PARMA - BRIDGING THE PERFORMANCE GAP BETWEEN IMPERATIVE AND LOGIC PROGRAMMING

ANDREW TAYLOR

▷ Parma is an experimental high-performance Prolog compiler for the MIPS RISC architecture [?]. It was the first logic programming implementation to obtain performance comparable to imperative languages. It depends heavily on a global static analysis phase based on abstract interpretation. This paper describes the important components of Parma's implementation and discusses performance results including analysis of the incremental benefits of some components of the compiler. ◁

1. INTRODUCTION

Parma is an experimental Prolog compiler for the MIPS architecture. Its performance exceeds previous Prolog implementations by an order of magnitude. This paper will describe the important innovations made in Parma to achieve this performance. An overview of logic programming implementation research, including Parma's place in it, can be found in Van Roy's excellent history [?]. Earlier versions of this work were presented in [?] and [?]. An extensive description of Parma can be found in [?].

2. ANALYSIS

By the far the most crucial component in Parma's performance is a global analysis phase which examines the program as a whole to gather information for use in the compilation of the program.

Address correspondence to Department of Computer Science and Engineering, University of New South Wales, Sydney 2052 Australia (e-mail: andrewt@cse.unsw.edu.au)

Peter Van Roy, concurrently and independently, achieved comparable performance using similar methods [?]

Parma is named for *Macropus parma* and for *Parma microlepis*. It can also be interpreted as an acronym - Prolog for A Risc Machine Architecture.

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

The paradigm for much of the work on analysis of Prolog including Parma's global analysis phase is a technique named abstract interpretation. It was first applied to imperative languages [?]. Parma's use of abstract interpretation was inspired by Mellish's seminal work [?]. Its analysis phase is implemented by roughly 4,500 lines of SICStus Prolog [?]. The analysis assumes the entry point to the program is a call to the predicate *main*/0.

Parma's application of abstract interpretation incorporated no theoretical advances but it did demonstrate that the use of abstract interpretation to gather detailed information was feasible in practice. The control of abstract interpretation was done in a relatively simple and ad-hoc manner. More efficient and elegant methods have since been developed so the algorithm Parma employs is only of historical interest. It is is described in [?].

Much more effort was devoted to the design of Parma's abstract domain. It included an important innovation - representation of implementation artifacts such as reference chains [?]. The insight that abstract interpretation could gather information about operational characteristics of programs, such as dereferencing and trailing, was probably the most important aspect of this work. Previous Prolog implementations had accepted the cost of such operations as inherent in the language. The information gathered by Parma's analysis phase allows the majority of such operations to be removed.

The nature of Parma's abstract domain is not only governed by the demand for information of the subsequent compilation phases, some features of the abstract domain are necessary to prevent loss of information during analysis.

Here is a terse description of the abstract domain Parma employs:

free(may_alias, must_alias, is_aliased, trail) If two free variables have the same *must_alias* values then they must be aliased. If two free variable can be aliased then they must have the same *may_alias* values. Both *must_alias* and *may_alias* values are implemented as free Prolog variables in Parma. *is_aliased* is false if the free variable is not aliased to any other free variable. If *trail* is false the variable does not need trailing when bound.

unknown(may_alias) Any term whose free variables may only be aliased to those indicated by *may_alias*.

bound(may_alias) As **unknown** but cannot be a free variable.

ground Any term that contains no free variables.

constant, number, float, integer, atom and [] Constants and sub-divisions of the constants.

term(functor($type_1$**, ...** $type_n$**))** Any term of form *functor*/*n* whose arguments are in $type_1$, ... $type_n$, respectively where $type_1$, ... $type_n$ are members of this abstract domain.

list(*car_type***,** *cdr_type***)** Any term in *cdr_type* or one or more chained list ('.'/2) cells all with the first argument of each in *car_type* and the second argument of the last in *cdr_type*.

Parma's abstract domain includes a compound symbol which represents a structure of a particular functor and contains symbols describing the possible arguments

of the structure. This allows precision to be maintained where structures are used to aggregate data analogously to Pascal's *record* data type.

The unrestricted introduction of this compound symbol would make the abstract domain a lattice of infinite depth. Parma avoids this and hence simplifies control of abstract interpretation by restricting the nesting of compound symbols to less than a constant bound. A nesting limit of four was sufficient to maintain precision in all the small to medium-sized programs we examined. It is likely that it is not sufficient in many programs with complex data structures.

A more general handling of recursive types was too ambitious at the time when Parma's analysis phase was implemented. Simply establishing the practicality of utilising global analysis in a Prolog compiler was a useful result when this work was undertaken. No doubt similar work now would take a more general approach.

Many programs implement recursive data types such as lists and binary trees. These cannot be represented precisely in Parma's abstract domain. This is unfortunate as the parts of the programs which manipulate these data structures are often those where most execution time is spent and hence the parts where good compilation is most desirable.

Lists, in particular, pervade Prolog programs making their efficient handling vital so an ad hoc solution was adopted. The symbol *list*(*car*, *cdr*) represents the symbols in the infinite sequence: *cdr*, '.'(*car*, *cdr*) , '.'(*car*, '.'(*car*, *cdr*)) , . . .

For example, the symbol *list*(*integer*, []) represents precisely all nil-terminated flat lists of integers. The same nesting restriction that applies to *structure* symbols applies to the *list* symbols ensuring that the abstract domain remains a lattice of finite depth.

Earlier versions of Parma omitted the *cdr* argument of the *list* symbol. It was implicitly assumed to be '[]'. This did not allow the exact representation of difference lists [**?**] resulting in significant information loss when they are employed. Difference lists are not prevalent in benchmark programs but skilled Prolog programmers make significant use of them. It can be expected that not only will they appear in real programs but they will appear in precisely those places where efficiency is important. The inclusion of the *cdr* argument not only allowed exact handling of difference lists but also slightly simplified abstract interpretation.

The combination of aliasing and the *list* compound symbol is awkward because the *car* argument symbol possibly corresponds to multiple sub-terms. There are several possible choices for the meanings of the aliasing information in this context. Parma made two simplifying restrictions. The *must_alias* argument of *free* symbols is limited in scope to within the *car* argument. The *may_alias* argument of *free*, *unknown* and *bound* symbols has unlimited scope. As a result there is no way to precisely represent a list of variables that must be aliased to each other. It is also not possible to precisely represent a list of variables that cannot be aliased to each other.

3. REPRESENTATION

The paradigm of the abstract machine has dominated research in Prolog implementation since Warren produced the WAM [**?**]. The simplicity and elegance of the WAM, which are great virtues in other respects, become a straight-jacket for implementations wishing to exploit a conventional machine to its fullest. For example, the dereferencing operation of the WAM model is expensive to implement

on a conventional machine and can often be avoided but it is integral to the instructions of the WAM. The WAM hides operations basic to the efficiency of an implementation.

Parma makes no use of the WAM's instruction set or any similar abstract machine. For the purposes of this paper it can be assumed that Parma uses the data representations of the WAM. In the later stages of Parma's development an experimental representation for variable reference chains was employed. It was eventually discovered this representation offered no advantages over that of the WAM. This does not affect the results presented later.

One important exception is that Parma treats specially predicate argument positions known from the analysis phase to always be an unaliased unbound variable on entry and to always be bound when the predicate exits. In these cases a reference to an uninitialised word is passed by the caller allowing the cost of initialisation to be avoided. Van Roy and Despain[?] extend this by allowing such values to be returned in a register in some circumstances.

Considerable care was taken to minimise the cost of tag handling operations. Like the WAM, tags are placed in the least significant bits of words. This allows much of the tag manipulation operation around arithmetic operations to be folded with the arithmetic [?]. It also means the tag need not be removed when obtaining the *car* or *cdr* of a list cell or an argument of a compound term if the offset used is appropriately adjusted.

Parma uses a similar memory model to the WAM. It does defer as much as possible storing information in choicepoints to reduce the cost of shallow backtracking [?]. Like the WAM predicate arguments are passed through registers.

It is important to note that Parma's data and instruction referencing characteristics are significantly different to the WAM. There have been several studies of Prolog memory referencing characteristics and cache behaviour based on the WAM, such as [?]. These can not be safely applied to Parma.

4. COMPILATION

Parma's compilation phase is implemented by roughly 6,000 lines of SICStus Prolog. The execution times of the algorithms employed in this stage are of little interest. The algorithms, including construction of indexing code, are linear in the size of the code produced or, at least, could be with more careful implementation. Unlike the analysis stage, reducing execution time was not a high priority in the implementation of the compilation stage. As a result Parma's overall execution time, at roughly 2-4 lines compiled per second, is slower than most users would like. The translation from Prolog clauses to MIPS assembly language occurs in ten separate phases.

The first phase transforms clauses to a simpler form but which is still legal Prolog. Basically unifications are moved out of clause heads and the subgoals in the clause body and split into their simplest components. This is convenient both for the analysis phase and the subsequent compilation phases. In the third phase a small set of unambitious transformations are applied at the source level. This mainly involves reordering of some conjunctions. This reordering only involves unifications and calls to some built-in predicates. The primary motivation for this reordering is that it makes generation of the indexing code easier.

Instruction	Description
load R_1, R_2[O]	R_1:= memory[R_2+ O]
store R_1, R_2[O]	memory[R_2+ O] := R_1
Op R_1, R_2, R_3	R_1:= R_2 Op R_3
Op R_1, R_2, I	R_1:= R_2 Op I
	$Op \in \{+, -, *, /, mod, \|, \&, <<, >>\}$
if R_1 $Comp$ R_2*Label*	if condition PC := *Label*
	$Comp \in \{==, !=, >, >=, <, =<\}$
goto R_1	PC := R_1
call *Label*	SUCCESSP := PC + 4, PC := *Label*

R_i is a register. I is an integer or a label. O is an integer.

TABLE 4.1. Intermediate Language Description.

This is followed by the phase which translates the Prolog clauses to low-level instructions of an intermediate language. This is where most of the work of compilation occurs. The translation in this phase requires careful and extensive attention to detail but really contained no startling innovation. The use of the information from the analysis phase is pervasive at this stage. The most important use is in strength reduction of unification operations. General purpose unification is a very expensive operation requiring up to 100 MIPS instructions to implement. The information from the analysis phase often allows a unification to be implemented in a few instructions.

The analysis information is also important in producing efficient code for disjunctions when determinacy can be inferred.

The intermediate language produced by this stage is basically 3-address code [?] reduced to load-store form. This representation is well suited to some of the improving transformations that must be applied to produce good code such as manipulation of load and store instruction offsets.

Parma's load/store intermediate language is described in Table **??**. It has a number of named registers used for representing the execution state and argument passing. In addition an infinite number of general purpose registers are available. In the last phases of compilation the register allocator assigns these to actual machine registers.

The translation of intermediate code to MIPS assembly language is broken into eight phases mainly involving manipulations of the code as basic blocks and single-entry-multiple-exit blocks. Most of the techniques involved are applicable to many languages and thus can be found in standard texts such as [?]. Some care was taken to match the techniques used to the common code patterns generated by Prolog compilation. For example, the frequent manipulation of stacks produces sequences of code with many modifications and uses of the stack pointer. Improving transformations were added specifically for these code sequences. Compilers for conventional imperative languages might not include such transformations because they would be of little importance.

Register allocation is done using a simple algorithm. The abundance of registers on the MIPS meant that for the benchmarks we compiled there was little contention for registers. This is likely not generally true and employment of more sophisticated

allocation algorithms from the literature would be wise for general use. It was found to be important that freed registers were reused in a FIFO manner to produce greater freedom for instruction scheduling by the assembler.

Parma does not explicitly check for stack overflow. It assumes that memory protection feature of the operating system can provide this without the cost of explicit checks. This was not implemented.

It is probably useful to review Parma's operation by following a small example through the phases of compilation. The example we will use is the well-known predicate *append*/3. Here is the program we will assume it is contained in:

```
main  :-
        append([a,b,c], [d,e], X),
        write(X),
        nl.

append([], L, L).
append([X|L1], L2, [X|L3])  :-
        append(L1, L2, L3).
```

Parma's first phase converts the clauses of *append* into a normalised form:

```
append(V1, V2, V3)  :-
        V1 = [],
        V3 = V2.

append(V1, V2, V3)  :-
        V1 = [V4|V5],
        V3 = [V6|V7],
        V6 = V4,
        append(V5, V2, V7).
```

The global analysis phase annotates these normalised clauses with the information discovered during the abstract interpretation of the program as follows:

```
append(V1, V2, V3)  :-
        V1{list(atom, nil)}  = [],
        V3{free(no_trail, no_deref)} = V2{list(atom, nil)}.

append(V1, V2, V3)  :-
        V1{list(atom, nil)} = [V4|V5],
        V3{free(no_trail, no_deref)} = [V6|V7],
        V6{free(no_trail, no_deref)} = V4{atom},
        append(V5, V2, V7).
```

The main compilation phase translates these annotated clauses to Parma's intermediate code. We have rendered the intermediate code in pseudo-C:

```
p_append_3()
{
        g1 = CONST([])
        if (call_1 == g1) goto L2
```

```
                call_3[0] = globalp
                g4 = globalp
                globalp = globalp + 8
                g5 = call_1[O_CAR]
                g4[O_CAR] = g5
                call_3 = g4 + O_CDR
                g6 = call_1
                call_1 = g6[O_CDR]
                goto p_append_3
        L2:
                *call_3 = call_2
                goto successp
        }
```

Subsequent phases apply improving transformations to this intermediate code and perform register allocation. Here is the result:

```
        p_append_3()
        {
                r2 = CONST([])
                if (call_1 == r2) goto L2
        L4:
                r3 = call_1[-2]
                call_1 = call_1[2]
                call_3[0] = globalp
                call_3 = globalp + 2
                globalp[-2] = r3
                globalp = globalp + 8
                if (call_1 != r2) goto L4
        L2:
                *call_3 = call_2
                goto successp
        }
```

The code generator translates intermediate code to the MIPS assembly language:

```
p_append_3:
                li      $2, 5
                beq     $arg1, $2, L2
        L4:     lw      $3, -2($arg1)
                lw      $arg1, 2($arg1)
                sw      $globalp, 0($arg3)
                add     $arg3, $globalp, 2
                sw      $3, -2($globalp)
                add     $globalp, $globalp, 8
                bne     $arg1, $2, L4
        L2:     sw      $arg2, 0($arg3)
                j       $successp
```

The MIPS assembler (not part of Parma) schedules instructions in delay slots and produces the final code for *append*:

```
0x4001d0:       24020005        li      v0,5
0x4001d4:       10820008        beq     a0,v0,0x4001f8
0x4001d8:       00000000        nop
0x4001dc:       8c83fffe        lw      v1,-2(a0)
0x4001e0:       8c840002        lw      a0,2(a0)
0x4001e4:       acd40000        sw      s4,0(a2)
0x4001e8:       22860002        addi    a2,s4,2
0x4001ec:       22940008        addi    s4,s4,8
0x4001f0:       1482fffa        bne     a0,v0,0x4001dc
0x4001f4:       ae83fff6        sw      v1,-10(s4)
0x4001f8:       03e00008        jr      ra
0x4001fc:       acc50000        sw      a1,0(a2)
```

5. PERFORMANCE

The benchmarking of Parma's performance was seriously limited by Parma's experimental nature. The absence of built-in predicates found in other systems excluded many programs. We adopted 17 of the benchmarks used by [?] and included two programs from other sources (*pri2* and *press1*). They range in length from 10 lines to 1138 lines. The results from the smaller programs should be treated with some caution because it is likely that the information from Parma's global analysis phase is atypically precise. They are available by ftp [?].

The hardware used for our measurements was a MIPS RC3230 desktop workstation configured with 72 megabytes of memory. It uses a 25MHz R3000 microprocessor with separate instruction and data caches each of 32 kilobytes. The caches are direct-mapped and write-through. The operating system used was RISC/OS 4.51, a version of Unix. The performance of this system is now modest in current terms. Systems using microprocessors with this architecture offer up to a factor of 10 better integer performance.

The execution times for the global analysis phase for these programs are more than acceptable, most falling in the 20-30 program lines/second range. It was pleasing that there was no indication of worse than linear increase in global analysis time as program size increases.

We have compared Parma with SICStus Prolog (Version 0.6 #14) [?]. SICStus 0.6 was in wide use when Parma was constructed. The times quoted are for SICStus 0.6's execution mode based on a byte-coded abstract instruction set. We also provided comparisons with the native code compilation mode of SICStus 3.0, the current version of SICStus Prolog. These times are for SICStus 3.0's native compilation mode.

The first column of Table ?? contains the benchmark execution times measured for SICStus in seconds. The remaining columns contain execution times for Parma with and without global analysis scaled inversely relative to this time. In other words, the number given is the speed-up factor over SICStus. We have resisted the temptation to provide any form of average of the times because we feel such a number would be misleading.

Parma offers a performance improvement of up to a factor of 66 over SICStus. Although the information from global analysis clearly has great benefits, it can be seen from Parma's performance without global analysis that Parma's compilation to MIPS native-code was responsible for much of the improvement.

	SICStus 0.6 compiled		SICStus 3 native code	Parma no analysis	Parma
program	seconds	scaled	scaled	scaled	scaled
nreverse	0.0056	1	3.9	4.1	34
tak	2.0	1	3.0	23.0	66
qsort	0.0083	1	2.7	7.4	39
pri2	0.019	1	2.4	7.9	19
serialise	0.0069	1	3.5	15.0	30
queens_8	0.041	1	3.4	8.7	29
mu	0.012	1	2.7	7.0	16
zebra	1.3	1	1.9	6.6	10
deriv	0.0019	1	3.2	4.4	24
crypt	0.18	1	2.1	17.0	53
query	0.047	1	3.4	8.0	15
prover	0.015	1	2.4	7.5	15
poly_10	0.68	1	3.9	6.2	17
browse	12.0	1	2.7	8.1	18
press1	0.20	1	1.0	7.9	17
reducer	0.54	1	1.8	5.5	7.9
boyer	8.3	1	2.1	3.7	4.1
nand	0.36	1	2.0	7.9	23
chat_parser	2.4	1	1.7	11.0	19

TABLE 5.1. Parma - Execution Times

In the cases where Parma performs best, e.g. *tak*, *qsort* and *crypt*, the information from global analysis is very precise and the resulting MIPS instructions are little different to those that would result from an equivalent C program. There are several factors involved in the cases where Parma produces the smallest performance gains. The information produced by global analysis for the *boyer* benchmark is poor. This is difficult to remedy. The heart of the problem is the predicate *rewrite_args*/3 which incrementally instantiates a compound term using the built-in predicate *arg*/3. The resulting term will always be ground but this can only be inferred by induction. The loss of precision for this predicate propagates through the entire program. A single exit mode declaration for the predicate *rewrite_args*/3, indicating that its third argument is always ground on exit, produces a performance improvement of more than a factor of 2. Little performance improvement is possible for the *zebra* benchmark because its performance is dominated by calls to the general unify routine. This could perhaps be remedied by sophisticated transformations. The *reducer* benchmark appears, as far as we can determine, less tractable.

These measurements give us confidence that Parma will offer many programs a factor of 15 or better performance improvement over SICStus 0.6. In some cases the performance gain will be less. The gap betwen Parma and SICStus 3.0 is much smaller. Parma is mostly at least 5 times faster, occasionally less. Some of this difference must result from the cost of extra features which SICStus 3.0 provides such as cyclic terms.

We also compared the size of the code produced for each benchmark. Without global analysis Parma produces slightly larger code than SICStus but in the worst

	reference chain analysis	trailing analysis	structure/list analysis	uninitialised variables
nreverse	2.31	2.16	1.34	1.35
tak	1.00	1.00	1.00	1.24
qsort	1.52	1.00	1.38	1.29
pri2	1.17	1.10	1.05	1.07
serialise	1.08	1.11	1.63	1.08
queens_8	1.07	1.03	1.29	1.10
mu	1.05	1.16	1.51	1.16
zebra	1.05	1.00	1.34	1.00
deriv	1.25	1.44	1.00	1.15
crypt	1.12	1.09	1.17	1.07
query	1.00	1.00	1.00	1.18
prover	1.06	1.13	1.26	1.03
poly_10	1.48	1.07	1.09	1.23
browse	1.14	1.07	1.48	1.12
press1	1.12	1.11	1.58	1.19
reducer	1.05	1.03	1.00	1.13
boyer	1.06	1.07	1.02	1.07
nand	1.17	1.05	1.29	1.15
chat_parser	1.11	1.09	1.19	1.02

TABLE 5.2. Parma - Evaluation of Individual Features

case, *boyer*, it is only a factor of 2.5 larger. With global analysis the code produced by Parma is significantly smaller than SICStus for most benchmarks. However, there is considerable variation and Parma produces larger code for three of the benchmarks. Even without global analysis the code Parma produces would probably not provoke significantly worse caching or paging performance than SICStus.

We wished to ascertain that the inclusion of several techniques employed by Parma was worthwhile. These features are unlikely to have negative effects on the code produced. Their costs are only in slowing compilation and increasing Parma's complexity.

We have evaluated the benefits of three features of Parma's abstract domain. Two of these are the sub-features of the variable symbol which represents reference chain and trailing information. The third feature is the inclusion of compound *list* and *term* symbols to allow exact representation of compound terms. We would have preferred to separately evaluate the benefits of the *list* and *term* symbols but this was not possible as their implementation was too closely intertwined. We have also evaluated the benefits gained by the compilation phase designating some predicate argument positions uninitialised.

We measured the benefit from each feature by compiling each benchmark with the feature disabled. Table **??** contains this time scaled with respect to the normal execution time. In effect this is the speed-up factor offered by this feature for that benchmark. As this is only the marginal benefit, comparisons must be made carefully. The marginal benefit of two features taken together need not be, and probably will not be, their product. It could be less because the features compete or it could be more because the features co-operate. For example, uninitialised

argument variables do not need trailing when bound but this would also often be determined by trailing analysis so these two features compete. Removing either of these features may have little effect because the other takes up the slack but removing both features would have a significant effect. Therefore their combined marginal benefit is larger than suggested by the figures in Table **??**.

The measurements demonstrate that all four features produce useful improvements in over half the benchmark programs. The improvement factors considered alone may not seem large but they are each significant components of Parma's high performance. As can be seen by comparing Table **??** to Table **??**, these four features by no means account for all of Parma's high performance. Much comes from the type information global analysis yields. This gain is not surprising as it is analogous to the benefit some existing implementations obtain with the information from user's mode declarations.

Although during the development of Parma we regularly compared the code it generated with that from a C compiler for analogous C programs, we did not realise the significance of the results until Van Roy [**?**] pointed out that the performance gap had been closed with imperative languages. For examples such as *qsort* the code Parma generated matched the performance of code generated by the MIPS C compiler for analogous C programs. It is important to note we retained the list data structure in the C program and the difference in data structures employed makes comparison between the two languages of limited value.

6. CONCLUSION

The importance of Parma lies in its span. The work began at the compiler coalface dealing with issues like delay slots, tag manipulation and instruction cycles. This bottom-up approach resulted in Parma's high performance. We knew what information was most desirable before we constructed the analysis phase. This allowed insights missed by other work on analysis.

Parma is not a complete Prolog implementation and never will be. It has not been used since the benchmarking presented above. It has been solely a vehicle to test ideas described. While the ideas embodied in Parma have since been the subject of some interest in implementation research, it is not clear that they have had, or will have, much practical impact. The author does not feel that these techniques are necessarily the best for implementing Prolog or other logic programming languages. Parma was a successful attempt to fit Prolog implementation into the paradigm used by conventional imperative languages. There may be other better possibilities outside this paradigm such as delaying compilation until execution time.

We feel that the most important lesson is that it is possible for sufficiently sophisticated implementation techniques to bridge a large gap between language and machine. This should relax the shackles of efficiency a little on the language designer.

7. ACKNOWLEDGEMENTS

Three anonymous referees supplied many constructive comments. Mats Carlsson supplied the timings for Sicstus 3. He also provided excellent support of SICStus Prolog from the other side of the world. I would also like to thank Norman Foo and Adrian Walker for their guidance and support.

REFERENCES

1. R. Sethi A. Aho and J. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, 1986.
2. M. Carlsson and J. Widen. Sicstus Prolog users manual. Technical Report R88007B, SICS, 1988.
3. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of fixed points. In *4th ACM Symposium on Principles of Programming Languages*, pages 78–88, 1977.
4. G. Kane. *MIPS RISC Architecture.* Prentice-Hall, Englewood Cliffs NJ, 1988.
5. C. S. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2(1):43–66, 1987.
6. P. Van Roy and A. Despain. The benefits of global dataflow analysis for an optimizing Prolog compiler. In *North American Conference on Logic Programming*, October 1990.
7. Peter Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming.* PhD thesis, University of California, Berkeley, 1990.
8. Peter Van Roy. 1983-1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming*, 1994.
9. L. Sterling and U. Shapiro. *The Art of Prolog.* MIT Press, 1986.
10. A. Taylor. ftp://ftp.cse.unsw.edu.au/pub/users/andrewt/prolog_benchmarks.gz.
11. A. Taylor. Removal of dereferencing and trailing in Prolog compilation. In *Sixth International Conference on Logic Programming*, July 1989.
12. A. Taylor. Lips on a MIPS: Results from a Prolog compiler for a RISC. In *Seventh International Conference on Logic Programming*, July 1990.
13. A. Taylor. *High Performance Prolog Implementation.* PhD thesis, University of Sydney, ftp://cse.unsw.edu.au/pub/users/andrewt/phd_thesis.ps.gz, 1991.
14. C.-J. Peng V.S. Madan and G. S. Sohi. On the adequacy of direct mapped caches for lisp and Prolog data references patterns. In *Proceedings of the North American Conference on Logic Programming*, October 1989.
15. D.H.D. Warren. *Applied Logic - Its Use and Implementation as a Programming Tool.* PhD thesis, University of Edinburgh, 1977.
16. D.H.D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, California, 1983.