Logic programming for the real world

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

Richard O'Keefe
ok@cs.rmit.edu.au
Department of Computer Science, Royal Melbourne Institute of Technology
Melbourne, 3001 Victoria, Australia

1 The objective

Logic programming languages have been around for more than twenty years. Despite their theoretical advantages over conventional programming languages, they have not had significant impact on the computer industry. Most of the 500 applications on the Prolog 1000 list[1] are small by industrial standards (fewer than ten exceed 100,000 lines of code), and very few have a user base exceeding a hundred people. We believe that changing this situation and establishing logic programming as a major paradigm in the non-academic world is an appropriate objective for a successor to Prolog.

Convincing programmers to switch to a new language from the languages they are currently using requires the new language to meet two criteria. First, the new language must be demonstrably superior to the user's current language in at least one aspect, and this superiority must be realizable as an economic advantage. Second, the new language must not lack any of the qualities that the programmer values in his or her current language.

To meet the first criterion, a new logic programming language need only realize the advantages that have been promised by logic programming advocates since the inception of the field. These advantages include the following.

- Logic programming languages are very expressive, high level languages that allow programmers to concentrate on what must be done, rather than how it should be done. By automating some of their more mundane tasks, logic programming allows programmers to spend their time on more productive endeavors. Automatic memory management, automatic management of searches, and transparent interfaces to relational and deductive databases are three examples. (The last two are unique to logic programming; they are not shared by functional languages.)
- Logic programming languages have much more useful formal semantics. This makes the automatic analysis and transformation of logic programs much simpler and therefore much more effective than the analysis and transformation of imperative programs. In theory, this should allow programmers to worry less about efficiency, which again makes them more productive.
- The semantics of logic programming languages need not prescribe an order of evaluation, which makes it much easier for a compiler to emit parallel code. By comparison, the automatic parallelization of code written in an imperative language requires extensive non-local analysis of the program, and for many programs acceptable results require significant user intervention [2].
- Logic programming languages have the potential to make debugging much easier, since one can implement declarative debuggers for such languages [12, 16, 19]. Realization of this potential would not only improve productivity, it would also eliminate a large fraction of the uncertainty in the schedules of major projects.

The realization of these advantages in full requires a pure language. A language such as Gödel [9], which allows the bulk of a program to be pure but still requires the use of impure features for I/O, may realize some of these advantages, but it is doubtful whether that is enough in the marketplace.

The qualities that a language must have to meet the second criterion include the following.

Support for the creation of reliable programs. As software systems become bigger and more complex, it becomes more difficult to foresee all the interactions between the pieces of the system, and the more likely

[1] ftp://src.doc.ic.ac.uk/packages/prolog-progs-db/prolog1000.v1.gz

that these interactions will have bugs. A language that prevents entire classes of bugs by catching them at compile time increases not only programmer productivity (the bugs don't have to be chased down one by one), but also the reliability of the final product (since no realistic testing process is 100% effective).

Support for the creation of efficient programs. Hardware is getting faster very quickly, but the sizes of the tasks that people expect computers to perform are increasing fast as well, and fast hardware continues to carry a hefty price premium over slow hardware. Other factors being equal, a fast software product will therefore beat a slow software product in the market. This implies that for a new language to be accepted, it must have an implementation that yields speeds competitive with the available implementations of the alternative languages.

Support for programming in teams. Most commercially important software systems are too big to be created in a reasonable time by a single programmer working alone. Working in teams is a fact of life. Team projects require languages that support modularity and information hiding, because without these programmers cannot effectively isolate themselves from the effects of changes made by other programmers, and therefore must spend much of their time not on forward progress but on reacting to changes in parts of the program out of their control.

Support for program maintenance. Any program with a sizable active user base needs maintenance, at least to adapt the program to new environments and new requirements. The more successful a software product is, the higher the fraction of its life-cycle cost that is devoted to maintenance. Since programmers often need to maintain code written by someone else, the language should make it easy for programmers to read and understand each other's work. It is also a significant advantage if the language helps maintainers to preserve the consistency of the system by pointing out which other parts of the system must be updated after a change.

Support for accessing external databases. Many companies depend absolutely on access to their databases. Such companies will not adapt a new language unless that language provides a convenient method for access to the data held in existing repositories.

2 The proposal

We propose that a successor to Prolog be a pure language with a strong type system, a strong mode system, a strong determinism system, support for higher order programming, explicit uniqueness information and a modern module system. The next few subsections justify our choice of these features by describing their contributions to our objectives.

We have been been working along these lines for several years. We have incorporated the results of our researches in the language Mercury. The publically available Mercury implementation [2] proves the feasibility of our proposal. The claims we make in the next few subsections are derived from our experience with writing 82,000 lines of Mercury code. We will therefore use Mercury in the rest of this paper as an example.

However, Mercury is by no means the only possible language design that fits within our proposal's framework.

2.1 A strong type system

We believe that the new language should have a strong type system that is very similar to the type system proposed by Mycroft and O'Keefe [14], which in turn was inspired by the type systems of functional languages such as Hope and ML, which are also the basis of the type system of Haskell. These type systems are based on the idea of discriminated union types, and support parametric polymorphism. The following examples shows the Mercury syntax for declaring new types and for declaring the type signatures of predicates.

```
:- type bintree(K,V) ---> empty
                     ;    tree(K, V, bintree(K,V), bintree(K,V)).

:- pred search(bintree(K,V), K, V).
```

Mercury supports heterogeneous collections through the universal type "univ". Values of any type can be converted to type univ and back again, with the back conversion being checked at runtime to ensure type safety. Mercury also supports a standard type "term" with similar operations, the distinction being that values of type "term" can

[2] http://www.cs.mu.oz.au/~zs/mercury.html

be manipulated by Mercury programs in the standard way, whereas terms of type "univ" are black boxes. On the other hand, conversions to and from type "term" require a representation change, while the conversions involving type "univ" do not.

An obvious and desirable extension of the current Mercury type system would be the support of constrained genericity through a mechanism analogous to the type classes of Haskell. One could also argue in favor of support for subtypes and undiscriminated union types.

The reliability advantages of strong type systems have long been recognized [1]. All the major imperative languages that aim to support the creation of reliable programs have had strong type systems for quite a while now. Ada and Eiffel are two languages that stand out in this respect. Even C was conceived as a typed extension of B; it can be used as a (mostly) strongly typed language if one turns on all compiler warnings and refrains from using casts.

By allowing term representations to be specialized, strong types contribute to efficient implementations. Type declarations help program readers to understand the intended uses of predicates, and therefore make both team programming and program maintenance easier. Having a type system at least as expressive as the type systems of SQL and of other data definition languages also simplifies interfacing to external database systems.

2.2 A strong mode system

There are two ways to view modes, the descriptive view and the prescriptive view. In the descriptive view, modes describe the instantiation states of the arguments of a predicate when execution enters the predicate and possibly when execution leaves the predicate. In the prescriptive view, modes assign responsibility for the instantiation of variables to the various subgoals of a predicate definition. Unlike descriptive modes, prescriptive modes can affect the computation rule. The original mode declarations of DEC-10 Prolog take the first view; the mode declarations of Parlog [3] and to a certain extent the delay declarations of Gödel [9], NU-Prolog and other coroutining Prologs take the second view.

We believe that the second, prescriptive view is more useful. Programmers write predicates to compute values from other values, and this view of modes captures this important aspect of the programmer's intention. Writing down mode declarations therefore helps readers of the program.

The delay declarations of Gödel and similar systems say that it is the responsibility of the caller to get the arguments to a certain state of instantiation, but do not assign any responsibility to the predicate itself. Since no predicate ever takes sole responsibility for instantiating anything, there is no way to be sure that the input requirements of the various calls in the program are met, which means the program may flounder at runtime. We prefer a mode system that allows the compiler to guarantee that it will detect all situations in which a variable has consumers but no producers. and can therefore guarantee that the program will not flounder.

In Mercury, every predicate must have one or more mode declarations. Each mode declaration states that if the predicate is called with its arguments in a given state of instantiation, then on success those arguments will be in another given state of instantiation. Achieving the initial instantiation state is the responsibility of the caller; achieving the final instantiation state is the responsibility of this predicate. The most common instantiation states are free and ground. The mode "in" maps ground to ground, while the mode "out" maps free to ground. These two modes suffice for the vast majority of predicates in many real programs (see e.g. [6, 11], and compare the in/out annotations on the parameters of procedures in Ada). Mercury, however, also allows programmers to define new instantiation states based on the structures of types. These are required for operations on partially instantiated data structures, e.g. using one predicate to create a list skeleton and another to fill in the elements of the list[3].

A mode declaration such as

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

states that it is the caller's responsibility to ground the first two arguments while it is the responsibility of this predicate to ground the last argument. It is the programmer's job to provide this design-level information. It is the compiler's job to reorder conjunctions containing calls to this predicate as necessary to ensure that the caller does ground the first two arguments, and to make sure that `search` will in fact ground its third argument. If the compiler fails in either task, it must reject the program. This implies that a Mercury implementation can omit the occur check without losing soundness, since failures of the occur check can happen only in the presence of a mode error.

[3] The current Mercury compiler has a limitation that causes it to reject any program that tries to fill in a partially instantiated data structure. This limitation will be removed in the future.

Mercury allows a predicate to have more than one mode, which each mode having a separate implementation. (This does *not* cause problems with code size in practice.) Because the body of the predicate may be reordered differently for each mode, Mercury has considerably more support for reversible predicates than Prolog does.

A strong mode system imposes discipline on the programmer's use of logical variables, which are Prolog's equivalent of pointers in terms of how easy it is to make mistakes in their use [1, 6]. The compile-time enforcement of this discipline effectively guarantees the absence of the class of errors that corresponds to the use of uninitialized variables in imperative programs. (Some modern imperative languages -e.g. C++, Eiffel, Ada, and the language described in [24]- also have mechanisms to reduce the incidence of this class of errors.) Modes contribute further to reliability by making declarative debugging more focused and therefore more productive; if the user knows which part of an answer is incorrect, then it makes sense to look for bugs in its producer, and if the producer turns out to be correct, then in the predicates that generated *its* inputs [16].

A strong mode system also provides information vital to an efficient implementation. Since each mode of a predicate has its own code, that code can be much more specialized: unifications can be implemented as simple tests and assignments, parameter passing can be streamlined, and indexing is simpler. Since aliasing is strictly limited, dereferencing can be totally avoided. The compiler's complete knowledge of the instantiation state of every variable at every point in the program means that unbound variables need no storage or initialization, and hence there is no need for a trail. This knowledge also allows the compiler to ensure the safety of negations and if-then-elses without any runtime checks. If Mercury is ever extended with subtypes, mode information will allow the compiler to insert subtype checking code in the minimal set of places. And to close this long list of advantages, we note that many high-level optimizations (e.g. [10, 13, 23]) are based on the availability of mode information.

Since most database systems do not allow the storage of variables, a strong mode system also simplifies interfacing to external databases.

2.3 A strong determinism system

One can divide modes of predicates into four classes based on the number of solutions different calls to them can have. The four classes are those that have exactly one solution, those that have at mode one solution, those that have at least one solution, and those that have any number of solutions. If programmers provide declarations regarding this aspect of the intended behavior of their predicates, then the compiler can exploit the redundancy this introduces to detect errors in the program. Like type and mode declarations, such determinism declarations directly reflect the intent of the original programmer, and thus make both team programming and program maintenance easier.

Mercury attaches determinism declarations to mode declarations, like this:

```
:- mode append(in, in, out) is det.     % exactly one solution
:- mode append(out, out, in) is multi.  % at least one solution
```

The compiler analyses the bodies of predicates to check that their determinism declarations are correct. Although this problem is undecidable in general, several fast approximate (but conservative) solutions are known [4, 20, 21], and they work well in practice. Our determinism analysis algorithm [22] is especially designed to take advantage of the extra information provided by the strong type and mode systems. This algorithm produces good error messages. For example, if one omits the base case from the definition of append, the resulting messages are:

```
append.m:8: In ‘append(in, in, out)’:
append.m:8:   Error: determinism declaration not satisfied.
append.m:8:   Declared ‘det’, inferred ‘semidet’.
append.m:14:  in argument 1 of clause head:
append.m:14:  unification of ‘HeadVar__1’ and ‘[X | Xs]’ can fail.
append.m:9: In ‘append(out, out, in)’:
append.m:9:   Error: determinism declaration not satisfied.
append.m:9:   Declared ‘multi’, inferred ‘nondet’.
append.m:14:  in argument 3 of clause head:
append.m:14:  unification of ‘HeadVar__3’ and ‘[X | Zs]’ can fail.
```

Messages like this can prevent one of the most frustrating experiences of a Prolog programmer, which occurs when a large computation that was intended to succeed fails instead, since the bug could be anywhere in the computation. Mercury's determinism system will pinpoint the bug at compile time. This capability is particularly useful during program maintenance. If a type is modified with the addition of a new function symbol, one must also modify all the predicates that perform a case analysis on that type. A strong determinism system makes this trivial; one just compiles the program and fixes the determinism errors reported by the compiler. These reports are very specific:

```
draw.m:42:  The switch on ‘Shape’ does not cover ‘circle’/2.
```

In practice, a very high percentage of all predicates are deterministic or semideterministic. An implementation can exploit this information by specializing the execution algorithm of these predicates, for example by always discarding the stack frames of such predicates as soon as they succeed. Interfaces to database systems can exploit determinism information to avoid runtime checks for the maintenance of functional dependencies.

2.4 Higher-order programming

Several decades of experience with functional programming languages has shown that higher-order functions are an extremely powerful and expressive programming construct. They make it easy to concisely express the essential part of an algorithm, while abstracting away the details that vary between different uses of that algorithm. By so doing, they allow programmers to avoid unnecessary dependencies, improve modularity, and increase code reuse.

Furthermore, experience with strongly typed logic programming languages such as Gödel and Turbo Prolog which lack higher-order facilities has shown those languages to be lacking in expressiveness; for example, in those languages it is not possible to write a generalized ‘sort’ predicate.

We therefore consider support for higher-order programming to be an essential feature of any potential successor to Prolog.

Mercury provides such support; higher-order predicate terms (closures) can be constructed using predicate names, lambda expressions, and/or currying, and they can be passed to other predicates and stored in data structures. Mercury’s type, mode, and determinism systems contain support for this, as shown by the following example:

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

2.5 Explicit uniqueness information

There is at least one aspect in which conventional imperative languages are undoubtably superior to existing logic programming languages: they provide a simple and efficient mechanism for the destructive modification of data structures. This low-level notion of destructive update cannot be expressed directly in predicate calculus.

Destructive update is critical for many algorithms; many data structures (for example, arrays and hash tables) are almost useless without it. For this reason, it is essential that any language which aims to have widespread use in industry provide some facility for destructive update.

Prolog provides the infamous `assert/1` and `retract/1` predicates, but they are cumbersome, very inefficient, and they destroy the declarative semantics. Other alternatives such as `setarg/3` and BinProlog’s blackboard variables are less cumbersome, but they still have a significant efficiency overhead[4], and they still destroy the declarative semantics.

Of course logic provides a perfectly adequate way of expressing modifications to data structures — you simply specify the relation between the old state and the new state. The problem is that even though in many situations this can safely be implemented by destructively updating the old state, compilers do not have enough information available to determine when it is safe to perform this optimization.

In theory, incremental global analysis might be able to provide uniqueness information, but in practice this turns out to be very difficult. Even if it weren’t, one must also consider that global analysis by itself can never *guarantee*

[4] In the case of `setarg/3`, this overhead comes from (1) the overhead of a procedure call; (2) the overhead of checking that the argument number is in range; (3) the overhead of a trail check; and in some cases (4) the overhead of unnecessary trailing.

that the compiler will be able to perform the optimization; even if the analysis is perfect, the program may not be. Uniqueness is quite fragile: a small change in one part of a program can affect uniqueness in many other parts of the program.

For this reason, Mercury's mode system provides "unique" modes [7], which allow a programmer to declare the necessary uniqueness information. Like all of Mercury's mode declaration, unique modes are checked by the compiler, which will reject them if it cannot prove that they are correct. The programmer can thus be sure that the compiler has the information needed to perform destructive update optimization. We believe that any language which aims to compete with imperative languages must provide a mechanism to achieve this effect.

2.6 A modern module system

The current Mercury module system is a simple copy of the module systems of imperative languages such as Modula-2. Each module has an interface section containing the declarations of the entities (usually types and predicates but possibly also instantiation states and modes) exported from the module, and an implementation section containing the definitions of the exported entities and the declarations and definitions of entities used only within the module. Types whose name is exported but whose definition isn't represent abstract data types. To get access to the entities exported by module A, module B must explicitly import module A, and module B cannot get access in any way to the entities private to module A.

There are several obvious potential improvements to this arrangement. First, one may want a module to be able to import only some of the entities exported by another module. Second, it may be useful to refine the current two-level visibility hierarchy (private: this module only, public: all modules) into a three level hierarchy which divides the other modules into two classes, those regarded as subcontractors of this module (e.g. friends in C++) and those regarded as clients, with subcontractors being given more access than clients. Third, it would be desirable to make the module name space hierarchical rather than flat.

3 Research opportunities

There are a large number of research questions that can be explored in the context of a language with the features that we have proposed. These include at least the following.

- What algorithms are required for the automatic accurate inference of type, mode and determinism information?
- What are the best ways to integrate constraint solving capabilities into the framework that we have proposed? (For example, would it be possible to adopt the approach taken in Trilogy [25]?)
- Can one integrate exception handling into this framework, and if so, how? What sort of semantics would be appropriate for a logic programming language that supports exceptions?
- What extensions to the mode system would be necessary to express parallel algorithms? What are the best ways to exploit the information provided by the type, mode, and determinism systems to reduce the overheads of a parallel implementation?
- What other useful idioms are not supported by the basic type, mode, and determinism systems that we have proposed? What extensions of these systems would be required to support these idioms? What would be the effect of these extensions on the compiler's checking and/or inference algorithms, on the efficiency of the generated code, and on programmers?
- Are languages of the kind we propose appropriate for writing operating systems?

4 Discussion

We have justified each proposed language characteristic on its own merits, but it is striking that most of them can also be seen as requirements for a fully declarative logic programming language. The argument goes like this. A pure language requires a pure way to do I/O. One can provide a basis for this by viewing I/O operations as transformations from one state of the world to another. The actual implementation of this view requires a way to ensure that the old state of the world is never referred to again. Explicit uniqueness information serves this purpose on forward execution, just as linear types do in a functional framework [26]. (The same arguments apply

system	speed
SWI-Prolog	1.00
NU-Prolog	1.95
wamcc	3.14
Quintus Prolog	3.74
SICStus Prolog fastcode	7.41
Aquarius Prolog	19.00
Mercury	36.52

Table 1: Benchmark speed ratios on a SPARCserver 1000

to other approaches to declarative I/O, such as those based on streams [8, 18, 21] and on monads [17].) Making sure that the old state is not referred to again even on backtracking requires determinism information (which is of course available for free in functional languages and in languages based on compulsory committed choice logic programming). Uniqueness and determinism systems require mode information, and the mode system in turn depends on type information for sufficient expressiveness.

Besides I/O, the most important use of the impure features of Prolog is to reduce the memory consumption of a program and increase its speed. However, our work shows that impure features are not required for either of these objectives. Even though the Mercury implementation does not yet exploit uniqueness information for compile-time garbage collection, it is already the fastest logic programming language implementation by a large margin. We have tested the speed of several Prolog systems on a collection of standard logic programming benchmark programs. 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, and about ten times the speed of Quintus Prolog and wamcc. Furthermore, the Mercury implementation generates relatively small executables. The Mercury compiler and the standard Mercury library, which are written in the intersection of Mercury, NU-Prolog and SICStus Prolog, have 74,000 lines of source code totalling about 2.4 Mb. The executable generated by Mercury with default optimization is 2.2 Mb; with full optimization it is 1.85 Mb (both statically linked). NU-Prolog generates a 1.9 Mb save file. The save file for SICStus compactcode is 4.4 Mb, while for SICStus fastcode it is 7.1 Mb.

We expect that a language organized along the lines we have proposed in this paper will be suitable for most industrial applications, but not for all applications for which people are using Prolog at the moment. Our approach trades some kinds of flexibility for improved reliability and performance. Historically this has usually been a good tradeoff [5], but some applications (or at least the currently standard approaches to them) require some of the flexibility we trade away. The main example of this is programs that use Prolog as a constraint solver over the Herbrand universe. (Constraint programs cannot be well moded in general, because whether or not a constraint solver can determine the value of a given variable from a set of equations is in many cases data dependent.) However, most Prolog programs do not need any constraint solving capabilities, and even CLP programs usually have significant portions that can be well moded.

Another issue is that pure, strongly typed and moded languages require a different programming style than Prolog, and therefore many of their idioms will be different from the idioms of Prolog. For example, in some cases Mercury enforces practices that in Prolog would merely be considered good style. Consider "defaulty" representations, in which a data structure has a catchall case. Such representations have long been recognized as bad style in Prolog [15] because they lead either to space leaks due to choice points being left around or to programs sprinkled with cuts. In Mercury defaulty representations are simply not allowed.

Another example is meta-programming. The standard Prolog meta-interpreter is not well typed or not well moded, and it requires impure code to test whether a user-level term represents a variable or not. In any case, as the work of John Lloyd over the years has demonstrated, writing meta-programs using the standard non-ground representation is unsound, because it conflates user-level and meta-level variables. If one chooses to use the ground representation, the standard features of Mercury are entirely sufficient for the convenient creation of meta-programs.

Different types of programming require different languages; no matter what the successor to Prolog will turn out to be, Prolog will remain better suited for some tasks (a small number of small tasks :-). Even if this were not the case, we would expect a significant number of logic programming practitioners to continue using Prolog and to continue to write untyped, unmoded programs. just as a large number of "functional" programmers continue to program in Lisp and its dialects. Hoping to convert *all* Prolog programmers is not an attainable goal.

References

[1] M. Bruynooghe. Adding redundancy to obtain more reliable and more readable Prolog programs. In *Proceedings of the First International Logic Programming Conference*, pages 129–133, Marseille, France, September 1982.

[2] D. Cann. Retire Fortran? A debate rekindled. Technical report, Lawrence Livermore National Laboratory, ftp://sisal.llnl.gov/pub/sisal/publications.dir/retire.ps.Z.

[3] K. Clark and S. Gregory. Parlog: a parallel logic programming language. Technical Report DOC 83/5, Department of Computing, Imperial College of Science and Technology, London, England, May 1983.

[4] 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.

[5] P. J. Denning. Sacrificing the calf of flexibility on the altar of reliability. In *Proceedings of the Second International Conference on Software Engineering*, pages 384–386, San Francisco, October 1976.

[6] W. Drabent. Do logic programs resemble programs in conventional languages? In *Proceedings of the Fourth IEEE Symposium on Logic Programming*, pages 389–396, San Francisco, California, August 1987.

[7] F. J. Henderson. Strong modes can change the world! Technical Report 93/25, Department of Computer Science, University of Melbourne, Melbourne, Australia, October 1993.

[8] P. Henderson. Purely functional operating systems. In J. Darlington, P. Henderson, and D. Turner, editors, *Functional programming and its applications*, pages 177–192. Cambridge University Press, 1982.

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

[10] D. B. Kemp, K. Ramamohanarao, I. Balbin, and K. Meenakshi. Propagating constraints in recursive deductive databases. In *Proceedings of the First North American Conference on Logic Programming*, pages 981–998, Cleveland, Ohio, October 1989.

[11] F. Kluzniak. Type synthesis for ground Prolog. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 788–816, Melbourne, Australia, June 1987.

[12] J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

[13] A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of shared data structures for compile–time garbage collection in logic programs. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 747–762, Jerusalem, Israel, June 1990.

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

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

[16] L. M. Pereira. Rational debugging in logic programming. In *Proceedings of the Third International Conference on Logic Programming*, pages 203–210, London, England, June 1986.

[17] S. Peyton-Jones and P. Wadler. Imperative functional programming. *Proceedings of the Twentieth Conference on the Principles of Programming Languages*, January 1993.

[18] E. Shapiro. Systems programming in Concurrent Prolog. In M. van Caneghem and D. H. Warren, editors, *Logic programming and its applications*, pages 50–74. Ablex, 1986.

[19] E. Y. Shapiro. *Algorithm program debugging.* MIT Press, 1983.

[20] 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.

[21] Z. Somogyi. Stability of logic programs: how to connect don't-know nondeterministic logic programs to the outside world. Technical Report 87/11, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1987.

[22] Z. Somogyi, F. Henderson, and T. Conway. Determinism analysis in the Mercury compiler. Technical Report 95/25, Department of Computer Science, University of Melbourne, Melbourne, Australia, August 1995.

[23] Z. Somogyi, K. Ramamohanarao, and J. Vaghani. A backtracking algorithm for the stream AND-parallel execution of logic programs. *International Journal of Parallel Programming*, 17(3):207–257, June 1988.

[24] R. E. Strom and S. Yemeni. Typestate: a programming language concept for enhancing software reliability. *Communications of the ACM*, pages 157–171, January 1986.

[25] P. J. Voda. Types of Trilogy. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 580–589, Seattle, Washington, August 1988.

[26] P. Wadler. Linear types can change the world! In *Proceedings of the IFIP TC2 Conference on Programming Concepts and Methods*, pages 547–566, April 1990.