Status of the Mercury system

Zoltan Somogyi, Fergus Henderson, Thomas Conway, Andrew Bromage,
Tyson Dowd, David Jeffery, Peter Ross, Peter Schachte and Simon Taylor
{zs,fjh,conway,bromage,trd,dgj,petdr,pets}@cs.mu.oz.au
Department of Computer Science, University of Melbourne
Parkville, 3052 Victoria, Australia
Phone: +61 3 9287 9160
Fax: +61 3 9348 1184

Keywords: Mercury, types, modes, determinism, functions, higher-order, implementation, optimizations, portability, performance.

1. Introduction

This paper is a report on the status of the Mercury system at the time of the release of version 0.6 of the system. We intend to demonstrate the system at the workshop.

The contents of this paper is as follows. Section 2 lists the main features of the language. Section 3 shows what is new in the Mercury system since release 0.5, which came out in early 1996. Section 4 gives a broad picture of the state of the implementation. Section 5 discusses the performance of the system, while section 6 lists our current and future research directions.

2. The Mercury language

Mercury is a new general-purpose programming language based on the paradigm of purely declarative programming, designed and implemented by a small group of researchers at the University of Melbourne, Australia. While Prolog is oriented towards exploratory programming, we designed Mercury to be useful for the development of large and robust "real-world" applications. It improves on existing logic programming languages by providing increased productivity, reliability and efficiency, and by avoiding the need for non-logical program constructs. Mercury provides the traditional logic programming syntax, but also allows the syntactic convenience of user-defined functions, smoothly integrating logic and functional programming into a single paradigm. Due to the declarativeness of the language, the availability of functions and the language's emphasis on the software engineering qualities of programs, programming in Mercury has a different flavor than programming in Prolog.

The main features of Mercury are:

- Mercury is a strongly typed language. Mercury's type system is based on many-sorted logic with parametric polymorphism, very similar to the type systems of modern functional languages such as ML and Haskell. Programmers should declare the types they need using declarations such as

```
:- type list(T) ---> [] ; [T | list(T)].
:- type maybe(T) ---> yes(T) ; no.
```

They should also declare the type signatures of the predicates and/or functions they define, for example

```
:- pred append(list(T), list(T), list(T)).

:- func sin(float) = float.
```

The compiler infers the types of all variables in the program. Type errors are reported at compile time.

- Mercury is a strongly moded language. The programmer should declare the instantiation state of the arguments of predicates at the time of the call to the predicate and at the time of the success of the predicate. Currently only a subset of the intended mode system is implemented. This subset effectively requires arguments to be either fully input (ground at the time of call and at the time of success) or fully output (free at the time of call and ground at the time of success).

  A predicate or function may be usable in more than one mode. For example, append is usually used in at least these two modes:

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

  If a predicate or function has only one mode, the mode information can be given in the predicate declaration.

  ```
  :- pred factorial(int::in, int::out).
  ```

  Mode declarations for functions are optional; if they are omitted, the default is that the function has one mode, in which all the arguments are input and the result is output.

  For each mode of a predicate or function, the compiler will infer the mode of each call, unification and other builtin in the body. It will reorder the conjunctions in the body as necessary to find a left to right execution order; if it cannot do so, it rejects the program. Like type-checking, this means that a large class of errors are detected at compile time. Note that different modes of a predicate or function will in general be reordered differently. In fact, the compiler treats each mode of a predicate independently after mode analysis, even generating separate code for each mode.

- Mercury has a strong determinism system [6]. For each mode of each predicate or function, the programmer should declare whether it will succeed exactly once (det), at most once (semidet), at least once (multi) or an arbitrary number of times (nondet). These declarations are attached to mode declarations like this:

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

  :- pred factorial(int::in, int::out) is det.
  ```

The compiler will try to prove the programmer's determinism declaration using a simple, predictable set of rules that seems sufficient in practice (the problem in general is undecidable). If it cannot do so, it rejects the program.

As with types and modes, determinism checking catches many program errors at compile time. It is particularly useful if some deterministic (det) predicates or functions each have a clause for each function symbol in the type of one of their input arguments. If this type changes, the compiler will report determinism errors for all of these predicates or functions, telling the programmer to put in code to cover the case when the input argument is bound to a newly added function symbol.

- Mercury allows programmers to express information about the uniqueness of data structures through the mode system. One simple way to do this is to replace '`in`' with '`di`' (short for "destructive input") and '`out`' with '`uo`' (short for "unique output") in the mode declarations of some of the arguments of a predicate or function. If an argument is declared with mode `uo`, the compiler will check that there will be no other references to the term returned as that argument when the predicate or function succeeds. If an argument is declared with mode `di`, the compiler will check that there will be no other references to the term passed as that argument when the predicate or function is called. Therefore the compiler is free to emit code that reuses the storage occupied by `di` arguments. (The compiler doesn't yet exploit this freedom except for builtin types such as '`io__state`' and '`unique_array`'.)

  A predicate or function can have several mode declarations that differ only in their uniqueness information:

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

  For each call, the compiler will pick the first mode whose conditions are satisfied.

- Mercury supports higher-order programming, with closures, currying, and lambda expressions.

  The type, mode and determinism of any higher order arguments of a predicate or function are stated in the declaration of the predicate or function. The standard map predicate, which takes a predicate as an argument and applies this predicate to each element of a list, can be implemented like this:

```
:- pred map(pred(T1, T2), list(T1), list(T2)).
:- mode map(pred(in, out) is det, in, out) is det.
:- mode map(pred(out, in) is det, out, in) is det.
:- mode map(pred(in, in) is semidet, in, in) is semidet.

map(_Pred, [], []).
map(Pred, [X | Xs], [Y | Ys]) :-
        call(Pred, X, Y),
        map(Pred, Xs, Ys).
```

  (It would be possible to add other modes and determinisms for map as well.) The next two calls to map demonstrate the use of currying and lambda expressions:

```
map(append([1]), [[2,3], [4,5]], A)
map((pred(Old::in, New::out) is det :- New is Old + 1), [6, 7], B)
```

- Mercury has a modern module system. Programs consist of one or more modules. Each module has an interface section that contains the declarations for the types, insts, modes, predicates and functions exported from the module, and an implementation section that contains the definitions of the exported entities and also definitions for types, insts, modes, predicates and functions that are local to the module. A type whose name is exported but whose definition is not, can be manipulated only by predicates in the defining module; this is how Mercury supports abstract data types.

  For predicates and function that are not exported, Mercury supports automatic type, mode and determinism inference.

- Last but far from least, Mercury is purely declarative: predicates in Mercury do not have non-logical side effects.

  Mercury does I/O through built-in library predicates that take an old state of the world and some other parameters, and return a new state of the world and possibly some other results. The language requires that the input argument representing the old state of the world be the last reference to the old state of the world, thus allowing the state of the world to be updated destructively. The language also requires that I/O take place only in parts of the program that are guaranteed to succeed exactly once, so that backtracking across the update will never be necessary.

  Mercury handles dynamic data structures not through Prolog's assert/retract but by providing several abstract data types in the standard Mercury library that manage collections of items with different operations and tradeoffs. Being a compiled language, Mercury does not have any means for altering the program at runtime, although we may later provide facilities for adding code to a running program.

Mercury programs have a straightforward declarative semantics, which is the Clark completion of a transformed version of the program; the transformations required are the standard DCG transformation, module qualification, and transformations to convert programs containing higher-order terms into first-order programs [11] and typed programs into untyped programs. Alternatively, one could omit the last transformation and give a semantics directly in terms of a parametric polymorphic version of many-sorted logic, and/or omit the second last transformation and give a semantics directly in terms of a higher-order logic. The semantics of Mercury programs is discussed in some detail in the Semantics chapter of the Mercury language reference manual [4].

The language design makes it significantly easier to create efficient implementations whose operational semantics is sound with respect to this declarative semantics. The mode system allows the compiler to guarantee the safety of negations (including those implicit in if-then-elses etc), the absence of floundering (any goal that would flounder has a mode error), and the absence of any need for the occur check.

For more details on the language, please see our previous papers [2, 6, 8] or the language reference manual [4].

3. What is new in the Mercury system

We have continued our development of the language after version 0.5 of the Mercury system was released in February 1996. The features we discuss below are now available in version 0.6, which was released on 2 August.

We have always intended to support functions in the final version of Mercury. Spurred partly by our discussions with other researchers at ILPS '95, we have recently implemented functions, including higher-order functions, in the Mercury compiler. These are defined using either conditional or unconditional equations. Examples:

```
:- func sum(list(int)) = int.
sum([]) = 0.                          % unconditional equations
sum([X | Xs]) = X + sum(Xs).          % using function calls

:- func max(int, int) = int.
max(X, Y) = Max :-                    % conditional equation
        (X >= Y -> Max = X ; Max = Y).

:- func min(int, int) = int.
min(X, Y) = (if X >= Y then X else Y).  % unconditional equation
                                        % using if-then-else expression
```

Calls to functions may appear anywhere a constructor can, e.g. in the arguments of predicate calls or in unifications. For each functor in the program, the typechecker figures out whether the functor is a data constructor or a call to an executable function.

The builtin arithmetic operations are now implemented as functions, and these have reverse as well as forward modes. For example, the declaration of the subtraction operation is:

```
:- func int - int = int.
:- mode in  - in  = uo  is det. % uo is "unique out"
:- mode uo  - in  = in  is det.
:- mode in  - uo  = in  is det.
```

This means that a goal such as A = B − C can be used not only to compute A from B and C, but also B from A and C or C from A and B.

The language allows functions (and predicates) to be implemented lazily. However, the current compiler does not do so. It implements functions by flattening, i.e. internally it treats a function with N arguments as if it were a predicate with N+1 arguments.

Again partly due to feedback at ILPS '95, we have recently extended semantic analysis to support type and mode inference. (We have supported determinism inference for a long time now, although this was unfortunately broken in version 0.5.) In version 0.6, programmers may specify the options '`--infer-types`' and '`--infer-modes`', which tell the compiler to consider '`:- func`', '`:- pred`' and '`:- mode`' declarations to be optional for functions and predicates that are local to a module; the compiler will infer the types and modes automatically. (Allowing these declarations to be omitted for functions or predicates that are exported from a module would require cross-module inference and thus interfere with separate compilation. Such omissions would also make it significantly harder for other programmers to make use of the functionality of the module.)

The existing type and mode checking algorithms infer the types and modes of each clause, using the declared types/modes of the called predicates/functions, and check that it matches the types/modes specified in the declaration for the predicate/function being checked. Type and mode inference basically work by starting with some initial assumptions about the types/modes of undeclared predicates, and then iterating the existing type and mode checking algorithms until a fixpoint is reached. Since the underlying mode system includes support for "unique" modes that can be used for destructive update, it was not very hard to make mode inference also infer these. Similarly, since the underlying mode system includes support for higher-order modes, it was easy for mode inference to infer such modes.

One limitation of our mode inference algorithm is that for simplicity of implementation, mode inference will delay calls only to predicates whose modes are declared. Therefore calls to such predicates must appear in the proper place within the calling predicate or function, otherwise the system will not be able to infer the right mode. The other limitation is that we have to place some bounds on the complexity of the inferred modes in order to ensure termination of mode inference; we're still determining the best way of doing this. Despite these limitations, the current implementation is able to infer modes for most of the predicates in the Mercury library, but fails on a few of the more complicated ones which use higher-order predicates in complicated ways, as a result of the current bounds being overly restrictive.

Although our current implementation of type inference has some problems with details such as the proper handling of overloading, we believe we have the found an algorithm that is guaranteed both to terminate and to infer the most general type assignment for all the predicates in the program if one exists. We expect this algorithm to be quite efficient, especially since it does not involve fixpoint iteration.

4. The Mercury implementation

The Mercury compiler is written in Mercury itself, and generates C code. Its structure is rather conventional: it is a series of phases covering syntax analysis, semantic analysis, high-level optimizations, code generation, and low-level optimizations.

We now allow programmers to interface Mercury and C code: Mercury code can call C code and vice versa (potentially recursively). When giving the definition of a Mercury predicate, the programmer may choose not to give a list of clauses but to instead write down the C code that should be executed when that predicate is called in a given mode. This was not too difficult to implement, given that the compiler generates C code (either ANSI C or C using the GNU C extensions). Since C doesn't support nondeterminism, implementing nondeterministic predicates or functions in C would not be straightforward. At the moment we require Mercury predicates defined by C code to be `det` or `semidet`.

Programmers may also declare that a given mode of a Mercury predicate may be called from C code. The declaration associates a C function name with that mode of that predicate. The Mercury compiler will define this C function in the C source file it generates for the module. It will also emit the function prototypes for these functions into a C header file associated with the module; this header file should be #included by the C code that wants to call the Mercury predicate. We currently allow C code to call only `det` Mercury predicates.

Our philosophy is that if there is an algorithm for an optimization, then it should be done by the compiler, not the programmer. We are therefore continuously working to add new optimizations to the Mercury compiler. The current list of optimizations supported by the compiler is:

- allocation of stack slots to variables by graph colouring
- using hash tables for indexing on strings

- using dense jump tables for indexing on many other types
- using array lookups instead of indexing tables of facts
- middle-recursion optimization
- inlining of small predicates
- inlining of predicates called only once
- elimination of "copy" unifications
- elimination of unused arguments
- propagation of higher-order constants
- peephole optimization
- short circuiting of jumps, calls and returns
- replacement of jumps by the instruction sequence at the target label
- delay of stack frame setup
- filling branch delay slots
- elimination of dead code
- elimination of duplicate code
- value numbering

We exploit the GNU C extensions to C (global register variables, nonlocal gotos, better support for assembler pseudo-ops) whenever possible. However, the only places where we use assembly code are register names in the declarations of global register variables, assembler pseudo-ops for creating labels, and a small number of instructions to save and restore the global pointer on Alphas. This approach has helped us to achieve a significant degree of portability. Mercury now works on a large number of platforms:

- Sun SPARC based machines running Solaris 2.x or SunOS 4.x
- DEC Alpha based machines running OSF/1 3.x
- SGI MIPS based machines running IRIX 5.x
- IBM POWER based machines running AIX 3.x or 4.x
- HP Precision Architecture based machines running HP-UX 9.x
- x86 PCs running Linux (ELF or COFF)
- x86 PCs running BSDI 1.1
- x86 PCs running Windows 95 (using gnu-win32)

We have recently started work on a debugger for Mercury programs. Our design for the debugger supports some parts of the program being loaded in bytecode form while the other parts are loaded in native code form. This will allow the debugging of even large programs with acceptable speed. The compiler started generating bytecode for the debugger in May 1996. We eventually plan to implement a declarative debugger for Mercury. The absence of side-effects should make this much easier, and the availability of type, mode and determinism information should significantly improve the usability of the debugger. For example, programmers should be able to specify type-specific prettyprinters that display different parts of a data structure in different levels in detail; when the programmer says that a particular part of a computed term is wrong, the debugger ought to be able to jump directly to the predicate or function that computed that part of the term; and determinism declarations ought to enable the debugger to avoid asking many questions about the intended success/failure of predicates and functions.

The Mercury system already has a profiler [5] which produces output very similar to the output of gprof [3]. This profiler gives feedback to programmers not only about which predicates and functions the program is spending most of its time in, but also where these predicates and functions are called from. This is important, since a program can be sped up not only by optimizing expensive predicates and functions but also by eliminating calls to them. The profiler works even with large programs; we have successfully used the profiler to improve the speed of the Mercury compiler itself.

system	speed
SWI-Prolog	1.00
NU-Prolog	1.95
wamcc	3.14
Quintus Prolog	3.74
SICStus Prolog, compactcode	2.22
SICStus Prolog, fastcode	7.41
Aquarius Prolog	19.00
Mercury, Boehm gc	9.24
Mercury, no gc	36.52

Table 1: Benchmark speed ratios on a SPARCserver 1000

5. Performance

Our testing shows that Mercury is the fastest system by a large margin among all the logic programming language implementations we have tested it against. These tests involved a collection of ten standard Prolog benchmark programs which we have transliterated into Mercury. For each system, table 1 gives the harmonic mean of the speedups that system achieved over SWI-Prolog. The table shows Mercury to be almost double the speed of Aquarius Prolog, almost five times the speed of SICStus Prolog fastcode (version 2.1), and about ten times the speed of Quintus Prolog and wamcc.

The current Mercury implementation can run without a garbage collector, or it can use Hans Boehm's conservative garbage collector [1]; we do not yet have a garbage collector native to Mercury. The figure in table 1 significantly understates the performance of Mercury when using the Boehm collector. The reason is that these benchmarks have to be repeated many thousands of times to achieve runtimes that can be measured reliably, and that while the Prolog systems and Mercury without garbage collection can recover all the memory used by one benchmark iteration before starting the next, the Boehm garbage collector cannot. This means that for these small benchmarks, the only system that actually performs garbage collection is the Mercury system with Boehm's collector.

For a more realistic evaluation of the speed of each system, one needs a much larger benchmark. The Mercury compiler and the standard Mercury library are written in the intersection of Mercury, NU-Prolog and SICStus Prolog, and are therefore suitable for a comparison of these three systems. The compiler and library have 103,000 lines of source code totalling about 3.6 Mb between them. Table 2 shows the results of our experiments using the compiler as a large, realistic benchmark. The second column shows the speed ratios of the five systems, this time normalized to the speed of SICStus compactcode, when running the ten small benchmarks used in table 1. The third column

System	speed on small benchmarks	speed on large benchmark	file size for large benchmark
NU-Prolog	0.9	N/A	2.8 Mb
SICStus Prolog, compactcode	1.0	1.0	6.0 Mb
SICStus Prolog, fastcode	3.3	1.5	10.3 Mb
Mercury, Boehm gc	4.2	5.6	2.3 Mb
Mercury, no gc	16.5	N/A	N/A

Table 2: Small vs large benchmarks on a SPARCserver 1000

shows speed ratios normalized the same way using the Mercury compiler as a large benchmark. The fourth column shows the size of the executable or save file for the large benchmark when compiled by each system. (Some entries are missing because executing the large benchmark requires a garbage collector.)

The figures show that although the ratio between the speed of Mercury with Boehm's garbage collector and the speed of SICStus fastcode is only 1.24 for the small benchmarks, it is 3.7 for the large benchmark, a very large difference. This ratio is in fact closer to 4.9, which is the ratio between the speed of Mercury *without* garbage collection and the speed of SICStus fastcode on the small benchmarks. This shows that our use of the Boehm garbage collector has much less performance impact on real programs than on small benchmarks, and that Mercury retains its significant performance advantage even for large programs.

The figures also show that Mercury generates relatively small executables. Mercury compiles 3.6 Mb of source into a 3.0 Mb executable file, of which 2.3 Mb remains after stripping (statically linked except for the standard C libraries). Even NU-Prolog's compact bytecode representation requires more space than this, while SICStus native code requires more than four times as much storage.

The main reasons for the efficiency of the Mercury implementation are the following.

- The guaranteed availability of type information allows us to use a different data representation for each type. Whereas conventional Prolog systems use tags to distinguish among a very limited number of functors, typically just nil and cons, Mercury can apply the same technique to almost all functors. This reduces heap consumption, because if the tag of a pointer to a heap cell uniquely identifies the functor of the term, then the heap cell need not have a field containing this information.
- The compiler knows exactly which variables are bound and which are free at every point in the program. Therefore
  - free variables do not need to be initialized;
  - there is no need for a trail (since resetting a word to garbage is a null operation);
  - builtins (unification, arithmetic operations, etc) do not need to check the state of instantiation of their arguments;
  - assignment unifications and other builtins can simply overwrite their output variable; and
  - callers can pass input arguments by value in registers.
- Our mode system prevents two free variables from ever being aliased. Therefore we never need to perform dereferencing, and callees can return output arguments by value in registers (or in memory, if that turns out to be more efficient).
- Determinism information allows us to avoid incurring the overheads needed to cope with a predicate succeeding more than once unless the predicate can indeed succeed more than once. Most predicates are deterministic or semideterministic, which means their stack frames can be simpler and can be discarded on success.
- Knowing which variables are bound when and what their types are allows us to generate simple indexing code that is fast yet compact. The specialization of data representation for each type means that many indexing operations can use dense jump tables or (when indexing tables of facts) simple array lookups. If the type of a variable allows only one functor (e.g. the type pair, and other tuple constructors), then indexing is a null operation.

- We generate separate code for each mode of a predicate. There is no program size explosion because few predicates (fewer than 5% in our experience) have more than one mode. The code we emit for each mode is small because it need not contain any code either to test the initial state of instantiation of any argument or to handle any initial state of instantiation other than the one given in the mode declaration.

Some Prolog implementations (e.g. Aquarius [10], Parma [9]) perform extensive analysis of the program in order to obtain some of these benefits. This can be a very difficult task for several reasons.

- Analyses must usually assume that the program is perfect, even though we all know that during development and even afterwards many programs are far from perfect. For example, if the analyzer of a Prolog compiler discovers that a given argument of a given predicate is bound at the time of almost all calls, but not at all calls, it must assume that this is what the programmer wanted, even though it may be an error.

- Some optimizations require the analysis to be perfect. For example, a Prolog system cannot use different data representations for different types unless the program is well typed (and many Prolog programs aren't) and the compiler knows the type of *every* variable in the program.

- Some predicates are used in more than one mode, and analyzers can do a better job if they keep separate information about each mode for purposes such as indexing. Many of these modes differ only in trivial ways. Analyzers must choose between treating these separately, which increases analysis time, or treating them together, which reduces analysis accuracy. For example, if X and Y are known to be ground, should the analyzer treat append(X, Y, [a, _]) and append(X, Y, [_, b]) separately or together?

- Analyzers must have a limit on the complexity of their abstract domains if they wish to guarantee termination of the analysis. These limits may force them to ignore useful information. For example, Aquarius may know that a given argument of a given predicate is a list, or that it is an integer, but it cannot know that it is a list of integers, because its abstract domain isn't that deep. This seriously reduces the performance of Aquarius on e.g. the crypt benchmark.

Unlike a Prolog code generator, the Mercury code generator may assume that the programs it deals with are perfect in certain important ways (i.e. the declared or inferred types, modes, and determinisms are correct), because the semantic analyzer will reject programs that are not. It may assume that the semantic analyzer has given it perfect information, because the analyzer must reject the program if it cannot figure out e.g. the type of a variable. It has guidance from the programmer about which modes of which predicates should be treated separately, and it knows from declared or inferred type and maybe mode information just how deep (and what shape) an abstract domain needs to be in order to handle a particular piece of code.

6. Current and future work

Browsing	We have a team of students working on a graphical tool for browsing Mercury programs.

Debugging	We have another team of students working on a debugger for Mercury programs, and we have recently hired a research assistant to work on the same project.
Optimization	We are working to implement more optimizations, including compile-time garbage collection and structure reuse as well as program transformations to promote tail recursion. We are also working to extend existing optimizations such as inlining that at present only work within a module to work across modules as well.
Execution models	We are continuing to refine our existing execution model e.g. by investigating new parameter passing conventions. We are also working a new execution model based on the translation of Mercury to imperative languages in a much more natural manner (e.g. with a one-to-one correspondence between Mercury variables and imperative language variables.) This model resembles the work of Nilsson [7].
Parallelism	We are investigating the stream AND-parallel implementation of Mercury. This involves generalizing the mode system as well as the modification of the execution algorithm and runtime system.
Constraints	We have already linked the CLP(R) constraint solver to Mercury via the Mercury foreign function interface. Later this year we intend to start looking into the closer integration of the two systems.
Deductive databases	We have just started work on the integration of database features such as transactions into Mercury. Mercury is intended to replace Aditi-Prolog as the user language of Aditi, a deductive database system under development in our department.

7. Availability

The Mercury implementation is under the General Public License of the Free Software Foundation. Both the implementation and the papers describing it are available on the World Wide Web at the URL http://www.cs.mu.oz.au/mercury.

References

[1] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807–820, 1988.

[2] T. Conway, F. Henderson, and Z. Somogyi. Code generation for Mercury. In *Proceedings of the Twelfth International Conference on Logic Programming*, pages 242–256, Portland, Oregon, December 1995.

[3] S. Graham, P. Kessler, and M. McKusick. Gprof: a call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120–126, Boston, Massacusetts, June 1982.

[4] F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, February 1996.

[5] F. Henderson, T. Conway, Z. Somogyi, and P. Ross. The Mercury user's guide. Technical Report 96/11, Department of Computer Science, University of Melbourne, Melbourne, Australia, February 1996.

[6] F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the Nineteenth Australasian Computer Science Conference*, pages 337–346, Melbourne, Victoria, February 1996.

[7] J. F. Nilsson. On the compilation of a domain-based Prolog. In *Proceedings of the Ninth IFIP Congress*, pages 293–298, Paris, France, 1983.

[8] 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.

[9] A. Taylor. LIPS on a MIPS: results from a Prolog compiler for a RISC. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 174–185, Jerusalem, Israel, June 1990.

[10] P. Van Roy and A. Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1):54–68, January 1992.

[11] D. H. D. Warren. Higher-order extensions to Prolog: are they needed? In J. E. Hayes, D. Michie, and Y.-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood, 1982.