative of Mercury programs, since to make bootstrapping easier, the compiler does not make use of all the language features. However, we have found that even many heavily nondeterministic programs (e.g. the nine-queens benchmark) spend most of their time in det or semidet procedures,

and therefore they benefit significantly from the better treatment of such procedures.

7 Conclusion

The concept of determinism in logic programs has been around for a long time. Several researchers [2, 3, 6] 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 [9, 10] 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 80,000 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 approach is also much more effective in improving the efficiency of compiled code. Benchmarks show the Mercury compiler to be about twice as fast as Aquarius Prolog, about five times as fast as SICStus Prolog, and about ten times as fast as Quintus Prolog [11].

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 [12] and Gödel [4]) 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.

The first public beta release of the Mercury system was on 18 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] T. Conway, F. Henderson, and Z. Somogyi. Code generation for mercury. In *Proceedings of the Twelfth International Conference on Logic Programming*, Portland, Oregon, December 1995.

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

[3] S. K. Debray and D. 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.

[4] P. M. Hill and J. W. Lloyd. *The Gödel programming language*. MIT Press, 1994.

[5] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[6] K. 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.

[7] R. A. O'Keefe. *The craft of Prolog*. MIT Press, 1990.

[8] H. Sawamura and T. 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.

[9] G. 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.

[10] G. 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.

[11] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, to appear.

[12] J. Thom and J. Zobel. NU-Prolog reference manual. Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.