Automatic Mode Analysis For Concurrent Logic Programs: Implementation and Evaluation

B. Massey and E. Tick

CIS-TR-92-09
April 1992

Abstract

Mode analysis of logic programs is a fundamental tool for increasing the efficiency of compiled code. Dataflow analysis and abstract interpretation have been the traditional mode analysis techniques, but Ueda and Morita recently introduced an innovative method based on *paths*. Intuitively, moding the paths of a program involves statically finding all subterms that are of interest during execution and determining the modes of these subterms. This paper describes the first implementation of an algorithm for automatic mode analysis based on paths, along with empirical measurements of its practicality. Our implementation differs significantly from the graph propagation method originally suggested, and we show that our complexity is potentially quite low, making the method practical.

Department of Computer and Information Science
University of Oregon

1 Introduction

Mode information has been shown to be quite useful in the efficient compilation of logic programming languages. Primarily, mode information facilitates the strength reduction of unification operators into matches and assignments. There are numerous methods for automatic derivation of mode information from logic programs, e.g., [3, 4, 1, 6]. In committed-choice logic programs [7], the logic variable is overloaded to perform synchronization. Mode information can thus be used to optimize code generated for argument matching. We are particularly interested in mode analysis because it enables us to do static "sequentialization" analysis [5]. This analysis determines if automatic sequentialization of body goals is safe, i.e., cannot result in deadlock. The potential advantage of such a scheme is vastly improved efficiency in register allocation and procedure call protocol.

In Prolog and Parlog, though not in FCP or FGHC, the programmer may supply "argument modes." Intuitively, an input or output argument mode indicates whether a data value is required as input or will be produced as output. Automatic mode generation has the advantage of avoiding programmer error in declaring modes, and can lead (as in the algorithm presented here) to a much richer set of derived information. For example we may wish to derive the mode of *each variable in the syntactic program*, or perhaps modes of objects that are not explicitly mentioned.

Ueda and Morita [8] outlined a mode analysis technique that showed great elegance and potential for efficiently deriving rich mode information. Essentially they suggested propagating a set of mode constraints around a control-flow graph representing the program. The analysis was restricted to "moded" flat committed-choice logic programs, although we do not regard this as a major drawback. Our work is a clarification of their method, a description of an algorithm and its implementation, and evaluation of the method's performance. This paper is the first reported implementation of this kind of mode analysis technique, although we do not use graph propagation. We also present empirical results concerning the mode characteristics of a set of FGHC benchmarks.

The paper is organized as follows. Section 2 reviews the path concept and describes the procedure by which interesting paths are generated. Section 3 describes the fundamental data representation used in mode analysis. Section 4 describes the mode analysis axioms and their incorporation into an algorithm, and discusses the algorithm's complexity, correctness, and completeness. The relationship to other work is summarized in Section 5. Section 6 presents empirical measurements characterizing the performance of the algorithm. We summarize our conclusions and future research in Section 7.

2 Path Generation

The first stage of our algorithm generates a finite set of paths whose modes are to be considered. We adopt Ueda and Morita's notion of "path" as follows: A path p "derives" a subterm s within a term t (written $p(t) \vdash s$) iff for some predicate f and some functors $a, b, \ldots$ the subterm denoted by descending into t along the sequence $\{< f, i >, < a, j >, < b, k >, \ldots\}$ (where $< f, i >$ is the i^{th} argument of the functor f) is s. A path thus corresponds to a descent through the structure of some object being passed as an argument to a function call. We refer to f as the "principal functor" of p. A program is "moded" if the modes of all possible paths in the program are consistent, where each path may have one of two modes:

> *Definition*: If a path has *output mode*, any variable derived by that path *may* (but need not) be bound only within the body of the path's principal functor. The path *will never* derive a constant or a variable which is bound in a caller of the principal functor. □

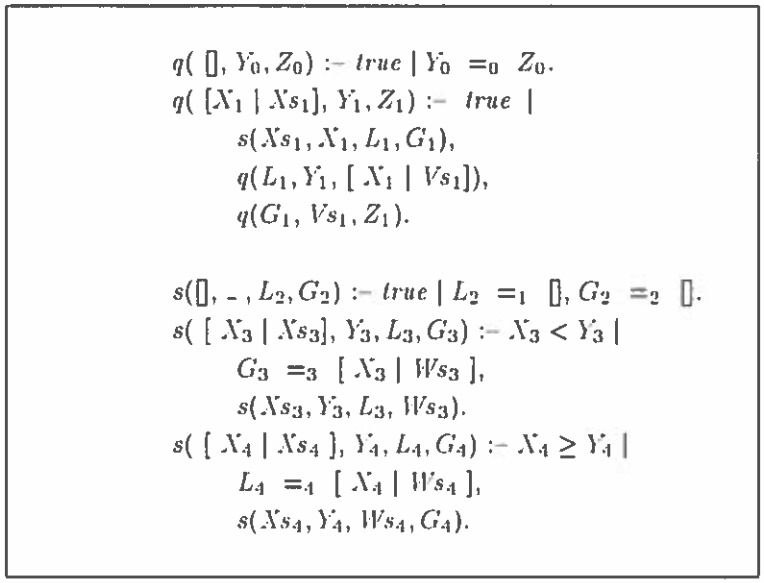
> *Definition*: If a path has *input mode*, any variable derived by that path *may* (but need not) be bound only by callers of the path's principal functor. The path *will never* derive a constant or a variable which is bound by the principal functor itself. □

Only "interesting" paths are generated in the first stage of the algorithm. There are three classes of interesting paths. The first class consists of paths that directly derive a named variable in the head, guard, or body of some clause. All such paths can be generated by a simple sequential scan of all heads, guards, and body goals of the program.

The second class consists of paths which derive a variable v in some clause, where a proper path through the opposite side of a unification with v derives a variable v'. More formally, consider a unification operator $v = t$ where v is a variable and t is some term other than a variable or ground term. Let v' be a variable appearing in t at path q, i.e., $q(t) \vdash v'$. Then if p is a path deriving v (by which condition p is also interesting), then the concatenated path $p \cdot q$ is also an interesting path. All paths in this second class may be generated by repeated sequential scanning of all unification goals until no new interesting paths are discovered. The necessity for repeated scans is illustrated by such clauses as

$$a(X, Z) :- Y = c(X),\ Z = b(Y).$$

where the interesting path $\{< a, 2 >, < b, 1 >, < c, 1 >\}$ given by the first unification body goal will not be generated until the interesting path $\{< a, 2 >, < b, 1 >\}$ in the second unification body goal is generated. Such repeated scans should occur infrequently



$q([], Y_0, Z_0) :- true \mid Y_0 =_0 Z_0.$
$q([X_1 \mid Xs_1], Y_1, Z_1) :- true \mid$
$s(Xs_1, X_1, L_1, G_1),$
$q(L_1, Y_1, [X_1 \mid Vs_1]),$
$q(G_1, Vs_1, Z_1).$

$s([], \_, L_2, G_2) :- true \mid L_2 =_1 [], G_2 =_2 [].$
$s([X_3 \mid Xs_3], Y_3, L_3, G_3) :- X_3 < Y_3 \mid$
$G_3 =_3 [X_3 \mid Ws_3],$
$s(Xs_3, Y_3, L_3, Ws_3).$
$s([X_4 \mid Xs_4], Y_4, L_4, G_4) :- X_4 \geq Y_4 \mid$
$L_4 =_4 [X_4 \mid Ws_4],$
$s(Xs_4, Y_4, Ws_4, G_4).$

Figure 1: Quicksort FGHC Program: Weak Canonical Form (Clauses 0–4)

in practice. In any case not more than a few scans are necessary — no greater number than the syntactic nesting depth of expressions containing unification operators.

The third class of interesting paths is generated by noting that if a path starting on the right-hand side of a unification body goal (i.e., a path of the form $\{<=, 2>\}\cdot s$) is interesting, then so is the corresponding path starting on the left-hand side of that unification (i.e., $\{<=, 1>\}\cdot s$).

As an example of path generation, consider a Quicksort program[1] written in FGHC, shown in Figure 1. The program shown is the result of applying some syntactic transformations to produce a weak canonical form required by the subsequent algorithms. The left-hand side of every unification operator = /2 has been made a variable. No "extra" variables appear in unification body goals: all variables appearing in a unification body goal must also appear in some non-unification body position or in the head. All variables are qualified by the clause number in which they appear, in order to retain scoping information for subsequent passes. In addition, each unification operator is labeled with a unique integer.

In general, all interesting paths of a program are generated in a few sequential passes. The 39 interesting paths of Quicksort, as shown in Table 1, are generated in two passes. In this example the unification body goals provide four additional interesting paths during the first pass, and no additional interesting paths during the second pass.

[1] We use this program as an example throughout the paper.

input		output	
user	builtin	user	builtin
$\{< s/4,1 >\}$	$\{<\text{`}<\text{'}/2,2 >\}$	$\{< q/3,2 >\}$	$\{<=_0,1 >\}$
$\{< s/4,2 >\}$	$\{<\text{`}<\text{'}/2,1 >\}$	$\{< s/4,3 >\}$	$\{<=_1,1 >\}$
$\{< q/3,1 >\}$	$\{<\text{`}\geq\text{'}/2,2 >\}$	$\{< s/4,4 >\}$	$\{<=_2,1 >\}$
$\{< q/3,3 >\}$	$\{<\text{`}\geq\text{'}/2,1 >\}$	$\{< s/4,3 >,< .,1 >\}$	$\{<=_3,1 >\}$
$\{< q/3,1 >,< .,1 >\}$	$\{<=_0,2 >\}$	$\{< s/4,3 >,< .,2 >\}$	$\{<=_4,1 >\}$
$\{< q/3,1 >,< .,2 >\}$	$\{<=_1,2 >\}$	$\{< s/4,4 >,< .,1 >\}$	$\{<=_3,1 >,< .,1 >\}$
$\{< q/3,3 >,< .,1 >\}$	$\{<=_2,2 >\}$	$\{< s/4,4 >,< .,2 >\}$	$\{<=_3,1 >,< .,2 >\}$
$\{< q/3,3 >,< .,2 >\}$	$\{<=_3,2 >\}$		$\{<=_4,1 >,< .,1 >\}$
$\{< s/4,1 >,< .,1 >\}$	$\{<=_4,2 >\}$		$\{<=_4,1 >,< .,2 >\}$
$\{< s/4,1 >,< .,2 >\}$	$\{<=_3,2 >,< .,1 >\}$		
	$\{<=_3,2 >,< .,2 >\}$		
	$\{<=_4,2 >,< .,1 >\}$		
	$\{<=_4,2 >,< .,2 >\}$		

Table 1: Interesting Paths of Quicksort (23 input, 16 output)

An important question is whether this set of paths represents a *minimal and complete* set of paths for the mode analysis. The answer to this may depend upon the use to which the mode analysis is put — nonetheless, as discussed in Section 4.6, there is good reason to believe that some fundamentally important paths are currently not generated. However, in the benchmarks examined here the paths generated by the above algorithm prove to be almost entirely sufficient (see Section 6).

3 The Partition Table

The mode information for paths is derived from knowledge of the relationships between paths. Thus, a systematic and efficient means of keeping track of these relationships as analysis proceeds is required. The computation of path relations is optimized by partitioning the paths such that all the paths in each partition have the same mode. First, each path is placed in a unique partition, indicating that its mode is unknown and that its relationship with the modes of other paths is unknown. There are also two special partitions which are initially empty: the *input* and *output* partitions. Various relationships between the partitions are then asserted by predicates as we proceed through the analysis.[2] In order of decreasing precedence, the predicates are:

- Predicate $in(p)$ asserts that the path p must have input mode. First, if the partition S containing p is the output partition, the assertion is reported as con-

[2]This does *not* imply that we use an `assert` builtin in a logic programming implementation!

tradictory and ignored. If S is the *input* partition, the assertion is a tautology, and is ignored. Otherwise, the partition containing p is merged with the *input* partition, and this information is propagated across all lower precedence relations between paths previously asserted. If a contradiction is discovered at any point during propagation, the assertion is reported as contradictory and ignored. Predicate *out*(p) asserts that the path p must have output mode — this case is analogous to, and identical in precedence to, the *in*(p) case.

- Predicate *same*(p,p') asserts that p and p' must have the same mode. Let the partitions S and S' contain p and p' respectively. If S and S' are identical, the assertion is a tautology and is ignored. Otherwise, the two partitions are merged, and the result is propagated across all lower precedence relations previously asserted. If a contradiction is discovered, an error is reported and the assertion is ignored.

- Predicate *opposite*(p,p') asserts that p and p' must have inverse modes. Let partitions S and S' contain p and p' respectively. If both paths are in the same partition, an error is reported and the assertion is ignored. Otherwise, the relationship between the partitions is recorded.

The data structure used to record these relationships is known as the *partition table*, and the mode analysis is merely a sequence of partition-table updates. Note that the partition table may be partially pre-initialized. Sources of such information include user mode declarations and previous mode analysis of modules related to the module being analyzed.

4 Mode Analysis

The second stage in the algorithm is to derive the modes of paths generated by the first stage. This is accomplished by finding absolute modes for a small number of paths and then examining relationships between the modes of paths. This mode analysis exploits the rules outlined by Ueda and Morita. Their axioms are clarified and reformulated in Figure 2 [3] [5]. Again the algorithm repeatedly scans sequentially through the program, this time deriving modes of paths. The critical insight is that given a variable v, the modes of all paths deriving v must be related via the mode axioms. The mode analysis algorithm proceeds in four steps:

1. Assert absolute modes for some paths (Section 4.1).

[3] In Figure 2, $m(p)$ means the mode of path p

§1. For some path p in a clause, $m(p)$ = in, if either

1. p leads to a non-variable in the head or body, or
2. p leads to a variable which occurs more than once in the head, or
3. p leads to a variable which also occurs in the guard at path p_h and $m(p_h)$ = in

§2. Two arguments of a unification body goal have opposite modes, for all possible p, or more formally: $\{\forall p\ \mathtt{m}(<=,1>\mathtt{p}) \neq \mathtt{m}(<=,2>\mathtt{p})\}$.

§3. If there are exactly two "occurrences," we have two possibilities:

1. If both occurrences are in the body, the modes of their paths are inverted.
2. If there is one (or more) occurrence in the head and one in the body, the modes of their paths are the same.

§4. If there are more than two "occurrences" of a shared variable (i.e., at least two occurrences in the body), the situation is even more complex:

1. If the body contains more than two occurrences of the shared variable and the head has no occurrences, then one of the modes is 'out,' and the others are 'in.'[a]
2. If the head contains one (or more) occurrences of the shared variable (so the body has two or more occurrences), then the modes are as follows:
   (a) If the mode of the head occurrence is 'in,' the modes of all body occurrences are 'in' as well.
   (b) If the mode of the head occurrence is 'out,' then *one* of the body occurrences is 'out,' and the other body occurrences are 'in.'

[a]This means that one of the occurrences is designated as the producer of this variable.

Figure 2: Ueda and Morita's Mode Derivation Axioms

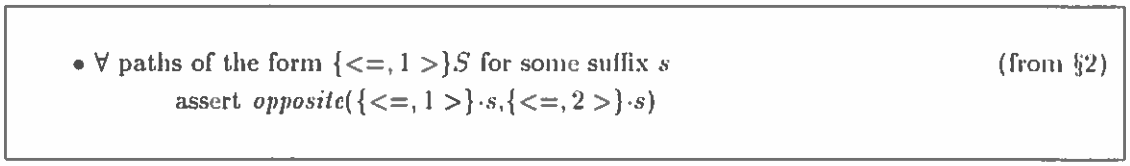


Figure 3: Unification Analysis Algorithm

2. Assert that all paths on opposite sides of a unification operator have opposite modes (Section 4.2).

3. Proceed sequentially through the variables derivable from interesting paths, asserting all binary relations between paths (Section 4.3).

4. Repeatedly consider multiway relations asserted by the clauses (Section 4.4).

4.1 Syntactic Analysis

During the first step in mode analysis a single syntactic pass is made over the program, noting paths which lead to constants, variables, and guard arguments. We record all occurrences of each variable in the module being analyzed which are derivable from each interesting path; information which will be used by all succeeding steps of the algorithm. Whether the variable occurrence was in the head, guard, or body of its clause is also recorded. Constants in interesting path positions are noted and the partition table is updated according to §1.1 of Figure 2. This will generally be a rich source of information about *in* paths. In Quicksort, this fixes the modes of paths such as $\{< q, 1 >\}$ as *in*. The modes of paths leading to non-unification guard arguments in the partition table are then asserted according to §1.3 of Figure 2. In Quicksort, this fixes the modes of paths such as $\{<\text{`}\geq\text{'}, 1 >\}$ as *in*.

4.2 Body Unification Analysis

This step simply asserts that corresponding paths on opposite sides of a body unification goal have opposite modes. This relationship corresponds to §2 of Figure 2, and is implemented according to the algorithm shown in Figure 3. It is generally efficient to assert these relationships early, since it allows greater propagation of information asserted by later steps. For example, in Quicksort clause #0 $opposite(\{<=_1, 1 >\}.\{<=_1, 2 >\})$ is asserted.

Note the universal quantifiers in the algorithms of Figures 3, 4, and 5. Our implementation of these depends on the fact that a *finite* (and indeed a small) set of paths are generated for the target program, as contrasted with the work described in Section 5.

```
• ∀ variables v occurring more than once in a head position          (from §1.2)
        ∀ paths p deriving v
                assert in(p)

• ∀ variables v
  if v occurs exactly twice in a clause at paths p and p'
  (counting all head occurrences as one) then {
        if both occurrences are in the body then {                   (from §3.1)
                assert opposite(p,p')
                ∀ suffixes s  s.t.  p·s is interesting and p'·s is interesting
                        assert opposite(p·s,p'·s)
        } else {                                                     (from §3.2)
                assert same(p,p')
                ∀ suffixes s  s.t.  p·s is interesting and p'·s is interesting
                        assert same(p·s,p'·s)
    }   }
```

Figure 4: Binary Analysis Algorithm

4.3 Binary Analysis

This step derives the modes of paths which have binary relations. These relationships correspond to §1 and §3 of Figure 2. These rules are implemented according to the algorithm shown in Figure 4. Note that the ordering of operations of this algorithm is somewhat arbitrary. This particular ordering was chosen both for efficiency and ease of implementation, but it is not unlikely that some other order could be faster or simpler. In particular, each rule is currently applied in turn to all of the applicable objects in the program. Reversing the nesting order so that all possible rules are applied to each syntactic object in turn would not affect the correctness of the algorithm and might provide some speed increase.

For Quicksort clause #0 in the previous step $opposite(\{<=_1,1>\}.\{<=_1,2>\})$ was asserted.
$same(\{<=_1,1>\},\{<q,2>\})$ and $same(\{<=_1,2>\}.\{<q,3>\})$ are now asserted. The database thus automatically concludes that $opposite(\{<q,2>\}.\{<q,3>\})$, a fact used in subsequent analysis.

4.4 Multiway Analysis

Once all the consequences of binary relationships between paths in the program have been established, if there are still interesting paths in the partition table whose modes are ambiguous, they may be resolved by applying the multiway rule §4. It would be possible in principle to do this analysis in the same way as for the previous rules, establishing constraints between partitions. However, several factors mitigate against this. First, at this point in the analysis, it is expected that there will be few partitions to consider in a typical program, so the efficiency gain of the database-driven approach is relatively unnecessary. Secondly, the non-binary constraints of the multiway rule make database constraint maintenance and propagation *much* more difficult. Finally, it is difficult to obtain any better output representation in ultimately ambiguous cases than an enumeration or summary of possibilities. Thus there is no strong motivation at this point for clever analysis.

Therefore the multiway rule is implemented according to the recursive generate-and-test algorithm of Figure 5. Starting with the partition table output by binary analysis, the algorithm examines all possible values for each set of mutually constrained paths generated by occurrences of a variable meeting the conditions of §4 of Figure 2. Each of these possibilities is tested by applying the multiway algorithm recursively to the remaining constraints, and the resulting collections of partition tables are merged and returned. Thus, the overall structure of the call graph of the test algorithm is a tree whose leaves are each either a partition table or a failure indication — the output of the test algorithm is simply the collection of partition table leaves of the tree. Note that the shape of the tree is determined only by the output of the iterative generate algorithm: thus, the recursive test function could easily be made iterative via standard transformations. Note also that since our implementation of this algorithm is in FGHC, subtrees will naturally be evaluated concurrently in a parallel implementation of the language.

In Quicksort only clause #1 meets the multiway criteria, where X_1 occurs once in the head and twice in the body. The three possibilities to be checked are summarized in Table 2. By this point in the analysis, however, we have already derived that $\{< q, 1 >, < ., 1 >\}$ is *in* and that $\{< s, 2 >\}$ is *in*, so we find that $\{< q, 3 >, < .., 1 >\}$ is *in*. This example is nice because it is completely determinate — the multiway rule derives only one set of possible modes for the program. In general, this may not be the case and several possible final modings for the program will be emitted.

```
multiway( V, t ) {          — V is set of variables, t is partition table
    test( generate( V ), t )
}
generate( V ) {
    R = ∅                   — R is set of tuples (v,Q): v is variable, Q is set of paths
    ∀v ∈ V s.t. v occurs ≥ 3 times in a clause
        let p_0...p_n be the paths deriving v
        ∀ suffixes s  s.t.  a proper subset Q of {p_0 · s...p_n · s} is interesting and |Q| ≥ 2
            R = R ∪ {(v, Q)}
    return( R )
}
test( R, t ) {
    if R = ∅ then
        return( {t} )
    select some (v,Q) from R
    R' = R \ {(v, Q)}
    if ∃p ∈ Q s.t. some prefix of p ⊢ v in a head position {
        T = test( R', update( t, input( Q ) ) )    — T is set of partition tables
        ∀p' ∈ Q \ {p}
            T = T ∪ test(R', update( t, output( {p, p'} ), input( Q \ {p, p'} ) ) )
    } else {
        T = ∅
        ∀p ∈ Q
            T = T ∪ test( R', update( t, output( {p'} ), input( Q \ {p'} ) ) )
    }
    return( T )
}
```

Figure 5: Multiway Analysis Algorithm

	path		
	$\{< q,1>,< .,1>\}$	$\{< s,2>\}$	$\{< q,3>,< ..1>\}$
1	in	in	in
2	out	in	out
3	out	out	in

Table 2: Possible Modes For X_1

4.5 Complexity

The complexity of the algorithm can best be understood by examining its component pieces. Everything up to the beginning of binary analysis is fundamentally linear on the length of the program — a small fixed number of passes are made over the program to derive facts about it. The binary analysis is also close to linear on the number of variables in the program meeting the constraints of Figure 4. A significant quadratic component derives from the fact that inner loops of the analysis iterate over a set of suffixes of paths — the size of this set is approximately linearly proportional to some measure of the "complexity" of the program.

The multiway analysis is difficult to analyze. If it were performed first, it would be exponential in the number of variables meeting the constraints of Figure 5, but by the time it is actually performed, most alternatives contradict the known modes, and thus are not explored further. In practice, as shown by the timings in Section 6, the time spent in this analysis seems to be reasonably short.

4.6 Consistency, Completeness, and Safety

Some important practical and theoretical issues are raised by these algorithms. Some of these issues include the consistency, completeness, and safety of the mode analysis. It is not difficult to prove that the mode analysis algorithm is consistent, in the sense that if at some point in the analysis, path p is shown to have mode m, and if some subset of the interesting paths implies that p does not have mode m, then the algorithm will derive and report this contradiction. However, this consistency property is less useful than is desired, for several reasons. The first is quite simple — if the algorithm does report a contradiction, there is no obvious way to automatically correct it, or even to determine the minimal subset of paths involved in the contradiction. It becomes entirely the user's responsibility to correct the program so that it is consistently moded.

The current implementation will report any contradiction, ignore the contradictory assertion, and proceed with the derivation. This allows the user to examine the final modes produced by the analysis and determine which might be incorrect. In our experience, this is usually sufficient to correct the problem. In the absence of user intervention, this also in practice allows the modes of most of the remaining paths to be determined. For example, when using the mode information for sequentialization [5], we may sequentialize all calls not involving a conflicting path, and then "safely" compile calls involving the conflicting path.

The second weakness in this form of consistency is more subtle: the non-modedness of a program may not be detectable if the analysis uses the wrong set of paths! This leads directly to a reasonable definition of a *complete* set of paths:

Definition: A set of paths generated for a program is *complete* iff the existence of a consistent moding for the set of paths implies that the program is fully moded. □

Thus, the infinite set of all possible paths is a complete set; however, we are really interested in finite complete sets and in particular in a *minimal* complete set of paths for the program. As an example of the incompleteness of our path generation algorithm, consider

$g(X)$:- *true* | $Y = [\ 1\ ]$, *aux1*(Y, X).
$h(Y)$:- *true* | $g(Y)$, *aux1*$(Y, 2)$.
aux1($[\ X\ |\ \_\ ]$, Y) :- *true* | $Y = X$.

With the path generation as described, the path $\{< g, 1 >, < ., 1 >\}$ will never be considered. But without noticing that this path is *out*, it is impossible to discover that the path $\{< aux1, 2 >\}$ is *out* and thus derive the non-moded call in $h/1$.

It is difficult to extend path generation in such a way as to obtain a finite complete set of paths, much less a minimal one. Our current plan for doing so involves implementing an abstract interpreter over the domain of paths of maximum length k, where k is the length of the longest path generated by the naive algorithm described above. Certainly a naive depth-k bound must be avoided since it has exponential complexity.

Note that the path generation algorithm previously described is *unsafe*. It is also a consequence of the incomplete set of generated paths that even if the program contains information about the mode of a path, that information may not be derived by the mode analysis algorithm. Nonetheless, most generated paths in typical programs are moded by this analysis, and if the program being analyzed is known to be moded, all modes derived are correct.

One may thus use the mode information derived by the algorithm as advisory information; alternatively, one may insist that the input program be moded correctly. If incomplete mode information is derived, the user may explicitly supply the missing information by inspection. This is the approach currently taken in our benchmark analysis. Finally, in some cases the modes of some paths simply cannot be determined because they depend on the modes of the query itself. In these cases, the programmer may explicitly supply query modes to the analyzer by preloading the partition table.

5 Comparison With Other Work

The original mode analysis scheme is due to Ueda and Morita [8]. Instead of limiting the domain of analysis to a finite set of paths, they define the rules over the closure of

all possible paths. The unique feature of the path generation is that it uses a graph-based representation of paths, where the directed graph represents a set of prefixes of possible paths. No algorithm is given, however, for deriving the directed graph of the target program. An examination of their sample directed graphs indicates that any automated technique would have to solve the same difficulties in path generation discussed in Sections 2 and 4.6. The modes are then propagated over the directed graph according to the mode-analysis rules. This propagation is analogous to the propagation via partition tables described in Section 4. We believe our algorithm is no worse in performance than the graph scheme, and is easier to implement. In fact, our initial attempts to utilize graphs led us to our current algorithm.

The mode analysis rules were simplified to their present form by Korsloot and Tick [5], who also gave a set of simple inference rules for deriving paths and their modes. However, no selection algorithm is given for application of the inference rules. That work still deals with an infinite set of possible paths, but considers only those paths with a finite known prefix. As testament to the inherent difficulty of dealing with inference rules, they failed to consider certain "interesting" paths and thus incompletely moded the Quicksort example!

The algorithm described in this paper is the logical extension of the previous works to a finite domain of paths. We represent the relationships of a finite set of paths in such a way that all mode information directly available about this set of paths in a program may be efficiently derived.

6 Empirical Results

Our experimental implementation of the path generation and mode analysis algorithms consists of approximately 4,000 lines of code comprising 23 modules totaling about 500 clauses, written in FGHC running under the PDSS system [2]. We examined the moding and execution characteristics of the analysis of a group of seven FGHC programs listed in Table 3, including one module (msort) from the implementation itself. The table shows the number and percentage of derived (input and output) paths, as well as the number of paths that could not be derived. The benchmarks averaged 3.3% of non-derivable paths, with some variance. As mentioned in Section 4.6, there are two approaches which appear viable for reducing the percentage of non-derivable paths. One method is to generate a richer set of paths to drive the mode analysis. Currently, we generate only "simple" paths, which are local to each clause. This can be extended to more elaborate generation, effectively abstract interpretation on a domain of "important" paths.

benchmark	# procs	# clauses	paths			explicit modes needed
			input	output	not derived	
qsort	3	6	23 (59.0)	16 (41.0)	0 (0.0)	0
msort	4	11	40 (52.6)	32 (42.1)	4 (5.3)	1
prime	7	12	37 (57.8)	27 (42.2)	0 (0.0)	0
cubes	9	16	79 (56.0)	62 (44.0)	0 (0.0)	0
pascal	11	22	68 (60.2)	45 (39.8)	6 (5.3)	2
waltz	21	54	138 (61.1)	88 (38.9)	28 (12.4)	7
triangle	42	80	645 (88.4)	85 (11.6)	0 (0.0)	0
average			(62.2)	(37.1)	(3.3)	

Table 3: FGHC Benchmarks: Path Analysis with No Explicit Information

A second method, illustrated in these measurements, is to have the programmer give explicit mode declarations to help the analysis. The final column of Table 3 gives the number of explicit path modes needed to permit the derivation of all paths. Note that only a fraction of the unknown paths are needed to fully constrain those remaining. Waltz has the largest requirement because its data structure manipulation is far more complex than that of the other programs. Local path generation does not suffice because deeply nested subterms are decomposed through chains of procedure invocations. Five declarations for Waltz state the seven explicit paths needed to uncover all 28 unknown paths:

:- *mode spawn*(_, ?, _, _, _).
:- *mode fromLStoList1*(_, [?|_], _, _).
:- *mode group*([?|_], ?, _, _, _, _).
:- *mode genEdges*([?|_], _, ?, _, _, _, _).
:- *mode group*(_, _, _, _, _, _, _, [*edge*(_, _, ?)|_]).

These were easily introduced by hand, although as noted we are actively pursuing a more complete method of path generation to avoid any need for declarations.

Table 4 categorizes the paths by type. User paths are paths defining variables in user-defined procedures (c.f. =/2 paths, which are less interesting, except for assignment and for propagating modes within the analysis). User input paths are further split into 1-paths (top-level formal parameters of procedures) and k-paths for $k > 1$. Assignments are =/2 output 1-paths.

By type, user paths constitute the largest portion with 52%. 15.9% are user output paths, and the user input paths split almost evenly between paths of length one and greater. The length of the k-paths is highly program dependent, although usually the

benchmark	user input		user	builtin		total
	1-paths	k-paths	output	assign	others	
qsort	4 (10.3)	6 (15.4)	7 (17.9)	5 (12.8)	17 (43.6)	39
msort	6 (8.0)	11 (14.5)	11 (14.5)	12 (15.8)	36 (47.3)	76
prime	10 (15.6)	5 (7.8)	13 (20.3)	6 (9.3)	30 (46.9)	64
cubes	22 (16.7)	25 (17.7)	32 (22.7)	7 (5.0)	55 (39.0)	141
pascal	19 (16.0)	13 (10.9)	17 (14.3)	16 (13.4)	54 (45.4)	119
waltz	52 (20.5)	35 (13.8)	39 (15.4)	29 (11.4)	99 (39.0)	254
triangle	155 (21.2)	449 (61.5)	44 (6.0)	37 (5.1)	45 (6.2)	730
average	(15.5)	(20.2)	(15.9)	(10.4)	(38.2)	

Table 4: Breakdown of Paths by Type: Raw Counts and (Percentages)

benchmark	path creation + syntactic analysis	mode analysis			total	paths	paths/sec
		unification	binary	multiway			
qsort	0.62 (41.9)	0.11 (7.4)	0.72 (48.6)	0.03 (2.0)	1.48	39	26.4
msort	2.09 (44.8)	0.27 (5.8)	2.12 (45.5)	0.18 (3.9)	4.66	76	16.3
prime	1.39 (43.4)	0.22 (6.9)	1.45 (45.3)	0.14 (4.4)	3.20	64	20.0
cubes	5.59 (37.3)	0.98 (6.5)	5.37 (35.8)	3.05 (20.3)	14.99	141	9.4
pascal	5.06 (39.7)	0.64 (5.0)	6.12 (48.0)	0.92 (7.2)	12.74	119	9.3
waltz	27.23 (40.5)	2.69 (4.0)	26.79 (39.9)	10.46 (15.6)	67.17	254	3.8
triangle	175.09 (32.1)	19.48 (3.6)	290.20 (53.1)	61.37 (11.2)	546.14	730	1.3
average	(40.0)	(5.6)	(45.2)	(9.2)			

Table 5: Execution Time for Analysis: Raw Seconds and (Percentages)

average length is close to two. Assignments constitute 10.4% of all paths. Surprisingly, a large percentage (38%) of the paths are builtin paths needed only for mode propagation during analysis.

Table 5 gives the execution times of the analysis on a Sparcstation II under PDSS. Unfortunately, this preliminary implementation suffers because the path-table access functions are linear in the number of generated paths. This in turn worsens the performance of most of the algorithms from linear to quadratic. We expect the implementation to be approximately linear on the number of paths after the re-implementation of our path table e.g., as a hash table. Furthermore, PDSS is an emulation-based system, and the timings include the full impact of frequent garbage collections.

However, these measurements do indicate the approximate *relative* weight of each phase of the analysis. We see that the multiway rule, although potentially exponential, is in practice quite cheap. Almost all of the computation (85%) arises from path generation and binary mode analysis. Complex programs show significant (11–20%) multiway analysis. The last column of the table estimates program complexity by the

metric of paths analyzed per second. As explained above, the current performance is quadratic (i.e., paths^2/second is linear in Table 5), which in fact confirms that we can linearize the implementation.

7 Conclusions and Future Research

This paper describes an implementation of an innovative compile-time path generation and mode analysis technique for committed-choice languages. We have shown that the analysis can be implemented efficiently by first generating a small set of "interesting" paths, and then moding the paths according to the rules suggested by Ueda and Morita. By acting on multiway relations last, we avoid exponential problems. Most of the computation occurs in path generation and binary mode analysis. Characteristics of FGHC benchmarks show that the algorithm behaves efficiently, moding all but 3.3% of interesting paths. We give the static frequency of path type occurrences, which is useful information for language implementors.

Future research includes two major targets. First, we will explore methods of path generation that further increase the ratio of derivable paths. Second, we plan to implement sequentialization analysis [5] on top of these algorithms in the context of the Monaco shared-memory multiprocessor FGHC system.

Acknowledgements

This research was supported by an NSF Presidential Young Investigator award, with matching funds generously provided by Sequent Computer Systems Inc.

References

[1] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *International Conference and Symposium on Logic Programming*, pages 669–683. University of Washington, MIT Press, August 1988.

[2] T. Chikayama *et al.* Overview of the Parallel Inference Machine Operating System PIMOS. In *International Conference on Fifth Generation Computer Systems*, pages 230–251, Tokyo, November 1988. ICOT.

[3] S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, July 1989.

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

[5] M. Korsloot and E. Tick. Sequentializing Parallel Programs. In *Phoenix Seminar and Workshop on Declarative Programming*. Sasbachwalden, FGR, November 1991.

[6] C. S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming*, 2(1):43–66, April 1985.

[7] E. Y. Shapiro, editor. *Concurrent Prolog: Collected Papers*, volume 1,2. MIT Press, Cambridge MA, 1987.

[8] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *International Conference on Logic Programming*, pages 3–17. Jerusalem. MIT Press. June 1990.