Global Flow Analysis as a Practical Compilation Tool [1]

M. V. HERMENEGILDO, R. WARREN, AND S. K. DEBRAY

Abstract

This paper addresses the issue of the practicality of global flow analysis in logic program compilation, in terms of speed of the analysis, precision, and usefulness of the information obtained. To this end, design and implementation aspects are discussed for 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 and, as an example of an application, a study of the usefulness of the mode information obtained in reducing run-time checks in independent and-parallelism. Based on the results obtained, 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 [8, 22, 30, 32, 33, 34], coupled with parallel execution [5, 10, 15, 20, 35], 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 [2, 7, 29]. However, global analysis offers the potential to attain substantially better object code and therefore even higher execution speeds.

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. This has given rise to a great deal of research in flow analysis-based optimization of logic programs (e.g. see [3, 9, 18, 21, 31, 32, 34, 24]).

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 in this technique is to give a finite description of the behavior of a program by symbolically executing it 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 *abs_f* be related as follows: for every x in the concrete computational domain, $f(x) \sqsubseteq \gamma(abs\_f(\alpha(x)))$.

Though the idea of abstract interpretation has been applied to logic programs by various researchers [1, 17, 18, 19, 23, 25], relatively few practical implementations appear to have actually been reported in the

[1] A preliminary version of this paper appeared in *Proc. 1988 International Conference on Logic Programming*. The work of M. Hermenegildo was supported in part by MCC's Deductive Computing Lab. and also in part by ESPRIT project 2471 "PEPMA." The work of S. Debray was supported in part by the National Science Foundation under grant number CCR-8702939.

literature: at this time, the only implemented systems that we are aware of, apart from those described in this paper, are those of Janssens [18], Mellish [24], Taylor [31], and Van Roy [34]. However, in order that the analysis and optimization of large programs be practical as a compilation tool, it is necessary that such analysis algorithms be both precise and efficient, and that the resulting information be of use for the intended purpose, be it proving properties of the program or improving execution speed. The question remains then regarding 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 analyses of Prolog programs. It reports on the design, implementation, and performance of two 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 deals with implementation issues: it briefly introduces the concept of "abstract compilation" used in these two systems (Section 2.1) and discusses various implementation approaches and their tradeoffs regarding extension tables, program transformations, treatment of builtins, etc. (Sections 2.2-2.3). Section 3 offers speed and precision performance figures and a discussion of these results. Section 4 presents as an example an application of the mode information obtained in the compilation of logic programs for independent and-parallel execution. Finally, Section 5 summarizes our conclusions, which indicate that quite good precision can be attained and at a reasonable cost.

2 Implementation Issues

Although abstract interpretation of logic programs has been proposed by various researchers, the paucity of reported implementations seems to suggest that its implementation may be regarded as computationally expensive. We argue that such a perception is not justified, and that if properly implemented, global flow analysis systems for logic programs need not be overly expensive. In this section, various implementation issues that are relevant to the efficiency of global dataflow analysis systems are discussed.

2.1 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 $P' = \tau(P)$ which, when executed, yields precisely the desired flow information about the original program P (see Figure 1). This transformation can be thought of as a partial evaluation of the abstract interpreter with respect to the input program P being analyzed[4].

The transformation τ is determined by the flow information desired. Abstract interpretation of a program consists essentially of "simulating" its execution over an abstract domain. This is done by specifying, as part of the abstract interpretation, an "abstract operation" *abs_f* for each primitive operation f of the language. To see how this should be done, it is necessary to make the primitive operations of the language – in our case, application of substitutions and unification – explicit. Let these primitive operations be denoted by predicates *app_subst* and *unify*: $app\_subst(\theta, t, t')$ is true if and only if the substitution θ, applied to the term t, yields the term t', i.e. $t' = \theta(t)$; and $unify(\theta, t_1, t_2, \sigma)$ is true if and only if the terms t_1 and t_2, unified in the context of the substitution θ, yield the substitution σ, i.e. $\sigma = \psi \circ \theta$, where ψ is the most general unifier of $\theta(t_1)$ and $\theta(t_2)$. Consider the execution of a clause $p(\bar{T_0}) :- q_1(\bar{T_1}), \ldots, q_n(\bar{T_n})$. Initially, each variable in the clause is uninstantiated. First, the arguments in the head of the clause are unified with those in the call to yield a substitution θ_0. The first literal in the body is then evaluated in the context of this substitution; if this succeeds yielding a new substitution θ_1, the next literal in the body is evaluated in the context of θ_1, and so on. Finally, when all the literals in the body have been successfully evaluated, yielding

$$\begin{tikzcd}[column sep=huge, row sep=huge]
\text{Source Program} & & \text{Source Meaning} \\
P \arrow[rr, "\text{semantics } \mu"] \arrow[d, "\text{transformation } \tau"'] & & M \arrow[d, "\text{abstraction } \alpha"] \\
P' \arrow[rr, "\text{semantics } \mu"] & & M' \\
\text{“Approximate” program} & & \text{Simplified Meaning}
\end{tikzcd}$$

Figure 1: Analysis, abstraction and “approximate” programs

a substitution θ_n, the “return value” is obtained by applying θ_n to the tuple of arguments in the head of the clause.

This operational behavior can be made explicit by rewriting the clause as follows:

$p(\bar{X}_{in}, \bar{X}_{out})$:-
  $unify(\mathbf{id}, \bar{X}_{in}, \bar{T}_0, \theta_0)$,
  $apply\_subst(\theta_0, \bar{T}_1, \bar{T}_{1,in})$, $q_1(\bar{T}_{1,in}, \bar{T}_{1,out})$, $unify(\theta_0, \bar{T}_{1,in}, \bar{T}_{1,out}, \theta_1)$,
  $apply\_subst(\theta_1, \bar{T}_2, \bar{T}_{2,in})$, $q_2(\bar{T}_{2,in}, \bar{T}_{2,out})$, $unify(\theta_1, \bar{T}_{2,in}, \bar{T}_{2,out}, \theta_2)$,
  $\ldots$,
  $apply\_subst(\theta_{n-1}, \bar{T}_n, \bar{T}_{n,in})$, $q_n(\bar{T}_{n,in}, \bar{T}_{n,out})$, $unify(\bar{T}_{n,in}, \bar{T}_{n,out}, \theta_n)$,
  $apply\_subst(\theta_n, \bar{T}_0, \bar{X}_{out})$.

where $\bar{X}_{in}$, $\bar{X}_{out}$, $\bar{T}_{i,in}$ and $\bar{T}_{i,out}$, $i = 1, \ldots, n$, are distinct new tuples of variables, and **id** is the identity substitution. Each k-ary predicate—which can be thought of as a predicate that takes one argument that is a k-tuple of terms—has been modified to have two arguments: the first, subscripted “*in*”, representing the tuple of arguments at the call to the predicate, and the second, subscripted “*out*”, representing the tuple of arguments at the return from that call.

It is important that we maintain separate sets of “calling” and “return” arguments. One reason for doing this is to make explicit the operational aspects of a logic program computation (since this is what an abstract interpretation tries to mimic). We contend that it also has declarative virtues, since it makes explicit the distinction between a term before a substitution is applied to it, and the term that results after the application of the substitution. The most important reason for this, however, is to anticipate a technical difficulty in abstract interpretation — certain kinds of static analyses require that the connection between "calling" and "return" values be maintained explicitly during analysis in order to avoid undue loss of precision.

The corresponding abstract interpretation computation can now be described simply by replacing the primitive operations *app_subst* and *unify* by the corresponding operations over the abstract domain, denoted by *abs_app_subst* and *abs_unify* respectively:

$$
\begin{array}{l}
\mathit{abs\_p}(\bar{X}_{in},\ \bar{X}_{out})\ \text{:-} \\
\quad \mathit{abs\_unify}(\alpha(\{\mathbf{id}\}), \bar{X}_{in}, \bar{T}_0, A_0), \\
\quad \mathit{abs\_app\_subst}(A_0, \bar{T}_1, \bar{T}_{1,in}),\ \mathit{abs\_q}_1(\bar{T}_{1,in}, \bar{T}_{1,out}),\ \mathit{abs\_unify}(A_0, \bar{T}_{1,in}, \bar{T}_{1,out}, A_1), \\
\quad \mathit{abs\_app\_subst}(A_1, \bar{T}_2, \bar{T}_{2,in}),\ \mathit{abs\_q}_2(\bar{T}_{2,in}, \bar{T}_{2,out}),\ \mathit{abs\_unify}(A_1, \bar{T}_{2,in}, \bar{T}_{2,out}, A_2), \\
\quad \ldots, \\
\quad \mathit{abs\_app\_subst}(A_{n-1}, \bar{T}_n, \bar{T}_{n,in}),\ \mathit{abs\_q}_n(\bar{T}_{n,in}, \bar{T}_{n,out}),\ \mathit{abs\_unify}(A_{n-1}, \bar{T}_{n,in}, \bar{T}_{n,out}, A_n), \\
\quad \mathit{abs\_app\_subst}(A_n, \bar{T}_0, \bar{X}_{out}).
\end{array}
$$

where $\alpha(\{\mathbf{id}\})$ represents the abstract domain element corresponding to (the singleton set containing) the identity substitution. The A_i are "abstract substitutions", i.e. abstract domain elements representing sets of substitutions. The resulting program is referred to as the "approximate" program.

While this transformation suffices to describe the computation over the abstract domain, it may not be suitable for direct evaluation by a top-down interpreter, e.g. Prolog. One reason for this is that abstract interpretation requires that all possible computation paths in the program be explored. Moreover, this program may not terminate if executed directly by a top-down interpreter. Thus, additional machinery is needed to force every computation path in the program to be explored and to ensure termination once a fixpoint has been reached. We address both these issues by evaluating the approximate program using extension tables [12]: this involves augmenting the approximate program with code to maintain and manipulate such extension tables.

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—with tongue firmly in cheek—we refer to as "abstract compilation," was (to the best of our knowledge) first suggested in [9]. Both the MA^3 and Ms systems use this technique in their implementations.

An important issue from the perspective of efficiency of analysis is not only how the transformation of the program is performed—since the transformation process obviously represents overhead—but also how the "approximate" program is incorporated into the Prolog system for execution. The issue of program transformation will be returned to later, after introducing the techniques for dealing with extension tables. The approach taken in order to make the "approximate" program executable will depend greatly on the characteristics of the underlying system. The most immediate alternative is to "assert" the transformed clauses into the database. Global analysis is then performed by simply calling the entry point of the transformed program. In a system in which asserted code is fully compiled, including indexing, this is a desirable solution because of its simplicity. In many systems, however, asserted code is actually interpreted and sometimes not even indexed. In those cases the performance advantage of "abstract compilation" is lost, since although one level of interpretation is eliminated another may be added. An alternative solution is to make the approximate program fully compiled by storing it in a temporary file and loading it into the Prolog system using the

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 [28].

standard compiler. There is an obvious tradeoff between these two alternatives: program assert overhead and perhaps slow analysis (dependent on the implementation of `assert`) vs. I/O and program compilation overhead but with a lower analysis time.

2.2 Implementation of Extension Tables

An important component of a flow analysis system is the *extension table* [12], 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.[2] 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. Figure 2 shows a simplified version of the program transformation used by the MA^3 system applied to the familiar `qsort` example. `'$unify'` goals perform the abstract unification, while the `'$findmode'` goals perform the failure-driven exploration of execution paths and LUB calculations.

[2]Note that section 4 presents results from an implementation where the global analyzer is *part* of a (parallel) Prolog system. However, in this case the whole compiler, as is often the case, is written in standard Prolog, and the considerations in this section still apply.

```
% Original program

qsort([],R,R).
qsort([X|L],R,R0 ) :-
        partition(L,X,L1,L2),
        qsort(L2,R1,R0),
        qsort(L1,R,[X|R1]).

% Transformed program

'compute$MODE'(qsort(A,B,C),Mode,Mode) :-
        '$unify'(qsort([],F,F),qsort(A,B,C)).
'compute$MODE'(qsort(A,B,C),InMode,OutMode) :-
        '$unify'(qsort([H|I],J,K),qsort(A,B,C)),
        '$findmode'(partition(I,H,L,M),InMode,N),
        '$findmode'(qsort(M,O,K),N,P),
        '$findmode'(qsort(L,J,[H|O]),P,Outmode).
```

Figure 2: Approximate program transformation in MA^3.

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 magnitude slower than doing a simple unification, see Table 1. This would be less of a problem if access to asserted clauses was 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. This can incur a significant cost, 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, which in turn increases the time and space taken to assert it.

In the Ms analysis system, the extension table is 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 have the structure shown in Figure

```
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) :- ...
```

Figure 3: Approximate program transformation in Ms.

3.

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.

2.3 Other Optimizations

Because of the high cost of `assert`, and the relatively slow speed of asserted code, 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 in Figure 3, 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$pred (InMode, ExtTbl, OutMode) :-
       mode_iterate( m, p$cl(_, InMode, ExtTbl, _), OModes),
       lub(OModes, OutMode).
```

The predicate `mode_iterate`, which is defined and compiled as part of the main analysis program, is given by the following:

```
mode_iterate(N, Call, OModes) :-
    N > 0 ->
       (OModes = [OMode | ORest],
        copy_terms(Call, Copy),
        arg(1, Copy, N),
        arg(4, Copy, OMode),
        call(Copy),
        N1 is N - 1,
        mode_iterate(N1, Call, ORest)
```

```
        ) ;
        OModes = [].
```

While this makes some extra term copying necessary at runtime (m copies of the `p$cl` template have to be created), the overhead involved is usually more than offset by the savings in `assert`. This is in some ways similar to the `'$findmode'` predicate used by MA^3. Note that if input and output modes are always ground terms, as in the Ms system, then the call to `copy_terms/2` above can be replaced by two calls to `functor/3`.

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 eliminate clauses that are redundant with respect to success pattern computation. This of course depends on the granularity of the abstract domain. For example, assuming an abstract domain that represents all ground terms by a single element of the abstract domain, given the set of facts and clauses

```
p(a,b).
p(c,[a,b,c]).
p(X,X).
p(e,f).
p(X,Y):- g(X), h(a,Y).
p(X,Y):- g(X), h(f(b),Y).
```

they can be represented by transforming only the following subset:

```
p(a,b).
p(X,X).
p(X,Y):- g(X), h(a,Y).
```

This optimization is especially effective for "database" predicates, or tables, which are defined entirely by unit clauses. As an example of the utility of this optimization, consider the benchmarks presented in Section 3. SB-Prolog's assembler, which is used in the *asm* benchmark, contains tables defining instruction names, opcodes, and their sizes: most of these clauses can be eliminated for mode inference purposes. The *peephole* benchmark, which is SB-Prolog's peephole optimizer, contains large tables that contain information about registers used and defined by different instructions: many of these can likewise be eliminated. The *read* benchmark, consisting of a Prolog tokenizer and parser, contains a table of operators and a table defining "special characters", which can also be subjected to this optimization. By eliminating redundant clauses in this manner, two kinds of savings are realized: the space and time taken to create and assert the approximate program decreases; and the time taken in the fixpoint computation also decreases. In our experiments, the speedups obtained from this optimization ranged up to a factor of 2 in some cases.

Another interesting issue is the treatment of builtin predicates. One simple alternative is to simply ignore such predicates in the analysis. This is however not desirable because a great deal of information can be derived from builtin predicates: first, the output modes of many builtin predicates are known and can be applied to subsequent goals in the path. Second, builtin predicates often require particular entry modes (for example, some arguments must be ground, others may have to be unbound variables) or otherwise they fail. An example of this is the `is/2` arithmetic predicate which requires its second argument to be ground (and an arithmetic expression). If it can be determined during the analysis that such conditions are not met then it can be concluded that the rest of the current path will not be executed resulting in analysis time saved and potentially increased precision. In addition, if no information is available regarding an argument for

Builtin	Input Mode	Output Mode
is/2	?, ground	ground, ground
</2	ground, ground	ground, ground
put/2	ground, ground	ground, ground
length/2	?,?	?, ground
var/1	?	var
number/1	?	ground

Table 2: Examples of builtin predicates modes.

which a builtin predicate enforces a particular mode, it can be assumed that if execution is to continue after that predicate, then the argument must have been bound to that mode. Table 2 shows some examples of modes for builtins in a simple $\{?, var, ground\}$ domain.

Finally, in order to provide a starting point for the abstract analysis a number of "query forms" are generally given to the analyzer along with the program, corresponding to the possible points at which execution of the program may be invoked (alternatively, all possible queries to all possible predicates in the program should be considered, but this will generally severely limit the amount of information that can be obtained from the analysis). In addition, ideally query forms should also include the set of abstract entry substitutions for each of these possible entry points. It is interesting to note that in a Prolog system with modules, such as Quintus Prolog [29], the module entry point information can actually be used as query forms, since it determines the points at which the program can be accessed from outside. This property is used in the MA^3 system so that in general the user does not need to provide any additional information to the global analysis system beyond the normal module declarations, global analysis thus not imposing any additional burden on the programmer. For example, a Quintus module declaration such as

```
:- module(foo, [main/2]).
```

which is found at the beginning of a file would instruct the system to perform global analysis of this file, starting with the `main/2` predicate. Of course, since no information is available at this point regarding input abstract substitutions the analysis would start with `:- main(?,?).` The user can of course provide additional information regarding the input abstract substitutions (for example, in MA^3 via `:-imode` and `:-omode` declarations).

2.4 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.

3 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 3-4 and 5-6 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 + rdtok*, the public-domain Prolog tokenizer and 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.

Tables 3-4 give 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 "approximate" program. Thus, all these numbers could be improved by improving the efficiency of `assert`. 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 Tables 3-4 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 2.2) is used, substantially better performance could be obtained.

Tables 5-6 attempt to characterize the "precision" of the inference systems (differences in the total number of argument positions in a program between tables 5 and 6 arise from differences in the set of predicates considered to be "builtins" by the two mode inference systems). Table 5 gives the precision of the MA^3 system, in terms of the 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 6 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

Benchmark	**Analysis Time** T_1	**Total Compile Time** T_2	T_1/T_2
asm	63.70	96.22	0.66
boyer	26.01	45.22	0.58
browse	33.32	40.32	0.83
func	38.20	55.14	0.69
peephole	23.45	40.32	0.58
preprocess	79.84	102.17	0.78
projgeom	3.70	6.83	0.54
queens	2.86	5.92	0.48
read	64.23	82.67	0.78
serialize	4.35	7.44	0.58

Table 3: 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 4: Ms Compile vs. Analysis times (secs, using SB-Prolog 2.3.2, Sun 3/50)

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 5: 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 6: Precision of the Ms system.
TAP = Total # of argument positions; IAP = # of "interesting" arg. positions.

within a term, etc. As a result, there are two sources of imprecission: (*i*) the inability to reason about "free" arguments; and (*ii*) lack of information about term structures. In an attempt to distinguish between the 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 lower absolute precision figures.

4 An Application: And-parallelism Detection

This section discusses the application of mode inferencing to the generation of Independent/Restricted And-parallelism [10, 15, 14], an efficient type of parallelism in which only independent goals are executed in parallel and one of the main applications of the MA^3 system. 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. This application is presented as a specific example of the usefulness of the information obtained from global flow analysis.

The parallelization process is herein viewed as a transformation of the original Prolog program into an &-Prolog [15, 13] program which contains (possibly conditional) parallel conjunctions of goals. Although &-Prolog supports several types of parallelizing expressions the discussion is herein limited for conciseness to the generation of *Conditional Graph Expressions* (CGEs) [15]. CGEs are a mechanism (derived from DeGroot's ECEs [10]) for the generation and control of and-parallelism. CGEs can 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. Chang et al. [3], DeGroot [11], Jacobs and Langen [16], and Warren, Muthukumar, Rossi, and Hermenegildo [15, 14, 27], among others, 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, and how to determine conditions for independence for each group. 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 given by the conjunction [15, 14]:

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

Bench.	#CGE	Ovhd.	% modes inferred	Usefulness of Abs. Int.			
				% uncond. cge		*checks/cge*	
				w/o ai	**w ai**	**w/o ai**	**w ai**
AVG	N/A	38.9	83.34	9.31	52.2	3.0	0.74
asm	123	33.3	81.4	27.6	47.2	1.6	0.8
boyer	10	30.1	55.0	30.0	60.0	2.3	1.6
browse	9	65.2	78.7	0	44.4	2.2	0.5
matrix	3	38.3	82.3	0	33.3	4.7	0.6
peephole	27.0	25.6	91.6	0	70.4	4.2	0.4
projgeom	4.0	36.0	85.2	0	50.0	4.5	1.0
queens	7	30.2	100.0	14.3	71.4	2.5	0.4
read	42	48.1	89.3	11.9	59.5	2.2	0.8
serialize	3.0	43.3	86.6	0	33.3	3.0	0.6

Table 7: Performance of the abstract interpreter and annotator

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 existential variables are always unbound [14]. However, local analysis proves to be of relatively limited utility. 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, CGE checks can be significantly reduced and sometimes eliminated altogether at compile time through partial evaluation with the mode information.

Table 7 summarizes some of our experiments in applying inferred mode information to CGE generation. The results correspond to the "MEL" annotation algorithm [27], coupled with MA^3. The table shows for each benchmark the number of CGEs generated, the fraction (overhead) added by the global analysis time to the actual compilation time, the percentage of modes inferred, the percentage of unconditional CGEs generated (i.e. for which no run-time checks are needed), and the average number of checks per CGE. A new benchmark (*matrix*, a matrix multiplication program) is also shown in this table. The "Ovhd." figures given in this table represent actual overhead, i.e. the percentage of time added to compilation by global analysis (as opposed to the fraction of compilation time represented by the analysis). The reader may note that these figures are also lower than those given in the previous section. This is due to the fact that in this section the global analyzer is measured while embedded within the &-Prolog compiler, while for the measurements in the previous section the analyzer was extracted from the &-Prolog compiler and run standalone on top of Quintus Prolog, in order to make comparison with the Ms system more meaningful. The last two columns are given with and without abstract interpretation for comparison. The number of checks per CGE is significantly reduced when global analysis is applied and in a good number of cases unconditional CGEs are generated (i.e. CGEs with no checks), resulting in parallel execution with no independence detection overhead. It can be seen that only a minor improvement of these results would make it feasible to avoid run-time checks altogether by simply generating parallel code for unconditional CGEs and sequential code (rather than a CGE) for the conditional ones (as proposed in the "UDG" annotation method proposed in [27]). The usefulness of global flow analysis in this application is therefore clear. In fact, the results presented in Table 7 represent *lower bounds* on CGE optimization and are expected to improve as our analysis and parallelization tools, which are not directly the subject of the paper, mature. Most significantly, the results presented are based on MA^3 *inferring term groundness only.* Recent results [26, 17] show that it is possible to infer both groundness and *independence* information with a high degree of accuracy. This and other refinements should continue to optimize the parallelization process, 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. In addition, the same techniques can be applied to the generation of other types of (non CGE-based) execution graphs as supported by &-Prolog and 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 at compile-time multiple binding environment maintenance overheads in OR-parallel systems, thus extending the usefulness of this application of global flow analysis.

5 Conclusions

Global flow analysis offers information which can be 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 global dataflow analyses need not be too expensive computationally to be practical. 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, and showed positive results from applying the information generated by MA^3 to the problem of avoiding run-time checks in independent and-parallelism. 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.

References

[1] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.

[2] M. Carlsson. *Sicstus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[3] J.-H. Chang and Alvin M. Despain. Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis. In *International Symposium on Logic Programming*, pages 10–22. IEEE Computer Society, July 1985.

[4] M. Codish. Personal communication, July 1986.

[5] J. S. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Norwell, Ma 02061, 1987.

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

[7] S. Debray. The SB-Prolog System, Version 2.3.2: A User's Manual. Technical Report 87-15, Dept. of Computer Science, University of Arizona, March 1988.

[8] S. K. Debray. A Simple Code Improvement Scheme for Prolog. In *Sixth International Conference on Logic Programming*, pages 17–32. MIT Press, June 1989.

[9] S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, pages 207–229, September 1988.

[10] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[11] D. DeGroot. A Technique for Compiling Execution Graph Expressions for Restricted AND-parallelism in Logic Programs. In *Proc. of the 1987 Int'l Supercomputing Conf.*, pages 80–89, Athens, 1987. Springer Verlag.

[12] S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Fourth IEEE Symposium on Logic Programming*, pages 264–272, September 1987.

[13] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*. MIT Press, June 1990.

[14] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[15] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[16] D. Jacobs and A. Langen. Compilation of Logic Programs for Restricted And-Parallelism. In *European Symposium on Programming*, pages 284–297, 1988.

[17] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[18] G. Janssens. *Deriving Run-time Properties of Logic Programs by means of Abstract Interpretation.* PhD thesis, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, March 1990.

[19] N. Jones and H. Sondergaard. A semantics-based framework for the abstract interpretation of prolog. In *Abstract Interpretation of Declarative Languages*, chapter 6, pages 124–142. Ellis-Horwood, 1987.

[20] L. Kale. Parallel Execution of Logic Programs: the REDUCE-OR Process Model. In *Fourth International Conference on Logic Programming*, pages 616–632. Melbourne, Australia, May 1987.

[21] H. Mannila and E. Ukkonen. Flow Analaysis of Prolog Programs. In *4th IEEE Symposium on Logic Programming*. IEEE Computer Society, September 1987.

[22] A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The impact of abstract interpretation: an experiment in code generation. In *Sixth International Conference on Logic Programming*, pages 33–47. MIT Press, June 1989.

[23] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. *Information Processing*, pages 601–606, April 1989.

[24] C. S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming*, 2(1), April 1985.

[25] C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 463–475. Imperial College, Springer-Verlag, July 1986.

[26] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[27] K. Muthukumar and M. Hermenegildo. The DCG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[28] F. Pereira. Prolog Benchmarks. *Prolog Electronic Digest*, 5(56), August 1987.

[29] *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.

[30] J.-C. Tân. Prolog Optimization by Removal of Redundant Trailings. Technical report, Dept. of Computer Science, National Taiwan University, Taipei, April 1989.

[31] A. Taylor. Removal of dereferencing and trailing in prolog compilation. In *Sixth International Conference on Logic Programming*, pages 48–60. MIT Press, June 1989.

[32] A. Taylor. LIPS on a MIPS: Results from a prolog compiler for a RISC. Technical report, Association for Logic Programming, June 1990.

[33] A. K. Turk. Compiler Optimizations for the WAM. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 657–662. Imperial College, Springer-Verlag, July 1986.

[34] P. Van Roy and A. M. Despain. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. In *Proceedings of the North American Conference on Logic Programming*, pages 501–515. MIT Press, October 1990.

[35] D. H. D. Warren. OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science. Springer-Verlag, March 1987.