On the Practicality of Global Flow Analysis of Logic Programs

Richard Warren
Manuel Hermenegildo

Advanced Computer Architecture
Microelectronics and Computer Technology Corporation
Austin, TX 78759

Saumya K. Debray

Department of Computer Science
University of Arizona
Tucson, AZ 85721

ABSTRACT

This paper addresses the issue of the practicality of global flow analysis in logic program compilation, in terms of both speed and precision of analysis. It discusses design and implementation aspects of two practical abstract interpretation-based flow analysis systems: MA^3, the MCC And-parallel Analyzer and Annotator; and Ms, an experimental mode inference system developed for SB-Prolog. The paper also provides performance data obtained from these implementations. Based on these results, it is concluded that the overhead of global flow analysis is not prohibitive, while the results of analysis can be quite precise and useful.

1. Introduction

The extensive use of advanced compilation techniques [1, 23, 21], coupled with parallel execution [5, 7, 13, 16, 25], appears to be a very promising approach to achieving improved performance in logic programming systems. Existing systems are based largely on local analysis (i.e. clause-level or, at most, procedure-level, as in the WAM). Such techniques have already brought substantial performance improvements to popular Prolog systems [11, 21, 22]. However, global analysis offers the potential to attain substantially higher execution speeds. This has given rise to a great deal of research in flow analysis-based optimization of logic programs (e.g. see [3, 9, 17, 18]). These theoretical studies have proven the correctness of different types of analysis and their termination properties. However, in order that the analysis and optimization of large programs be practical, it is necessary that such analysis algorithms be both precise and efficient. The question remains then about whether flow analysis can actually be done routinely with useful precision in a reasonable amount of time, and,

if so, what implementation techniques might be used to achieve this goal.

This paper addresses the issue of the practicality and implementability of flow analysis of Prolog programs. It reports on the design, implementation, and performance of two practical abstract interpretation-based flow analysis systems: MA^3, the MCC And-parallel Analyzer and Annotator; and Ms ("Mode system"), an experimental flow analysis system developed for SB-Prolog. Section 2 briefly introduces the concept of "abstract compilation" used in these two systems while Section 3 discusses various implementation approaches and their tradeoffs. Section 4 presents a sample application of the mode information obtained and Section 5 offers performance figures and a discussion of these results. Finally, Section 6 summarizes our conclusions which indicate that quite good precision can be attained and at a reasonable cost.

2. Preliminaries

2.1. Dataflow Analysis of Logic Programs

The purpose of dataflow analysis is to determine, at compile time, properties of the terms that variables can be bound to, at runtime, at different points in a program. Since most "interesting" properties of programs are undecidable, the information obtained via such static analyses is typically conservative. Nevertheless it can be used in many cases to improve the quality of code generated for the program.

Most of the flow analyses that have been proposed for logic programming languages are based on a technique called *abstract interpretation* [6]. The essential idea here is to give finite descriptions of the behavior of the program by symbolically executing the program over an "abstract domain," which is usually a complete lattice or cpo of finite height. Elements of the abstract domain and those of the actual computational domain are related via a pair of monotone, adjoint functions referred to as the *abstraction* (α) and *concretization* (γ) functions. In addition, each primitive operation f of the language is abstracted to an operation f' over the abstract domain. Soundness of the analysis requires that the concrete operation f and the corresponding abstract operation f' be related as follows: for every x in the concrete computational domain, $\gamma(f'(\alpha(x))) \leq f(x)$.

2.2. Abstract Compilation

A naive implementation of a global flow analysis system, based on the technique suggested by the name "abstract *interpretation*," might proceed by modifying a standard meta-circular interpreter to compute over the abstract domain. An alternative is to specialize such an abstract interpreter to deal with only the program under consideration. This can be

done by making a single pass over the program P to be analyzed and producing a transformed program $\tau(P)$ which, when executed, yields precisely the desired flow information about the original program P (see Figure 1).[1] The transformation τ is determined by the flow information desired. The practical benefit of this approach is that since the flow information is obtained by executing the transformed program directly, instead of having the underlying system execute the abstract interpreter which in turn symbolically executes the original program, one level of interpretation is avoided during the iterative fixpoint computation characteristic of dataflow analyses. Since much of the cost of global flow analyses is in these iterative fixpoint computations, this results in significantly more efficient analyses. The technique, which for lack of a better name we refer to as "abstract compilation," was (to the best of our knowledge) first suggested in [9].

3. Implementation Issues

Though the idea of abstract interpretation has been applied to logic programs by various researchers [2, 15, 19], its implementation has often been regarded as computationally expensive. As a result, few practical implementations have actually been reported in the literature. We argue, however, that this perception is not justified, and that if properly implemented, global flow analysis systems for logic programs need not be overly expensive. In this section, we concentrate on various implementation issues for efficient global dataflow analysis systems.

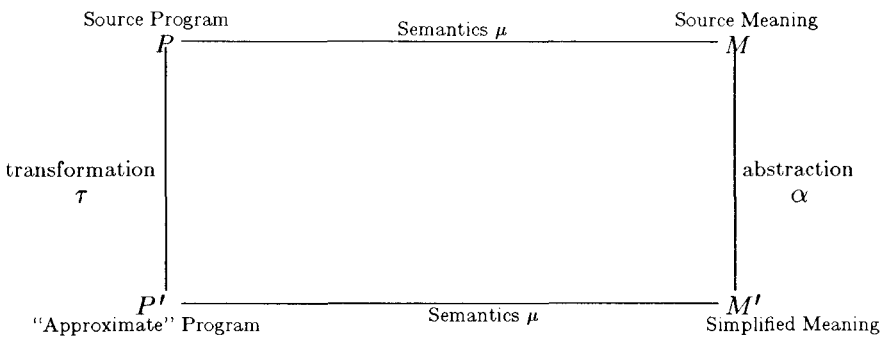

Figure 1: Analysis, abstraction and "approximate" programs

[1] That this transformation can be thought of as a partial evaluation of the abstract interpreter was suggested to us by Mike Codish [4]; see also [14].

3.1. Implementation of Extension Tables

An important component of a flow analysis system is the *extension table*, which is a memo structure that records dataflow information during analysis. A central issue in the design of the program transformation system, discussed in the previous section, is the implementation of this table: while the extension table module may appear to be a rather small component of the entire flow analysis system, design and implementation decisions made for this component can have profound repercussions on the design, implementation and performance of the remainder of the system. For this reason, the issues and tradeoffs involved are discussed at some length. It is assumed that the flow analysis system is being implemented on top of, rather than as part of, a conventional Prolog system. This means that there are two basic approaches to implementing the extension table: (*i*) as part of the Prolog database, with operations on the table effected via side effects, through *assert* and *retract*; and (*ii*) using Prolog terms as the data structures representing the table, with table operations affected via unification.

There are several advantages to implementing the extension table as part of the Prolog database. The most important of these is that the program transformation is simplified considerably: firstly, the table becomes a global structure that does not have to be passed around explicitly; more importantly, all execution paths in the program can be explored in a relatively straightforward way. For the analysis of a program to be sound, it is necessary that every execution path that can be taken at runtime be explored during analysis. If operations on the table are persistent across backtracking, then this can be effected simply by adding a *fail* literal at the end of each transformed clause. The effect of this, when the transformed clause is executed, is that after the body has been processed, execution is forced to backtrack into the next possible execution path. In this manner, every execution path in the program is considered during analysis (cuts in the source program are discarded during transformation, so they do not pose a problem). Moreover, once the transformed program has been implemented in this manner, another advantage becomes apparent: because execution is made to fail back as soon as an execution path has been explored, space used on the various Prolog stacks during the analysis of that path can be reclaimed relatively efficiently. The MA^3 system currently uses the Prolog database for extension table implementation.

The principal disadvantage in implementing the extension table as part of the Prolog database is that operations on the table use *assert* and *retract*, which are relatively expensive (e.g. in three representative systems, asserting a unit clause is between two and three orders of

System	*unification*	*assert*	*accessing asserted code*
Quintus 1.6	1.0	544-1477	300-930
SB-Prolog 2.3	1.0	3038-6075	103-144
Sicstus 0.5	1.0	359-678	308-639

Table 1: Normalized costs of some operations in representative Prolog systems (abstracted from the results of a benchmark suite due to Fernando Pereira [20])

magnitude slower than doing a simple unification, see Table 1). This might be less of a problem if access to asserted clauses were very fast. Unfortunately, as can be seen from Table 1, accessing asserted code is also relatively expensive in most current Prolog systems. There is also a hidden cost in the failure-driven exploration of execution paths: this approach requires that choice points be created at the entrance to predicates with more than one applicable clause. There could be a significant cost incurred in this, since the creation of a choice point is typically relatively expensive. The tradeoffs here, however, are more complex: for example, it is difficult to compare the cost incurred in creating these choice points with the time saved in failure-driven space reclamation as compared to garbage collection.

Another approach is to implement the extension table as a Prolog term, with operations on the table effected via unification. The principal advantage of this approach is that *assert* and *retract* are not necessary for manipulating the table. Instead, unification –which, as mentioned above, is two to three orders of magnitude faster– is used. The principal disadvantage of this approach is that because operations on the table are undone on failure and backtracking, the program transformation must explicitly force all execution paths to be explored. This makes the transformation more complex. The fact that the extension table has to be passed around explicitly as a parameter to all relevant predicates also adds to the size of the transformed program.

In the Ms analysis system, the extension table is currently maintained as a Prolog structure, and the exploration of every execution path in the program is guaranteed as follows: each transformed clause is given an extra argument, the *clause number*. Corresponding to each predicate there is a driver which calls each numbered clause in turn, collects the results, and returns a summary (in this case, their least upper bound) to the caller. Thus, the transformed predicates for a predicate p with m clauses look something like

```
p$pred(InMode, ExtTbl, OutMode) :-
        p$cl(1,InMode, ExtTbl, OutMode_1),
        . . .,
        p$cl(m,InMode, ExtTbl, OutMode_m),
        lub([ OutMode_1, . . ., OutMode_m ], OutMode).

p$cl(1, InMode, ExtTbl, OutMode) :- . . .
   ...
   ...
p$cl(m, InMode, ExtTbl, OutMode) :- . . .
```

In systems that support indexing on asserted clauses, an index will be created on the first argument (corresponding to the clause number) of the transformed predicate *p$cl*. This has the advantage that selection of the different clauses then becomes deterministic, so no choice points need to be created for the different *p$cl* calls. This, in turn, leads to space and time savings. On the other hand, this approach does not permit failure-driven space reclamation.

3.2. Handling Aliasing

Aliasing refers to the situation where two or more variables co-refer. An early approach to handling aliasing involved reasoning about the "safety" of variable bindings [9]; while sound, this was highly conservative. More recently, researchers have suggested a uniform treatment of the problem based on associating explicit *dependency sets* with variables [2, 10]. In either case, efficiency suffers because of the cost of having to explicitly maintain and propagate dependency sets.

An alternate approach, used in the MA^3 system and presented herein, is to retain the logic variable representation for unbound variables, rather than mapping them to special symbols such as "–" or "?" as in other mode inference systems [9, 17]. This has an advantage over other approaches in that unification in the underlying Prolog system can be used to keep track of aliasing between variables. Only ground terms are reduced to their "canonical" form, represented by a special (Prolog) constant '$ground', denoted herein for brevity by the symbol "Δ". Nonvariable terms are processed recursively, resulting in a pseudo-canonical form. Thus, the term $f(a, g(1, X))$ simplifies to $f(\Delta, g(\Delta, X))$.

Abstract unification in MA^3 is defined as follows: if either term being unified is a variable, then they are unified using standard unification; otherwise, if one of them is ground (i.e. is either a ground term, or is bound to the special constant Δ), then as a result of abstract unification all variables occurring in the second term become instantiated to Δ, representing the fact that they are ground as well; otherwise they must both be nonvariable terms different from Δ: in this case, if their principal functors

match, the arguments are processed recursively; otherwise, abstract unification fails. A call is processed as follows: it is first simplified as far as possible (e.g. by replacing ground terms by Δ). The calling pattern is then checked against the extension table to see whether there is a more general entry in the table. If such an entry exists, and has an associated success pattern, then this success pattern is returned. If there is a more general calling pattern in the extension table but no associated success pattern, execution fails. Otherwise, the calling pattern is entered into the extension table. Each clause for the called predicate is then processed with a fresh copy of the calling pattern, with variables renamed: abstract unification of the call with the head of the clause is carried out, the body of the clause processed, and a success pattern determined. The success patterns for the different clauses are then collected, and the least upper bound computed and associated with the calling pattern entry in the extension table. The process can be illustrated by the following example. Consider the program

```
:- module(test, [p/3]).          % Exporting p/3.
:- imode p(V,f(W,W), Δ).         % The call specification.

p(X,f(X,Z),g(Y)) :- q(X,Y,Z).
q(U,U,V).
```

The current output of the MA^3 analyzer for the above program is:

Input	Call	Output(s)
q(v,Δ,v)	q(_409,Δ,_409)	[q(Δ,Δ,Δ)]
p(v,nv,Δ)	p(_409,f(_413,_413),Δ)	[p(Δ,Δ,Δ)]

where "v" represents an unbound variable and "nv" a nonvariable term. Abstract unification of the call with the head of the clause for $p/3$ causes (i) X and Z to become unified; and (ii) Y to become instantiated to Δ. The calling pattern for q is therefore obtained as $q(X, \Delta, X)$. The predicate $q/3$ is now processed with this calling pattern. The reader may verify that the success pattern obtained for q is $<\Delta, \Delta, \Delta>$. At this point, the terms in the head of the clause are bound to $<\Delta, f(\Delta, \Delta), g(\Delta)>$. These are then simplified to yield the tuple $<\Delta, \Delta, \Delta>$, which is the success pattern for the clause.

This technique exploits Prolog's unification and logical variables to propagate aliases in a natural manner, avoiding the complications of having to maintain and update dependency sets at every stage. An added benefit is that because of the way the abstract unification is defined, the precision of analysis improves significantly. Despite these advantages, however, this technique suffers from one shortcoming: since Prolog variables are used to represent both the elements "free" and "unknown" in

the abstract domain, they are overloaded. As a result, two passes over a program are required to infer the "?" ("any," or "unknown") mode: the first using a "worst-case" representation of terms, the second using a "best case" binding. There is also the overhead associated with creating copies of terms repeatedly, but as the results reported in Section 5 indicate, these overheads are not unduly large (in any case, whereas the maintenance of dependency sets requires taking transitive closures, which costs $O(n^3)$ for a tuple of size n, a term can be copied in time proportional to its size). If the extension table is implemented using *assert*, then these copies can be created by simply using *call*/1; if it is implemented as a Prolog term, then copies must be created explicitly.

3.3. Other Optimizations

Because of the high cost of *assert*, it is advantageous to shift as much work as possible from within asserted code to within compiled code, so as to reduce the amount of asserting necessary. For example, it is substantially cheaper not to create and assert the *p$pred* clause shown at the end of Section 3.1, with m+1 literals in the body, directly as given. Instead, we define a compiled predicate *mode_iterate* that takes a template of the *p$cl* goals and the number of clauses m, invokes each of the *p$cl* goals, collects their individual output modes, computes the least upper bound of these and returns it as the overall output mode. This reduces the size (and cost) of asserting the *p$pred* clause significantly. The *p$pred* clause that is asserted now looks simply like

```
p$pred(InMode, ExtTbl, OutMode) :-
  mode_iterate( p$cl(_, InMode, ExtTbl, _), OutMode).
```

While this makes some extra term copying necessary at runtime (m copies of the *p$cl* template have to be created), the overhead involved (depending on the cost of *assert*) is usually more than offset by the savings in *assert*.

Another optimization that can result in significant reductions in the amount of code asserted, and cause substantial improvements in the speed of the system, is to check "database" predicates, i.e. predicates defined entirely by unit clauses, and eliminate clauses that are redundant with respect to success pattern computation.

3.4. Effects of Program "Cleanness" on Flow Analysis

While "impure" language features such as *var*/1, *nonvar*/1, *cut*, etc., can be handled without any trouble, a significant problem in reliable flow analysis is the use of features such as *call*/1, *not*/1, etc., where the argument appearing in the program text is a variable. Such goals are difficult and expensive to analyze correctly, and can affect the precision and efficiency of analysis significantly. A similar problem arises with *assert*

and *retract*. Neither of the two flow analysis systems described here address these problems at this time. What is curious is that in almost every program containing such "dirty" features that we looked at, their use was not really necessary, and seemed to be a hangover from an imperative programming style. Our experience indicates that (*i*) "clean" programs are desirable not only for their aesthetic and semantic appeal, but also for the very pragmatic reason that such programs are much more amenable to compiler analysis and optimization; and (*ii*) "unclean" features can often be avoided with a little effort during coding.

4. An Application: AND-parallelism Detection

As an example, this section discusses the application of mode inferencing to the generation of *Conditional Graph Expressions* (CGEs) [13] for AND-parallel execution, one of the major current applications of the MA^3 system [26]. Note, however, that the application of mode information is in general much broader, ranging from other high-level applications, such as the improvement of Prolog's backtracking behavior, to low-level applications relating to details of code generation in Prolog compilers. Together, they underscore the importance of mode information at all levels in optimizing compilers for high-performance logic programming systems.

CGEs are a mechanism (derived from DeGroot's ECEs [7]) for the generation and control of parallelism in Independent/Restricted AND-parallelism [7, 13] —an efficient type of parallelism in which only independent goals are executed in parallel. CGEs appear in the bodies of Horn clauses and augment such clauses with conditions which determine the independence of goals and provide control over the spawning and synchronization of such independent goals during parallel forward execution and backtracking. A CGE is defined as an independence condition *i_cond*, followed by a conjunction of goals, i.e.:

(*i_cond* // goal_1 & goal_2 & ... & goal_n)

i_cond is a sufficient condition (to be checked at run-time) which when met guarantees the independence of the goals in the conjunction. Operationally, $goal_1$ through $goal_n$ can be run in parallel if *i_cond* is met; otherwise they are run sequentially. Goals in a CGE may themselves be either standard Prolog goals or other CGEs so that complex execution graphs can be encoded. Such execution graphs and expressions can be generated by the user, but a more desirable situation is, of course, that they be generated automatically by the compiler. DeGroot [8], Chang et al. [3], and Warren and Hermenegildo [26, 13] have addressed this subject. The two main issues involved in the CGE generation process are how to associate the goals in a clause into groups for parallel execution (each group being the body of a CGE —*goal grouping*) and how to determine conditions for

independence for each group (*i_cond generation*). Given a particular goal grouping, and considering only local analysis (i.e. restricting the analysis to a single clause) a *sufficient i_cond* can be trivially given by the conjunction [13]:

ground(*list_of_variables*), indep(*list_of_tuples*)

where *list_of_variables* is the set of all variables which appear in more than one conjunct contained within the CGE, and *list_of_tuples* is the minimal set of pairs of non-shared variables which appear in different conjuncts. The ground check succeeds if every variable in *list_of_variables* is instantiated to a ground term when the test is made at runtime; the "indep" check succeeds if for all pairs in *list_of_tuples* the two variables in each pair are bound to terms which do not share variables.

The conditions above are sufficient but not necessary in the majority of cases. Since the "indep" and "ground" checks can be expensive (e.g. if the checks are performed on deeply nested structures) it is imperative to reduce them to the minimum. A limited number of checks can be eliminated by additional local analysis, using knowledge about the modes of builtins and the fact that first occurrences of variables are always unbound. However, local analysis proves relatively limited. On the other hand, our experience with the MA^3 system shows that, given a global analyzer capable of inferring groundness and independence of variables[2] CGE checks can be significantly reduced and sometimes eliminated altogether at compile time through partial evaluation with the mode information.

Table 2 summarizes some of our preliminary experiments in applying inferred mode information to CGE generation: the number of checks is significantly reduced and in some cases CGEs are generated with no checks, resulting in parallel execution with no independence detection

Benchmark	Total CGEs	ground chks	indep chks	CGEs w/no chks
queens	3	4/2	3/2	0/1
qplan	20	13/5	57/17	0/7
saltmust	8	8/2	30/10	0/6
deriv	4	4/0	16/4	0/0

Table 2 : Statistics on Conditional checks contained by CGEs (*without* mode information/*with* generated mode information)

[2] This information can also be supplied, if so desired, in the form of *:-imode* and *:-omode* directives (e.g. by the user) and then only local analysis is required.

overhead. Note, however, that the results presented in Table 2 represent *lower bounds* on CGE optimization and are expected to improve as our tools mature. Most significantly, the results presented are based on MA^3 *inferring term groundness only.* The system is currently being extended to also use variable independence information, generated using the techniques presented in section 3.2, and this and other refinements should continue to optimize the CGEs, further improving runtime performance.

Although we have concentrated on the issue of *i_cond* determination, the groundness and independence mode information is also essential in the goal grouping process, mode analysis therefore representing an important tool for the efficient implementation of AND-parallelism. I addition, the same techniques can be applied to the generation of other types of (non CGE-based) execution graphs and to other types of AND- and OR-parallel execution. For example, the knowledge that variables are ground (and therefore, read-only) could be used to selectively avoid multiple binding environment maintenance overheads in OR-parallel systems.

5. Performance

In this section we offer timings and other statistics obtained from the two inference systems presented in this paper (MA^3 and Ms). These figures support our claim that global program analysis need not be computationally overwhelming: the cost fraction corresponding to a flow analysis pass added to a typical Prolog compiler would seem to be of the order of 30-80%.

Tables 3 and 4 give two different performance perspectives, efficiency and precision. The benchmark programs used were the following: *asm*, the SB-Prolog assembler; *boyer*, from the Gabriel benchmarks, by Evan Tick; *browse*, from the Gabriel benchmarks, by Tep Dobry and Herve Touati; *func*, a functionality inference system written for SB-Prolog; *projgeom*, a program due to William Older; *peephole*, the peephole optimizer used in SB-Prolog; *preprocess*, a source-level preprocessor used in the SB-Prolog compiler; *queens*, a program for the *n*-queens problem; *read*, the public-domain Prolog parser by Richard O'Keefe and D. H. D. Warren; and *serialize*, by D. H. D. Warren. They constitute a set of "real" programs representing a wide mix of application areas, characteristics, and coding styles.

Table 3 gives analysis vs. compile times: as can be seen, flow analysis takes up 27-50% of the total compilation time in the Ms system (actual *analysis* time of a benchmark is compared to the time taken by the SB-Prolog compiler to compile the benchmark), and from 50-82% in the MA^3 system (idem. with respect to the Quintus compiler). In each case, most of the time charged to mode inference is in fact taken up in asserting the

Benchmark	Analysis Time T_1	Total Compile Time T_2	T_1/T_2
asm	60.50	93.73	0.65
boyer	23.13	42.19	0.55
browse	31.55	38.27	0.82
func	36.80	53.90	0.68
peephole	22.52	38.90	0.58
preprocess	73.17	96.21	0.76
projgeom	3.25	6.07	0.50
queens	2.67	5.47	0.49
read	60.18	78.11	0.77
serialize	4.15	7.17	0.58

Table 3(a): MA^3 Compile vs. Analysis times (secs, using Quintus 2.2, Sun 3/50)

Benchmark	Analysis Time T_1	Total Compile Time T_2	T_1/T_2
asm	103.76	242.84	0.43
boyer	48.30	140.32	0.34
browse	18.08	66.94	0.27
func	66.00	136.94	0.48
peephole	47.80	115.26	0.41
preprocess	94.66	194.88	0.49
projgeom	8.40	18.90	0.44
queens	9.60	19.16	0.50
read	68.32	155.90	0.44
serialize	6.90	19.12	0.36

Table 3(b): Ms Compile vs. Analysis times (secs, using SB-Prolog 2.3.2, Sun 3/50)

"approximate" program. Thus, all these numbers could be improved by improving the efficiency of *assert*.

Unfortunately, we did not have the time to implement different extension table strategies, as discussed in Section 3.1, within the same flow analysis system to test their relative performances. While MA^3 uses the Prolog database to implement the extension table and Ms passes around a Prolog term, we would caution against using the figures in Table 3 to draw conclusions regarding the relative efficiencies of these two approaches, since the speeds of the underlying Prolog systems and compilers were very different. It is also our intuition that if a combination of the techniques used in both systems (and described in Section 3) is used, substantially better performance could be obtained.

Table 4 attempts to characterize the "precision" of the inference systems. Table 4(a) gives the precision of the MA^3 system, in terms of the

Benchmark	TAP	# "hits"	% hits
asm	113	92	81.4
boyer	69	38	55.0
browse	47	37	78.7
func	130	81	62.3
peephole	36	33	91.6
preprocess	139	116	83.4
projgeom	27	23	85.2
queens	20	20	100.0
read	141	126	89.3
serialize	15	13	86.6

Table 4(a) : Precision of the MA^3 system†

Benchmark	TAP	IAP	# "hits"	hits/IAP(%)	hits/TAP(%)
asm	96	69	67	97.10	69.79
boyer	61	35	7	20.0	11.48
browse	42	30	21	70.0	50.0
func	118	87	58	66.67	49.15
peephole	34	21	16	76.19	47.05
preprocess	131	92	46	50.0	35.11
projgeom	27	24	22	91.67	81.48
queens	21	17	16	94.12	76.19
read	147	85	51	60.0	34.69
serialize	14	7	4	57.14	30.77

Table 4(b) : Precision of the Ms system†

Key: TAP: Total # of argument positions; IAP: # of "interesting" arg. positions.

† Differences in the total number of argument positions in a program between tables 4(a) and 4(b) arise from differences in the set of predicates considered to be "builtins" by the two mode inference systems.

percentage of argument positions whose modes were correctly inferred. The values range from 55% to 100%, in most cases lying in the 80%-90% range. Thus, MA^3 proves to be quite precise, presumably due to the tracking of variable aliasing and structures of terms. Table 4(b) gives the precision figures for Ms. Unlike MA^3, Ms uses an extremely simple abstract domain – "ground," "nonvariable" and "unknown" – and makes no attempt to keep track of the structures of terms, relative positions of embedded variables within a term, etc. As a result, there are two sources of imprecision: (*i*) due to the inability to reason about "free" arguments;

and (*ii*) because no information is kept about term structures. In an attempt to distinguish between loss of precision due to these two effects, two different measures of precision are used: the *relative precision*, expressed as the percentage of "interesting," i.e. non-free argument positions, whose modes are correctly inferred by the system; and the *absolute precision*, expressed as the percentage of all argument positions whose modes are correctly inferred. It can be seen that the relative precision of the Ms system ranges, in most cases, from 70% to over 95%; for programs that pass around a lot of partially instantiated structures, such as *func*, *preprocess*, *read* and *serialize*, the lack of information about term structure results in a drop in the relative precision to between 50% and 70%. The *boyer* program is something of an anomaly, but the unusually low precision of inference in this case can be traced to the inference system's lack of sufficient knowledge about the builtins *functor*/3 and *arg*/3. As might be expected in this case, the inability to represent and reason about free variables results in somewhat lower absolute precision figures.

6. Conclusions

Global flow analysis offers information which can be very useful both in optimizing compilers and in the efficient exploitation of parallelism, the combination of which currently appears to be the best approach towards achieving increased performance in logic programming systems. Our experiences with the implementation of two flow analysis systems for Prolog (MA^3, the MCC And-parallel Analyzer and Annotator and Ms, a flow analysis system for SB-Prolog), as reported in this paper, show that the perception that global flow analyzers are computationally too expensive to be practical (assumed from the paucity of reports on actual implementations of such analysis systems) is unfounded. We have proposed novel implementation techniques, shown an example of an actual application of the information generated, and discussed some precision and performance tradeoffs. In addition, we have provided performance data obtained from the MA^3 and Ms implementations analyzing sizeable programs. The results showed that these systems are indeed practical tools: analysis time typically increases conventional compilation time by about a factor of 2 to 3, and considerable flow information is obtained which can result in significant speedups in program execution. Moreover, much of the current overhead is due to having implemented only a particular subset of the techniques presented herein and to inefficiencies in the underlying Prolog implementations (e.g. in *assert*) which can be improved upon. Our conclusion is therefore that such techniques can be used to implement global flow analysis systems that are quite precise, yet not overly expensive. Therefore, it is argued that flow analysis has indeed reached the stage of practicality.

References

1. D. L. Bowen, *NIP: New Implementation of Prolog*, Dept. of Artificial Intelligence, University of Edinburgh, May 1984. Unpublished manuscript.

2. M. Bruynooghe, A Framework for the Abstract Interpretation of Logic Programs, Research Report 62, Katholieke Universiteit, Leuven, Belgium, Oct. 1987.

3. J. Chang, A. M. Despain and D. DeGroot, AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis, in *Digest of Papers, Compcon 85*, IEEE Computer Society, Feb. 1985.

4. M. Codish, personal communication, , July 1986.

5. J. S. Conery, *Parallel Execution of Logic Programs*, Kluwer Academic Publishers, 1987.

6. P. Cousot and R. Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, *Conf. Rec. 4th ACM Symp. on Prin. of Programming Languages*, , 1977, pp. 238-252.

7. D. DeGroot, Restricted AND-Parallelism, in *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1984.

8. D. DeGroot, A Technique for Compiling Execution Graph Expressions for Restricted AND-parallelism in Logic Programs, in *Proc. 1987 Int. Supercomputing Conf.* , Athens, 1987. Springer-Verlag.

9. S. K. Debray and D. S. Warren, Automatic Mode Inference for Prolog Programs, in *Proc. 1986 Int. Symp. on Logic Programming*, Salt Lake City, Utah, Sept. 1986, pp. 78-88.

10. S. K. Debray, Static Inference of Modes and Data Dependencies in Logic Programs, Tech. Rep. 87-24, Dept. of Computer Science, University of Arizona, Tucson, AZ, Aug. 1987.

11. S. K. Debray, The SB-Prolog System, Version 2.3.2: A User Manual, Tech. Rep. 87-15, Department of Computer Science, University of Arizona, Tucson, AZ, Mar. 1988.

12. W. Drabent, Do Logic Programs Resemble Programs in Conventional Languages?, in *Proc. Fourth IEEE Symposium on Logic Programming*, San Francisco, CA, Sep. 1987, pp. 389-396.

13. M. V. Hermenegildo, in *Independent/Restricted AND-parallel Prolog and its Architecture*, Norwell MA 02061, 1988. Kluwer Academic Publishers.

14. N. D. Jones, P. Sestoft and H. Søndergaard, An Experiment in Partial Evaluation; The Generation of a compiler generator, in *Proc.*

First International Conference on Rewriting Techniques and Applications, Springer-Verlag LNCS vol. 202., Dijon, France, 1985.

15. N. D. Jones and H. Sondergaard, A Semantics-Based Framework for the Abstract Interpretation of Prolog, in *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin (ed.), Ellis Horwood, .
16. L. V. Kale, The REDUCE-OR Process Model for Parallel Evaluation of Logic Programs, in *Proc. Fourth International Conference on Logic Programming*, Melbourne, May 1987, pp. 616-632.
17. H. Mannila and E. Ukkonen, Flow Analysis of Prolog Programs, in *Proc. 4th. IEEE Symp. on Logic Programming*, San Francisco, CA, Sep. 1987.
18. C. S. Mellish, Some Global Optimizations for a Prolog Compiler, *J. Logic Programming 2*, 1 (Apr. 1985), 43-66.
19. C. S. Mellish, Abstract Interpretation of Prolog Programs, in *Proc. Third International Logic programming Conference*, London, July 1986. Springer-Verlag LNCS vol. 225.
20. F. Pereira, Prolog Benchmarks, in *Prolog Electronic Digest vol. 5, no. 56*, Aug. 1987.
21. *Quintus Prolog Reference Manual*, Quintus Computer Systems, Inc., Mountain View, CA, Apr. 1986.
22. *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, Sweden, Sep. 1987.
23. A. K. Turk, Compiler Optimizations for the WAM, in *Proc. 3rd. International Conference on Logic Programming*, London, July 1986, 410-424. Springer-Verlag LNCS vol. 225.
24. P. Van Roy, B. Demoen and Y. D. Willems, Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection and Determinism, in *Proc. TAPSOFT 1987*, Pisa, Italy, Mar. 1987.
25. D. H. D. Warren, OR-Parallel Execution Models of Prolog, in *Proceedings of TAPSOFT 87*, March 1987. Springer-Verlag LNCS.
26. R. Warren and M. Hermenegildo, MA^3 A System for Automatic Generation of CGEs, MCC ACA-ST Technical Report, Microelectronics and Computer Technology Corporation, 1988.