Practical Static Mode Analyses of Concurrent Logic Languages

Evan Tick
University of Oregon

Miyuki Koshimura
Institute for New Generation Computer Technology

CIS-TR-94-06
March 1994

Abstract

One popular approach to improving the performance of fine-grain concurrent languages is to partition programs into threads. This requires static analysis to determine dependencies between tasks, to avoid placing a cycle within a thread. In the context of concurrent logic programming (CLP) languages, dependency analysis requires *mode analysis*. Simple argument modes are insufficient because dependencies can be hidden within complex terms.

This paper describes and compares four compile-time analysis algorithms, based on seminal work by Ueda and Morita, for deriving the *path modes* of concurrent logic programs. A path describes a subterm of a procedure argument. The analyses are based on constraint propagation over graphs, path partitioning, and model generation theorem proving. All four analyses were implemented in KL1 to allow critical comparisons. We discuss the issues of completeness and complexity, and present empirical performance measurements for a benchmark suite, to determine utility. We show that the time and space requirements of the analysis is comparable to compilation, and that completeness is not problem for the programs studied.

This paper was submitted to the *Journal of Programming Language Design and Implementation.*

Contents

1 Introduction **1**

2 Background: Paths and Modes **3**

3 Constraint Propagation Algorithm **4**

3.1 Data Structures 4

3.2 Graph Creation 8

3.3 Abstract Unification 9

3.4 On Completeness 12

4 Process Network Analyzer **13**

4.1 Distributed Unification 14

4.2 Terminating Unification 15

4.3 Accessing the Graph 15

4.4 Comparison with Static Analysis 15

5 Finite Domain Analysis **16**

5.1 On Completeness 18

6 Mode Inference Method **19**

6.1 Model Generation 19

6.2 Mode Analysis in MGTP 20

6.3 On Completeness 22

7 Performance Comparison **23**

7.1 Execution Measurements 24

8 Summary and Conclusions **29**

References **30**

1 Introduction

Mode information has been shown to be quite useful in the efficient compilation of logic programming languages. In concurrent logic programs, the logic variable is overloaded to perform synchronization. Mode information can thus be used to optimize code generated for argument matching, to avoid suspensions [20]. Pure demand-driven execution, also enabled by mode information, can lead to better resource allocation and minimal work expended [9]. Another optimization that can be driven by mode information is static partitioning of a concurrent logic program into threads of higher granularity, for more efficient multiprocessor execution [6, 8]. Mode information is useful not only for compiler optimization but also for static bug detection. In the latter, the analyzer warns the programmer that variable usage disobeys conventions (discussed below) and is thus likely to be erroneous.

In general, possible variable modes are '*in*' (meaning that a variable will not be bound by the current goal) and '*out*' (meaning that a variable will not be bound outside of the current goal). Traditionally, 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., [1, 3, 4, 11]. Another option is user declarations [2, 14], which we consider either incomplete or too much burden on the programmer.

We are interested in concurrent logic programs in the FCP(:, |) language family [13] that include ask (passive unification for input matching) and tell (active unification for exporting bindings) guards. This represents a broad class of flat committed-choice programs. Figure 1 shows a sort program used throughout the paper to illustrate the analysis techniques discussed herein. A sample query would be: ?- q([2,1,3], y, [ ]), returning y = [3,2,1].

A committed-choice logic program in this family is a set of guarded Horn clauses of the form: "H :- $A_1, \ldots, A_m : T_1 \ldots, T_n \mid B_1, \ldots, B_p$" where $m, n, p \geq 0$. H is the clause head, A_i is an Ask guard goal, T_j is a Tell guard goal, and B_k is a body goal. The ':' operator separates the guard types, and the commit operator '|' divides the clause between the guards and body. If $p = 0$ the clause is called a *unit clause*. A procedure is comprised of a set of clauses with the same principle functor and arity for H. The guards are "flat" when they are system-defined builtins.

Informally, a procedure invocation *commits* to a clause by matching the head arguments (passive unification) and satisfying the guard goals. When a goal can commit to more than one clause in a procedure, it commits to one of them nondeterministically (the others candidates are thrown away). Structures appearing in the head and guard of a clause cause *suspension* of execution if the corresponding argument of the goal is not sufficiently instantiated. A suspended invocation may be *resumed* later when the variable associated with the suspended invocation becomes sufficiently instantiated.

The satisfaction of the ask guards requires matching whereas the satisfaction of the tell guards requires full unification. There are basically two types of tell unification: atomic and eventual. For practical reasons [17, 13] we consider only eventual tell, where in effect the unifications are executed after commit. This corresponds to languages such as Strand and FGHC. Throughout this paper we show only *normalized* programs that explicitly list tell guards to avoid any confusion.

$$\begin{array}{l}
q(T_0, Y_0, Z_0) \mathrel{:-} T_0 \; = \; [\,] \; : \; Y_0 \; =_0 \; Z_0 \mid \mathit{true}. \\
q(T_1, Y_1, Z_1) \mathrel{:-} T_1 \; = \; [X_1 \mid Xs_1] \; : \; U_1 \; =_1 \; [X_1 \mid Vs_1] \mid \\
\quad\quad s(Xs_1, X_1, L_1, G_1), \\
\quad\quad q(L_1, Y_1, U_1), \\
\quad\quad q(G_1, Vs_1, Z_1). \\
\\
s(T_2, \_, L_2, G_2) \mathrel{:-} T_2 \; = \; [\,] \; : \; L_2 \; =_2 \; [\,], G_2 \; =_3 \; [\,] \mid \mathit{true}. \\
s(T_3, Y_3, L_3, G_3) \mathrel{:-} T_3 \; = \; [\, X_3 \mid Xs_3 \,], X_3 < Y_3 \; : \\
\quad\quad G_3 \; =_4 \; [\, X_3 \mid Ws_3 \,] \mid \\
\quad\quad s(Xs_3, Y_3, L_3, Ws_3). \\
s(T_4, Y_4, L_4, G_4) \mathrel{:-} T_4 \; = \; [\, X_4 \mid Xs_4 \,], X_4 \geq Y_4 \; : \\
\quad\quad L_4 \; =_5 \; [\, X_4 \mid Ws_4 \,] \mid \\
\quad\quad s(Xs_4, Y_4, Ws_4, G_4).
\end{array}$$

Figure 1: Quicksort FCP(:, |) Program: Normalized Form (Clauses 1–5)

A program successfully terminates when, starting from an initial user *query* (a conjunct of atoms), after some number of reduction steps, no goals remain to be executed, nor are suspended. Alternatively, the program *deadlocks* if only suspended goals remain. A third result is program *failure*, which in a flat language is an exceptional occurrence that signals program error.

Ueda and Morita [21] proposed a mode analysis scheme for such programs, based on the representation of procedure paths and their relationships as rooted graphs ("rational trees"). Unification over rational trees combines the mode information obtainable from the various procedures. For example, in a procedure that manipulates a list data stream, we might know that the mode of the *car* of the list (that is the current message) is the same mode as the *cadr* (second message), *caddr* (third message), etc. This potentially infinite set of "paths" is represented as a concise graph. Furthermore, a caller of this procedure may constrain the *car* to be input mode. By unifying the caller and callee path graphs, modes can be propagated. The analysis is restricted to "moded" flat committed-choice logic programs. These are programs in which the mode of each path in a program is constant, rather than a function of the occurrences of the path. This is not regarded as a major drawback, since most non-moded flat committed-choice logic programs may be transformed to moded form in a straightforward fashion.

The quicksort program is moded and in fact is *fully moded*, i.e., the modes of all paths are known. Quicksort can be used to illustrate the power of mode analysis. Quicksort can be entirely *sequentialized* (i.e., *partitioned* into a single thread), in the body goal order shown, by using the modes to derive goal dependencies. A sequential implementation can outperform a parallel implementation by better utilizing the underlying architecture by reduced procedure invocation overheads, better register allocation, and other benefits. We wrote an experimental FGHC-to-C compiler in another research project [8] to do this sequentialization, with the result that our compiler-generated C program sorted a list of 500 integers in 2.2 sec. compared to 10.5 sec. on Monaco, a fast parallel FGHC system [18] (on Sequent Symmetry). A handwritten C quicksort program (using the same list-based algorithm) ran in 1.5 sec., only 50% faster than the translated code.

We chose this example to motivate the point that significant performance improvements over traditional systems (Monaco is the fastest multiprocessor implementation of FGHC that we know of) can be achieved with this technique, and the speeds are getting closer to optimized C. Although more sophisticated partitioning, based on granularity estimation [6] or profiling [12], is needed to retain multiple threads for parallelism, mode analysis is still required for safety, i.e., to keep cyclic dependencies out of a single thread.

This article presents a detailed description and empirical performance evaluation of four alternative algorithms for implementing this basic mode analysis concept. The three most significant results of this work are: 1) The analyzers presented are the first implementations of the underlying theory; 2) The first empirical performance measurements of such analyses are presented; 3) The analyzers proved to be competative with compilation in terms of execution time and memory use requirements, and 4) It was found that the superior analysis techniques are based on graph unification implemented by process networks, and model generation theorem proving (MGTP) implemented by meta-compilation. These analyzers are either fastest or use the least memory, and proved to be complete for the benchmarks studied. We have already started to incorporate the most promising of these prototypes within a full compiler to enable some of the optimizations previously discussed.

This article is organized as follows. Section 2 reviews the notions of paths, modes, and Ueda and Morita's original concept of mode analysis. The following four sections describe specific algorithms and implementations performing mode analysis: constraint propagation over a static graph (Section 3), constraint propagation over a dynamic (process network) graph (Section 4), partitioning a finite-domain (Section 5), and model generation theorem proving (Section 6). The algorithms are empirically evaluated and compared in Section 7 and conclusions are summarized in Section 8.

2 Background: Paths and Modes

Ueda and Morita's notion of "path" is adopted 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. f is referred to 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: *in* or *out*. The following definitions are from Ueda and Morita [21].

Definition: P_{Atom} is a set of paths which begin with predicate symbols. P_{Term} is a set of paths which begin with function symbols. □

Example: Considering the quicksort example, $< q/3, 1 >< ./2, 2 >< ./2, 1 > \in P_{Atom}$, $< ./2, 2 >< ./2, 1 > \in P_{Term}$ □

Definition: We define the set of modes $M = P_{Atom} \rightarrow \{in, out\}$. This means that a mode assigns either *in* or *out* to every possible path of every possible instance of every possible goal. □

Example: Considering quicksort, the *cadr* of the first argument of procedure $q/3$ has an input mode specified as: $m(\{< q/3, 1 >, < ./2, 2 >, < ./2, 1 >\}) = in$. □

Definition: For a mode $m \in M$ and a path $p \in P_{Atom}$, a *submode* of m, denoted m/p, is a function from P_{Term} to $\{in, out\}$, such that $\forall q \in P_{Term}((m/p)(q) = m(pq))$. □

Example: Considering the quicksort example, when $p = < q/3, 1 >< ./2, 2 >$, m/p represents a function from a set of the paths from the *cdr* of the first argument of procedure $q/3$ to $\{in, out\}$. □

Analyses presented in this paper exploit the rules outlined by Ueda and Morita. Their axioms are clarified and reformulated in Figure 2 ($m(p)$ means the mode of path p).

3 Constraint Propagation Algorithm

In the constraint propagation algorithm [19], a graph is constructed representing the entire program.[1] hierarchically, we compute and combine graphs at three levels: modules, procedures, and clauses. Top-down, we envision first constructing such a graph for each module of the program, and then connecting the graphs via imported/exported procedures. Within a module, we utilize this same strategy of first constructing procedure graphs ("local" analysis) and then combining graphs via inter-procedure call sites. Within a procedure, we first construct clause graphs, and then combine them via the heads.

Graph combination is formally unification, as described in section 3.3. The methodology is guaranteed to terminate because graph unification can only reduce the structural complexity. Termination occurs when no further reduction is possible. However, the scheme is not guaranteed to terminate in the *minimum* number of graph reductions. In the future we intend to experiment with heuristics to guide graph reduction to quicker termination (see Section 3.3).

3.1 Data Structures

A program graph is a directed, multi-rooted, (possibly) cyclic graph composed of two types of nodes. To clearly illustrate the following definitions, Figure 3 presents a portion of the quicksort program graph. This portion corresponds to the initial graph for procedure $q/3$ (clause 2).

[1]To simplify the description of all the algorithms we discuss the analysis of a single program, rather than a collection of program modules. Currently all the implementations globally analysis a single program at one time. However, proper software engineering of these prototypes can easily produce *incremental* tools that can analyze single modules in isolation (given mode information stored away from previous analyses of other modules).

§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 tell unification have opposite modes, for all possible p, or more formally: $\forall p\ m(\{<= /2, 1>\} \cdot p) \neq m(\{<= /2, 2>\} \cdot p)$

§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*. This means that one of the occurrences is designated as the producer of this variable.
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*.

Figure 2: Ueda and Morita's Mode Derivation Axioms (for Moded FGHC)

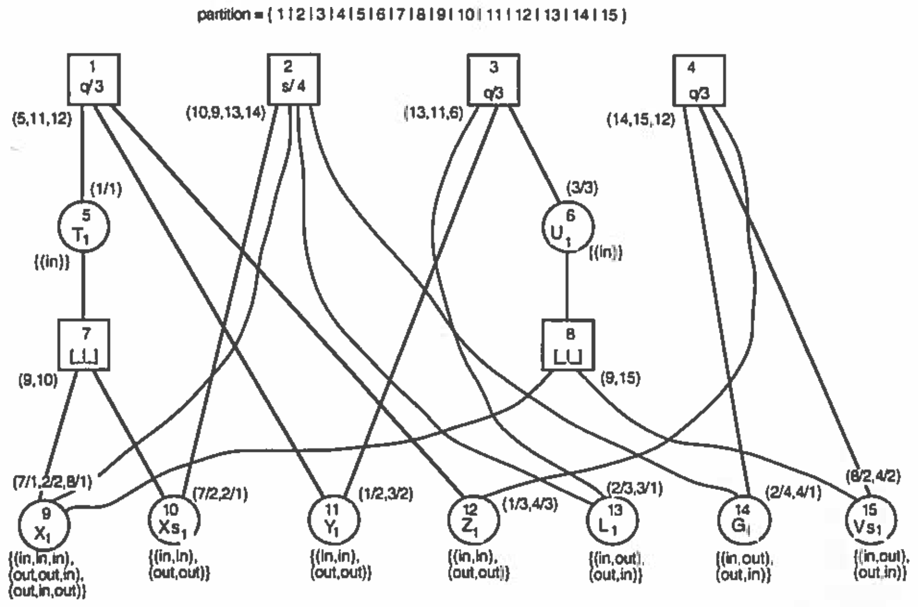

Figure 3: Initial Graph of Procedure $q/3$ (Clause 2), After Phases I-II

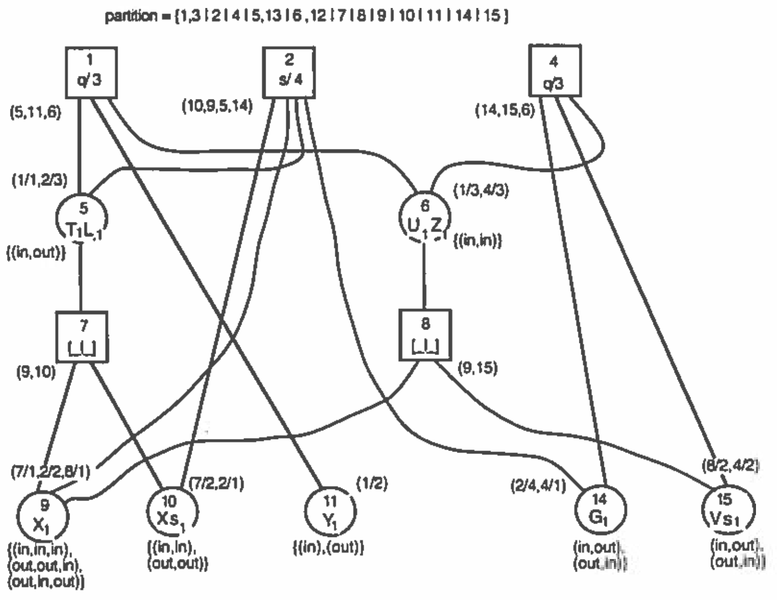

Figure 4: First Local Unification of $q/3$ (Clause 2)

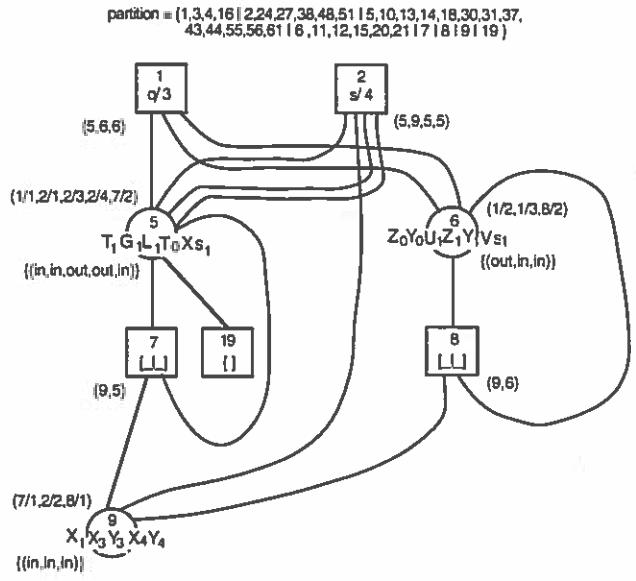

Figure 5: Minimized Graph of Quicksort

Definition: A *structure node* (drawn as a square) represents a functor with zero or more exit-ports corresponding to the functor's arity. If the node corresponds to a procedure name (for clause heads and body goals), there are no associated entry-ports (i.e., it is a root). If the node corresponds to a data structure, there is a single entry-port linked to a *variable node* unified with that term. A structure node contains the following information: a unique identifier, functor, and arity. □

Example: Consider node 8 in Figure 3 holding the list functor of arity two. The entry-port connects to variable node 6 (u_1) and the two exit-ports connect to variable nodes 9 and 15 (x_1 and vs_1 respectively). □

Definition: A *variable node* (drawn as a circle) represents a subset s of (unified) variables in a clause. Intuitively we think of these variables as aliases, and upon initial construction of the graph, s is a singleton (i.e., each unique variable in the clause has its own variable node initially). A node contains $k \geq 1$ entry-ports and $j \geq 0$ exit-ports, upon which directed edges are incident. A unique entry-port corresponds to each clause instance of each variable in s. An exit-port corresponds to a possible unification of the variable(s) to a term (exit-ports connect to *structure nodes*).[2]

[2]An invariant preserved by unification is that no two child (structure) nodes of the same parent (variable) node can have the same functor/arity (see discussion in Section 3.3).

A variable node contains the following information: a unique identifier and a *mode set* m. An element of m is a vector of length k containing self-consistent modes for the variable *instances* of s. To facilitate the implementation, each entry-port has a *name*: the identifier and exit-port number of its source node. Elements of m are *alternative* mode interpretations of the program. Initially m is computed by Ueda and Morita's rules.[3] intuitively, graph reduction results in removing elements from m as more constraints are applied by local and global unifications. A fully-reduced graph, for a fully-moded program, has a singleton m in each variable node. □

Example: Consider node 15 in Figure 3 with entry-ports named (8/2,4/2). node 15 holds variable vs_1 with mode set {(in,out),(out,in)}. this set derives from rule §3.1 in Figure 2. □

In general, initial graphs, like that in Figure 3, will be multi-rooted directed acyclic graphs. The initial roots correspond to clause head functors, body goal functors, and unification operators. In addition to the program graph, a *partitioned node set* is kept. Initially, each node is a singleton member of its own partition (disjoint set).

The mode analysis consists of three phases: i) creating a normalized form and initial graph; ii) removing unification operators from the graph, and iii) reducing the graph to a minimal form. These are described in the following sections.

3.2 Graph Creation

Phase I converts a flat committed-choice program into normalized form, an example of which is shown in Figure 1. Normalized form ensures that all variables are renamed apart among clauses within the same procedure and that each clause is flattened, i.e., all head structures and body goal structures are moved into ask and tell guards, respectively. An initial program graph is created from the normalized form, including mode sets for each variable node. The normalization implies a graph invariant that structure nodes cannot point to structure nodes (and variable nodes cannot point to variable nodes): the graph is a "layered network" (with cycles).

There is a minor trick required to deal with certain unification operators in phase I. Naively, a goal $x = f(y)$ would result in a structure node (=/2) pointing to another structure node ($f/1$). To guarantee the invariant of a layered network, a *dummy* variable node is place as an intermediary between these two structure nodes. This is effectively a nameless placeholder.

In phase II we remove all root nodes corresponding to builtin predicates. Intuitively, these predicates have fixed modes and thus their reduction acts as the boundary conditions anchoring subsequent constraint propagation by unification. After phase II, the resulting graph contains roots named only by clause heads and user-defined body goals. We list the reduction methods below.

[3]The size of m increases with the complexity of the rules, e.g., rule §4 (figure 2) can produce several vectors. By explicitly enumerating all possible modes initially, we simplify the analysis immeasurably.

- Passive unify operator in the ask guard (all head unifications have been normalized into the guard). The structure node corresponding to the operator has two exit-ports indicating the operands of the unification. The two variable nodes attached to these exit-ports can be *merged* as follows. A cross-product of two mode sets is taken, resulting in a set of vector pairs. We retain those pairs that have *in* mode for both entry-ports arriving from the unification node, and discard all other pairs. The two vectors in each remaining pair are concatenated, forming a new mode set. The two variable nodes are fused into one node containing the new mode set just computed. The entry-ports and modes corresponding to the unify operator are removed.

- Active (tell) unification goal. A similar merging operation is performed, keeping only those vector pairs that have *opposite* modes at the positions corresponding to the entry-ports arriving from the unification node (rule §2 in Figure 2).

- Other builtins. The modes of the variables in builtin goals are assigned by definition. For example, arithmetic assigns all RHS variables to *in* and the LHS variable to *out*. Vector builtins are assigned assuming that vectors remain fully ground throughout their lifetimes — this assumption is usually correct, but of course can lead to incorrect analysis. Unfortunately, given the aliasing problem, there is no way we know of to guarantee correct mode assignment to vector elements.

3.3 Abstract Unification

Phase III of the analysis is to reduce the graph to a *minimal form* by successive node unifications. We perform "local" reduction first by collapsing recursive call sites (by abstract unification) with associated clause heads. Next we perform "global" reduction by unifying root nodes from different procedures. The abstract unification algorithm is the same, however, for any two (node) arguments, local or global.

Figure 6 gives the general graph unification algorithm (a particular feature of the algorithm has been purposely removed to simplify the exposition: this is discussed in the next section). We use the notation that a variable node v has the fields: $v.in$ (vector of entry-ports, each of the form *id/index*, where *id* is the parent's node identifier and *index* is the parent's exit-port index), $v.out$ (set of exit-ports), and $v.modes$ (set of mode vectors). a structure node s has the fields: $s.out$ (vector of exit-ports), and $s.fun$ (functor/arity).

Unification is invoked as *unify(a,b)* of two nodes a and b (necessarily root structure nodes). The result is either failure, or success and a new graph (including the node partitioning) that represents the most general unification (mgu) of the two operands. implied data structures used by the algorithm include the graph, the disjoint sets (i.e., node partitioning), and a *mark table* associated with pairs of nodes.[4]

[4]Note that the new graph returned by *unify* has the same number of nodes as the original graph. However, for practical purposes *either* of the input roots can be discarded, turning unneeded nodes into garbage. The key point is that the graph must always be traversed according to the node partitions, so as the partitions grow larger, effectively the graph shrinks. To ensure that the graph shrinks in practice, unneeded nodes must be explicitly removed from the graph, an operation that we do not detail here.

```
unify(a, b) {
    ∀$a, b$ clear mark({$a$,$b$})                                        (1)
    sunify($a$, $b$)
}

sunify($a$, $b$) {
    if $a.fun \neq b.fun$
        return(failure)
    if mark({$a$,$b$}) clear then {
        set mark({$a$,$b$})                                             (2)
        $\forall k \in [1, arity(a)]$ {
            if vunify($a/k$, $a$.out[$k$], $b/k$, $b$.out[$k$]) failure then
                return(failure)
        }
    return(success)
  }  }

vunify($i$, $a_v$, $j$, $b_v$) {
    $a$ = find_set($a_v$)
    $b$ = find_set($b_v$)
    if mark({$a$,$b$}) clear then {
        set mark({$a$,$b$})
        $c$ = union($a$,$b$)                                            (3)
        — compute the compatible mode set
        define $u$ s.t. $a.in[u] = i$
                 $v$ s.t. $b.in[v] = j$
        $p = a.modes \times b.modes$
        $p' = \{(s,t) \in p \mid s[u] = t[v]\}$                         (4)
        if ($p'$ empty) then
            return(failure)
        $c.modes = \{ s \,||\, (m_1, m_2, ..., m_{v-1}, m_{v+1}, ..., m_k) \mid$
                  $(s,t) \in p',\; t = (m_1, m_2, ..., m_v, ...m_k) \}$       (5)
        — compute the entry-port identifiers
        let $(p_1, p_2, ..., p_v, ..., p_k) = b.in$
        $c.in = a.in \,||\, (p_1, p_2, ..., p_{v-1}, p_{v+1}, ..., m_k)$      (6)
        — compute the exit-ports identifiers
        $u = a.out \cup b.out$
        $f_a = \{s.fun \mid s \in a.out\}$
        $f_b = \{s.fun \mid s \in b.out\}$
        $i = \{s \mid fun \in f_a \cap f_b,\; s.fun \in u\}$
        $c.out = a.out \cup i$                                          (7)
        — unify children with the same functor/arity
        $\forall(x,y) \mid x \in a.out, y \in b.out$ {
            if $x.fun = y.fun$
                if sunify($x$, $y$) failure then                        (8)
                    return(failure)
        }
    return(success)
  }  }
```

Figure 6: One-Pass Rational-Tree Unification Algorithm

Procedures *sunify* (structure node unification) and *vunify* (variable node unification) follow recursive descents. Initially all marks are cleared (1). Circular structures that represent infinite paths are handled properly by *marking* node pairs at first visit (2). If a given node pair has been previously marked, revisiting them immediately succeeds. Note that we mark *pairs* instead of individual nodes to handle the case of unifying cyclic terms of unequal periodicity.

Two important operations for the disjoint sets data structure are *union(x,y)* and *find_set(x)*. Function *union(x,y)* unites two disjoint sets, where x belongs to the first disjoint set and y belongs to the second disjoint set. Procedure *union* returns the *canonical name* of the partition (3), i.e., the least identifier of the nodes. This facilitates reusing graph nodes while rebuilding the graph.[5] function *find_set(x)* returns the canonical name of the disjoint set containing x.

The major complexity in the algorithm is in procedure *vunify*, where the abstract unification must merge the modes of the two argument nodes. First, mode vectors that are contradictory are discarded (4). If all mode vectors are contradictory then a mode error has occurred and unification fails. Otherwise redundant modes are removed and the two mode vectors are concatenated (5). Next we create the entry-port identifiers associated with the new mode vector (6). Lastly, children of the argument nodes that share equal functor/arity must be recursively unified (8) The exit-port identifiers consist of a single exit-port for each pair of children unified, included with exit-ports for all children for which unification does not take place (7). Intuitively, a variable node forms or-branches with its children, whereas a structure node forms and-branches with its children. In other words, the least-upper-bound (lub) of the abstract unification semantics at a variable node is a union of the structures that potentially concretely unify with the variable node.

Local analysis continues with unification of roots among clauses composing a single procedure definition. Local analysis terminates when no two roots have the same functor and arity within a procedure graph. This is perhaps not a time-optimal strategy, but was selected, in our prototype, for its simplicity. Analyzing non-recursive clauses first, and then unifying these clause-head roots with recursive call sites in other clauses, is expected to terminate faster. The rationale is similar to quickly reaching a fix point in abstract interpretation by approximating recursive calls with their corresponding non-recursive clause input/output relationships.

After local analysis we perform global analysis which unifies roots among different procedure graphs. Global analysis terminates when each root in the entire program is unique.

> *Example*: Reconsider the quicksort program (Figure 1) to illustrate the unification algorithm. First, we consider the second clause of $q/3$ and construct the graph in Figure 3. Each node of the graph is assigned a unique identifier. Then entry-ports, exit-ports, and modes are shown.
>
> We start the local analysis for this graph by unifying the node 1 with node 3. The result of this unification is shown in Figure 4.[6] that is we unify all three

[5]Optionally, the canonical name can be defined to be a new identifier, avoiding node reuse. This might facilitate searching for the modes of top-level variables, e.g., for data-dependency analysis.

[6]By convention we remove nodes 3, 12, and 13 from the illustration to clarify that they represent redun-

corresponding argument positions of both nodes. The first pair of arguments unified is node 5 (entry-port 1) with node 13 (entry-port 2). Initially these nodes belong to their own partitions, so *find_set* returns 5 and 13. The *union* returns the canonical name 5, the minimum of the two. Thus node 5 is overwritten with the new (fused) variable node.

For example, all possible modes of node 5 entry-ports are $\{(in)\}$ and the mode set of node 13 is $\{(in,out),(out,in)\}$. the cross-product set is $\{(in)\} \times \{(in,out),(out,in)\} = \{(in,in,out),(in,out,in)\}$. Element $\{(in,in,out)\}$ is discarded because the first and the third positions (instances, or entry-ports, of the unified argument) do not match. Thus, the legal cross-product set is $\{(in,out,in)\}$. This is reduced to the mode set $\{(in,out)\}$ and entry-port vector (1/1,2/3), removing redundancies. Note that by fusing the entry-port vectors, the edge from node 2 to node 13 has effectively been rerouted to fused node 5.

Exit-ports need to be traversed in a recursive descent. In this case, the exit-port of node 5 is simply inherited by the new node since node 13 has no children. Then the second and third arguments of nodes 1 and 3 are unified, resulting in Figure 4. Node 1 and node 4 are then unified (not shown). This is the final local unification possible for clause 2 since all roots now have unique functors. We then do inter-clause analysis of $q/3$, unifying the roots of the two clauses of $q/3$ (not shown). This represents the most information that can be derived from $q/3$ alone. The $s/4$ modes are similarly derived locally, and then global analysis is performed, unifying the $s/4$ graph with node 2 of the $q/3$ graph. We show only the final resultant graph with all mode ambiguities removed in Figure 5.

Note that proper interpretation of the path modes requires the use of Ueda and Morita's "polarity rule" [21]. Thus if the path from a root passes through an odd number of *out* modes (each attached to some variable node in the path), then the final mode is the *opposite* of that listed. For example, $\{<q/3,2>, <./2,2>\}$ passes through mode *out* at node 6 (Figure 5) and terminates at mode *in* at node 6. Thus its correct mode is *out*. □

3.4 On Completeness

Unification as previously specified is not complete in degenerate situations of the type illustrated in Figure 7. In this case, variable X within procedure $f/2$ must not produce output for the program to be moded. If it *did* produce output, it would be sent out through the head and into the self-recursive call: a contradiction. Note that procedure $h/1$ induces X to produce output a. However, our previous unification algorithm will *not* detect the mode conflict. The heart of the problem lies within $f/2$ which does not explicitly mention structure $s/1$. If $s/1$ were introduced into this clause:

$$f(X,Y) \ :- \ X = s(\_),\ Y = b(X) \ : \ true \ | \ f(X,\_).$$

dant information. In the implementation, these nodes might be kept in the graph if space was not a critical resource.

$$g \;\text{:-}\; \mathit{true} : \mathit{true} \mid f(\_, Z),\; h(Z). \qquad (1)$$
$$f(X,Y) \;\text{:-}\; Y = b(X) : \mathit{true} \mid f(X, \_). \qquad (2)$$
$$h(b(s(W))) \;\text{:-}\; \mathit{true} : W = a \mid \mathit{true}. \qquad (3)$$

Figure 7: A Program with a Mode Contradition

the contradiction would become apparent to our analyzer. In general, however, it is impossible to determine which structures are relevant to a given clause.

The key point is: whenever a shared variable is passed, perhaps indirectly, to both a procedure head and a corresponding self call, we must not lose sight of rule §4 in Figure 2. This is a degenerate form of the rule because the occurrences within the head and self call must have identical modes.

To ensure that this constraint is enforced, in the complete algorithm variable nodes potentially go through three states: *normal*, *multiway*, and *shared*. Initially, a variable node is marked multiway if it is moded by rule §4, otherwise it is normal. If during graph reduction a multiway node is unified with itself, its status changes to *shared*. Unification of normal and multiway variable nodes proceed as previously described. However, unification of a shared variable node with another variable node (of any type) is treated specially. This is precisely the case that captures X in the example above.

First, the unification of the variable nodes themselves is performed as usual. However, the recursive descent is modified. If all mode vectors within the fused variable are *in* on the unification path, then this particular variable occurrence (represented by the entry port) is a consumer in all possible scenarios. In fact, *all variables within terms bound to that variable must be consumers also* because any producers would cause contractions as illustrated in the previous example.

During the unification's descent from the fused node, recursive unifications of child pairs proceed normally ((8) in Figure 6). However, singleton children for which unification does *not* take place are *no longer* inherited (cf. (7) in Figure 6). These children are not safe: they may erroneously be producers. Thus we descend each such child, removing all mode vectors with *out* along our path. This removes all contradictory senarios, potentially resulting in an empty mode for some node, i.e., unification failure.

4 Process Network Analyzer

The previous constraint propagation algorithm was alternatively implemented by a process network wherein each node of the graph was an active, concurrent process. Nodes communicated by message passing over streams to accomplish reduction. The motivations for moving the graph from a static data structure to an active process network are: 1) concurrency is increased because updating the graph no longer bottlenecks the computation; 2) unification of graph nodes corresponds to merging node processes, thus resource requirements made by the analyzer decrease as execution proceeds, and 3) an active process network is an elegant paradigm for this problem. This is an implementation alternative for the previous algorithm: it inherits completeness from that algorithm.

Translating the previous algorithm (Figure 6) requires the specification of how recursive unification can proceed via message passing, how the distributed unification can terminate (both successfully and by failure), and how the final mode information can be read from the reduced graph. These issues are described in detail in the following sections.

4.1 Distributed Unification

A `node` process is defined manage a graph node (either a variable or structure node). The `node` process contains state holding a unique integer identifier, a symbol (functor/arity for structure nodes and the atom `'$VAR'` for variable nodes), mode information, and a flag indicating if the node is from a clause head. Mode information consists of a set of mode vectors and a vector of entry ports, as described in Section 3.1. In addition, a `node` has an input stream, a list of output streams to children, and a global termination flag.

A `node` process acts on the following messages:

- `unify(+Id,+S0,-S1,+Parents,+Ans,+Done)`: receipt of this message indicates that this node is requested to initiate a unification with node `Id` on input stream `S0`. `Parents` are the two parent nodes who made this unification request. The results of the unification are `S1` which is the tail of the stream to node `Id` and `Ans`, a short-circuit chain for unification termination. `Done` is the short-circuit chain for message termination.

- `who(-Info,-In,-Out,+Done)`: receipt of this message indicates that this node is to be unified with another node, and therefore this node is to be terminated. Before termination, the state of the node is passed back to the initiator node via back-messages: `Info`, `In`, and `Out`. `Done` is the short-circuit chain for message termination.

- `echo(+Path,+Done,+Parent,+Polarity,-Out)`: `Path` is a list of steps representing path from the root. the output `Out` is produced by joining this path with the mode of this node. The mode at this node is dependent on inputs `parent` and `Polarity`. The parent is needed to select the proper element of the mode vector(s): if all vectors agree in mode, the node's mode is *in* or *out*. if the polarity is odd, the mode toggles. Otherwise, if any two vectors disagree, the mode is *dk* (don't know). `Done` is the short-circuit chain for message termination.

- `kill`: terminate this node process after sending `kill` messages to all children. This message only appears during destruction of the entire graph after algorithm completion.

- `[]`: if the input stream is closed, this node closes all of its child streams. Closing streams simply shuts down mergers associated with this node but does *not* terminate the children, cf. `kill`.

The implementation shared phases I and II with the previous algorithm, and then spawns a node process network from the static graph definition. The root list is grouped into pairs which are unified, then these resulting trees are unified and so on, forming a logarithmic tree of unifications. There is no attempt to sort the unifications to reduce the necessary work.

A `node` that receives a `unify/6` message is the "active" member of a reduction. It sends a `who/4` message to the "passive" member, who returns all its state information on a back message and terminates itself. For structure–structure unification, the node symbols are compared and if matched, the active node sends `unify/6` messages to one member of each pair of children. otherwise failure occurs.

For variable–variable unification, first the mode sets must be merged as described in Figure 6. If the merge is successful, then only children with matching functors are unified, by sending `unify/6` messages. Non-matching children are simply appended to the output stream list of the active node. If the mode set merge is a failure, then the unification fails.

4.2 Terminating Unification

There are two levels of termination occurring within the analysis: termination of an individual (tree) unification and termination of the entire (graph) unification. The former is accomplished by stringing a short-circuit chain through all nodes involved in a root-to-root unification. The far left-hand link is bound to 'yes'. Each successful node reduction shorts the chain. An failing reduction binds the right-hand link to 'no' which propagates to the far right-hand link.

Termination of the entire graph reduction is accomplished in two ways. successful termination must be indicated only after all messages have been processed. We string a short-circuit chain through every message to handle this. When a message is read, its link is shorted, and when all messages are read, the far left-hand link and far right-hand link are shorted. Additionally, if an individual tree unification fails, a global termination variable is set appropriately. All `node` processes share this termination variable and will discard all incoming `unify/6` messages once it is set.

4.3 Accessing the Graph

As described above, the graph can be accessed by `echo/5` messages. A node must be careful not to propagate an echo message if it already appears in the path contained in that message. This rather arbitrary method cuts cycles in the graph immediately. We could elect to dump out cyclic paths of any depth, but for the purposes of comparing the algorithms in this paper, all the analyzers were normalized to produce the same non-cyclic output.

4.4 Comparison with Static Analysis

Both of the previous analyses are complete, and being based on the same underlying algorithm, deserve qualitative comparison here before we introduce later empirical measurements in Section 7. The primary difference between the two implementations is that the active graph is fully concurrent. The static graph is sequentialized by necessity to update the graph consistently. One fix would be to partition the graph into independent subgraphs (finding the strongly-connected components of the call-graph), allowing concurrent reduction.

In general, the process network analyzer was more difficult to build than the static graph analyzer because the active graph confuses debugging. However, compared to other distributed algorithms, debugging was not overly burdensome because our abstract unifications monotonically approach the final state.

From profiling information we determined that the active graph analyzer spends most of its time checking for self-unification of a node (necessary for circular unification) and (to a lesser degree) manipulating mode vectors. To check for self-unification, we instituted a naming scheme wherein the identifiers of two nodes to be unified are concatenated to form the identifier of the new node. Thus node identifiers grow in size during reductions, and although we use difference lists to concatenate cheaply, the cost of checking membership within an identifier list grows. An alternative would be to allow both nodes to live (currently we terminate one of them to save space), and update the state in each to indicate the current minimum identifier of the alias set. We have not yet experimented with this option (it is very similar to method used in static graph implementation).

Mode vector manipulation requires finding the indices (within the vectors) of the mode elements being compared, and concatenation of the two vectors (less the duplicate mode element which is removed). Time is spent about equally between these main functions. Quickly finding indices requires a more sophisticated data structure than the current list. Quick concatenation requires either difference lists or bit vectors. Both are complicated by the removal of duplicate elements. In fact, the static graph implementation elected to forgo duplicate removal and used difference lists for mode vectors. This contributes to the increased space requirement for the static graph analysis (Section 7). The active graph implementation uses standard lists with removal. We need further experimentation to determine the best solution.

The space complexities of the active graph analyzer lie in spawning a process for each graph node. This working set churns through memory more quickly than the static graph implementation (which can exploit local memory reuse in PDSS to keep data copying low). Currently we do not constrain the number of processes, but this could be accomplished in the manner opposite to parallelizing the static graph analyzer: first finding groups of strongly-connected components of the program's call graph, and then analyzing only one group at a time. For example, a short-circuit chain could be used to force synchronization between one group and the next. In a multiprocessor system, explicit load distribution of the groups would be needed.

5 Finite Domain Analysis

In an effort to avoid circular unification altogether and much of the overheads of maintaining the graph, either statically or actively, a radically different algorithm was developed [7]. The first stage of this alternative algorithm generates a finite set of paths whose modes are to be considered. Only "interesting" paths are generated in the first stage of the algorithm: effectively those paths locally derived from the syntactic structure of the procedures. 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.

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

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

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 "$a(X, Z)$:- *true* : $Y = c(X),\ Z = b(Y)$ | *true*." where the interesting path $\{< a/2,2>,\ <b/1,1>,\ <c/1,1>\}$ given by the first tell unification goal will not be generated until the interesting path $\{<a/2,2>,\ <b/1,1>\}$ in the second tell unification goal is generated. Such repeated scans should occur infrequently 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 tell unification goal (i.e., a path of the form $\{<=/2,2>\}\cdot s$) is interesting, then so is the corresponding path starting on the left-hand side of that unification (i.e., $\{<=/2,1>\}\cdot s$).

In general, all interesting paths of a program are generated in a few sequential passes. The 39 interesting paths of quicksort, shown in Table 1, are generated in two passes. Note the correlation between these paths and the infinite paths represented in Figure 5. A depth-one traversal of the graph (which we call the "broken" paths in Section 7) produces all the interesting paths, in addition to two other paths: $\{<q/3,2>,\ <./2,1>\}$ and $\{<q/3,2>, <./2,2>\}$. These two paths are "hidden" because they cannot be derived from clause 2 of $q/3$ alone. However, the set of interesting paths produced is sufficient to mode the program in the sense of assigning an unambiguous mode to all syntactic variables. An important question is whether a finite set of paths represents a *minimal and complete* set of paths for the mode analysis in general. Unfortunately, this is not the case, as is discussed below.

Once we have generated a set of interesting paths, our algorithm proceeds by simply noting the modes of paths, first directly, and then by examining relationships between paths. There are essentially four different stages in the algorithm:

1. Assert absolute modes for some paths.
2. Assert that all paths on opposite sides of a tell unification have opposite modes.
3. Proceed sequentially through the variables derivable from interesting paths, asserting all binary relations between paths.
4. Repeatedly consider multiway relations (rule §4 Figure 2) asserted by the clauses.

The first three stages have linear complexity. The multiway analysis is exponential in the number of variables, but by the time it is actually performed, most alternatives contradict the known modes, and thus are not explored. We found multiway analysis contributed only 11% on average to the total analysis execution time, with one benchmark showing an extreme of 33% (see Section 7.1).

5.1 On Completeness

Some important practical and theoretical issues are raised by this algorithm. 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.

The major barrier to the consistency of this algorithm is somewhat 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. 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. (Recall that we say a program is "fully moded" if the modes of *all* paths are known, and "moded" if the modes of *some* paths are known).

Thus, the infinite set of all possible paths is a complete set; however, we are interested in *finite* complete sets and in particular in a *minimal* complete set of paths for the program. Our path generation algorithm is incomplete; because of this incompleteness in path generation, the mode analysis algorithm we constructed is *unsafe*. It is 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. Thus, the analysis is unsafe in the sense the compiler may not detect mode contradictions in erroneous (not fully-moded) programs, and thereby produce erroneous mode information for programs that should be rejected altogether. Nonetheless, most generated paths in typical programs are moded by the analysis, and if the program being analyzed is known to be moded, all modes derived are correct. Thus, the analysis algorithm can be a practical tool for many compiler optimizations.

In the next section we introduce another analysis technique that also loses completeness, but in a rather different way. Here we map the cyclic constraint graph onto axioms for theorem proving, which proves to be incomplete when limiting the nesting depth of axiom invocation. The analyzer is quite interesting and is shown to be "more" complete than the finite domain method.

6 Mode Inference Method

This section describes mode analysis for concurrent logic programs with a Model Generation Theorem Prover (MGTP) [5]. Mode analysis is a kind of fixed-point computation what corresponds to generating a model in MGTP. The generated model includes information which let us know the variable's mode and the mode consistency of the program. In this section we first describe how model generation theorem proving works and how the problem of mode analysis is mapped onto the theorem prover. We then discuss the issue of completeness how it relates to the finite domain analysis.

6.1 Model Generation

An MGTP clause is represented by an implicational form:

$$A_1, A_2, \ldots, A_n \rightarrow C_1 \ ; \ C_2 \ ; \ \ldots \ ; \ C_m$$

where A_i $(1 \leq i \leq n)$ and C_j $(1 \leq j \leq m)$ are atoms (atomic formula) ; the antecedent is a conjunction of $\{A_1, A_2, \ldots, A_n\}$; the consequent is a disjunction of $\{C_1, C_2, \ldots, C_m\}$. A clause is said to be *positive* if its antecedent is *true* (n=0), *negative* if its consequent is *false* (m=0), and otherwise *mixed* ($m \neq 0$, $n \neq 0$). The following two rules act on the model generation method:

- Model Extension Rule: If there is a clause, $A \rightarrow C$, and a substitution σ such that $A\sigma$ is satisfied in a model candidate M but $C\sigma$ is not satisfied in M, then extend the model candidate M by adding $C\sigma$ to M.
- Model Rejection Rule: If there is a negative clause whose antecedent $A\sigma$ is satisfied in a model candidate M, then reject M.

Model generation attempts to construct a model for a given set of clauses, starting with a null set as a model candidate. A model candidate M is a model of the given set of clauses if the above two rules cannot be applied to M. If the clause set is satisfiable, a model should be found. This method can also be used to prove that the clause set is unsatisfiable, by exploring every possible model candidate to ensure that no model exists for the clause set.

Example: To illustrate, Figure 8 shows a proof tree for toy problem. We start with an empty model candidate, $M_0 = \phi$. M_0 is first expanded into two

$C_1: \quad p(X), s(X) \rightarrow false.$
$C_2: \quad q(X), s(Y) \rightarrow false.$
$C_3: \quad q(X) \rightarrow s(f(X)).$
$C_4: \quad r(X) \rightarrow s(X).$
$C_5: \quad p(X) \rightarrow q(X); r(X).$
$C_6: \quad true \rightarrow p(a); q(b).$

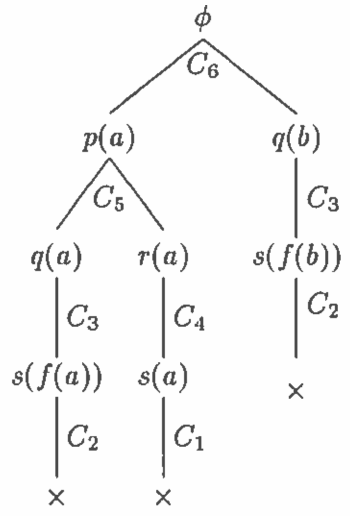

Figure 8: Sample MGTP Program and its Proof Tree

cases, $M_1 = \{p(a)\}$ and $M_2 = \{q(b)\}$, by applying the model extension rule to C_6. Then M_1 is expanded by C_5 into two cases: $M_3 = \{p(a), q(a)\}$ and $M_4 = \{p(a), r(a)\}$. M_3 is further extended by C_3 to $M_5 = \{p(a), q(a), s(f(a))\}$. Now with M_5, the model rejection rule is applicable to C_2; thus M_5 is rejected and marked as closed. On the other hand, M_4 is extended by C_4 to $M_6 = \{p(a), r(a), s(a)\}$, which is rejected by C_1. Similarly, the remaining model candidate M_2 is extended by C_3 to $M_7 = \{q(b), s(f(b))\}$, which is rejected by C_2. Now that there is no way to construct any model candidate, we can conclude that the clause set is unsatisfiable. $\square$

6.2 Mode Analysis in MGTP

In the mode inference method, formulae representing mode constraints are translated into a set of MGTP clauses. Mode analysis of the entire source program is reduced to computing a model of this set of clauses. We outline transformation from mode constraints to a set of clauses for MGTP using the quicksort program (Figure 1). The left column of Table 2 lists the mode constraints which are derived from quicksort. The right column lists the corresponding clauses.

The mode constraints have two forms: $m() = in$ (or $m() = out$) and $m/p = m/q$ (or $m/p = \overline{m/q}$).[7] Roughly speaking, $m() = in$ is translated to a positive clause $true \rightarrow m() = in$, and $m/p = m/q$ is translated to two mixed clauses $m/p \rightarrow m/q$ and $m/q \rightarrow m/p$. Within MGTP clauses, *m(Path,Mode)* is a relation showing that the mode of path *Path* is *Mode*. A path is represented in list notation, e.g., $\{<q,1><.,1>\}$ is translated to $[q/1, ./1]$. Here we refrain from including *arities* in order to simplify the exposition (recall that in Section 2, this path would be noted: $\{<q/3,1><./2,1>\}$). Some MGTP clauses are eliminated

[7]Recall that the definition of m/p is given in Section 2.

Mode Constraints	MGTP Clauses	
$m(<q,1>) = in$	$true \rightarrow m([q/1], in)$	(1)
$m/<q,2> = \overline{m/<q,3>}$	$m([q/2 \mid X], M) \leftrightarrow m([q/3 \mid X], \overline{M})$ †‡	(2)
$m/<q,1><.,1> = m/<p,2>$	$m([q/1, ./1 \mid X], M) \leftrightarrow m([p/2 \mid X], M)$	(3)
$m/<q,1><.,1> = m/<q,3><.,1>$	$m([q/1, ./1 \mid X], M) \leftrightarrow m([q/3, ./1 \mid X], M)$	(4)
$m/<p,2> = \overline{m/<q,3><.,1>}$	$m([p/2 \mid X], M) \leftrightarrow m([q/3, ./1 \mid X], \overline{M})$	(5)
$m/<q,1><.,2> = m/<p,1>$	$m([q/1, ./2 \mid X], M) \leftrightarrow m([p/1 \mid X], M)$	(6)
$m/<q,2> = m/<q,2>$	*eliminated*	
$m/<q,3> = m/<q,3>$	*eliminated*	
$m/<p,3> = \overline{m/<q,1>}$	$m([p/3 \mid X], M) \leftrightarrow m([q/1 \mid X], \overline{M})$	(7)
$m/<p,4> = \overline{m/<q,1>}$	$m([p/4 \mid X], M) \leftrightarrow m([q/1 \mid X], \overline{M})$	(8)
$m(<q,3>) = in$	$true \rightarrow m([q/3], in)$	(9)
$m/<q,3><.,2> = \overline{m/<q,2>}$	$m([q/3, ./2 \mid X], M) \leftrightarrow m([q/2 \mid X], \overline{M})$	(10)
$m(<p,1>) = in$	$true \rightarrow m([p/1], in)$	(11)
$m(<p,3>) = out$	$true \rightarrow m([p/3], out)$	(12)
$m(<p,4>) = out$	$true \rightarrow m([p/4], out)$	(13)
$m/<p,1><.,1> = \overline{m/<p,3><.,1>}$	$m([p/1, ./1 \mid X], M) \leftrightarrow m([p/3, ./1 \mid X], \overline{M})$	(14)
$m(<p,1><.,1>) = in$	$true \rightarrow m([p/1, ./1], in)$	(15)
$m(<p,2>) = in$	$true \rightarrow m([p/2], in)$	(16)
$m/<p,1><.,2> = m/<p,1>$	$m([p/1, ./2 \mid X], M) \leftrightarrow m([p/1 \mid X], M)$	(17)
$m/<p,2> = m/<p,2>$	*eliminated*	(18)
$m/<p,3><.,2> = m/<p,3>$	$m([p/3, ./2 \mid X], M) \leftrightarrow m([p/3 \mid X], M)$	(19)
$m/<p,4> = m/<p,4>$	*eliminated*	
$m/<p,3> = m/<p,3>$	*eliminated*	
$m/<p,4><.,2> = m/<p,4>$	$m([p/4, ./2 \mid X], M) \leftrightarrow m([p/4 \mid X], M)$	(20)

† $m(P, M) \leftrightarrow m(P', M)$ represents two MGTP clauses:
$m(P, M) \rightarrow m(P', M)$ and $m(P', M) \rightarrow m(P, M)$.

‡ $\overline{M}$ is the inverse mode of M i.e., $\overline{in} = out$ and $\overline{out} = in$.

Table 2: Mode Constraint → MGTP Clause Transformation (Quicksort)

because their mode constraints represent the reflexive law.[8] Mode consistency is ensured by the inclusion of a negative clause: $m(P, M), m(P, \overline{M}) \rightarrow false$ which means that the mode is inconsistent when a path P has two modes: *in* and *out*.

> Example: For quicksort, we have 7 positive (one literal) clauses and 24 mixed clauses. We start with an empty model candidate $M_0 = \phi$. M_0 is first expanded to $M_1 = \{m([q/1], in), m([q/3], in), m([p/1], in), m([p/3], in), m([p/4], in), m([p/1, ./1], in), m([p/2], in)\}$, by applying the model extension rule to the 7 positive clauses.
>
> One direction of rule (2) $m([q/3|X], M) \rightarrow m([q/2|X], \overline{M})$ is applicable to M_1 because $m([q/3], in)$ ($\in M_1$) can be unified with the antecedent $m([q/3|X], M)$ by substitution $\{[\,]/X, in/M\}$. So we get the consequent $m_1 = m([q/2], out)$. This means that M_1 is extended by (2) to $M_2 = M_1 \cup \{m_1\}$. Repeating a similar production of mode information, we get the final result: $M_1 \cup \{m([q/2], out), m([q/1, ./1], in), m([q/3, ./1], in), m([q/2, ./1], out), m([p/3, ./1], out), m([p/4, ./1], out)\}$.
>
> Note that straight application of model generation to mode analysis can cause an infinite generation. For example, the positive clause (11) $true \rightarrow m([p/1], in)$ and the one direction of (17) $m([p/1|X], M) \rightarrow m([p/1, ./2|X], M)$ causes an infinite generation. We have an infinite sequence $m([p/1], in), m([p/1, ./2], in), m([p/1, ./2, ./2], in), \ldots$.
>
> To avoid infinite generation, MGTP needs extra information with which it can detect the infinite loop. In its current implementation, MGTP attaches trace information to each model element. The trace information is a sequence of applied rules which let us know how to derive the model element. In the previous example, $m([p/1], in)$ is directly from a positive clause, $m([p/1, ./2], in)$ is derived by applying (17) to $m([p/1], in)$, and so on. Here, extended MGTP derives the sequences $< m([p/1], in), [\,] >$, $< m([p/1, ./2], in), [17] >$, $< m([p/1, ./2, ./2], in), [17, 17] >$ where the second element of the tuple is the trace.
>
> When MGTP derives the last element $< m([p/1, ./2, ./2], in), [17, 17] >$, MGTP regards the sequence as infinite and discard not only the last element $< m([p/1, ./2, ./2], in), [17, 17] >$ but also the previous one $< m([p/1, ./2], in), [17] >$. It does so because $m([p/1, ./2], in)$ can be regarded as the same information as $m([p/1], in)$ because both elements indicate the same node of the mode graph [21]. □

6.3 On Completeness

The mechanism to avoid infinite generation introduces incomplete analysis. Consider the program listed in Figure 9. From this program, we translate the MGTP clauses listed[9] in Figure 10. The program invocation " ?- $f(X), h(X)$." causes a unification failure between a

[8]They can be also eliminated by the tautology-elimination rule which is a theorem-proving technique.

[9]This list includes only clauses related to the discussion.

$$
\begin{array}{lr}
f(\ [\,]\ )\ \text{:-}\ \mathit{true} : \mathit{true} \mid \mathit{true}. & (1)\\
f(\ [\,A \mid B\,]\ )\ \text{:-}\ \mathit{true} :\ A\ =\ a \mid g(B). & (2)\\
g(\ [\,]\ )\ \text{:-}\ \mathit{true} : \mathit{true} \mid \mathit{true}. & (3)\\
g(\ [\,b \mid A\,]\ )\ \text{:-}\ \mathit{true} : \mathit{true} \mid f(A). & (4)\\
h(X)\ \text{:-}\ \mathit{true} :\ X\ = [A,b,A,b,b] \mid \mathit{true}. & (5)
\end{array}
$$

Figure 9: A Program with Failure (Mode Contradition)

$$
\begin{array}{ll}
\text{(2-1)} & \mathit{true} \rightarrow m([f/1,./1],\mathit{out})\\
\text{(2-2)} & m([f/1,./2|X],M) \rightarrow m([g/1|X],M)\\
\text{(4-1)} & m([g/1,./2|X],M) \rightarrow m([f/1|X],M)\\
\text{(5-1)} & \mathit{true} \rightarrow m([h/1,./2,./2,./2,./2,./1],\mathit{out})\\
\text{(Q-1)} & m([h/1|X],M) \rightarrow m([f/1|X],\overline{M})
\end{array}
$$

Figure 10: A Subset of MGTP Clauses for Sample Program

and b, indicating a mode contradiction in path $\{<h,1><.,2><.,2><.,2><.,2><.,1>\}$. MGTP, however, doesn't detect the possibility of error.

From (5-1) and (Q-1), an element $m([f/1,./2,./2,./2,./2,./1],in)$ is derived. And the successive applications of (2-2) and (4-1) to the element yield $m([f/1,./2,./2,./1],in)$. One more application can yield $m([f/1,./1],in)$, which disagrees with (2-1). The last application, however, is prohibited to protect against infinite generation.

This shows that MGTP's ability to avoid infinite generation is incomplete. For this kind of example, protection can be relaxed. For example, the right side of $m([g/1,./2|X],M) \rightarrow m([f/1|X],M)$ is simpler than the left side, having fewer symbols. The application of this kind of clause to an element generates a simpler element, so the successive applications of these clauses must terminate at some time. For these clauses, infinite generation is not a problem. We have not yet ensured that generating a model with relaxation yields enough mode information. Completeness will be proved in a future work.

7 Performance Comparison

In this section we examine the characteristics of the four algorithms by evaluating the analysis of a benchmark suite on each implementation. A preliminary investigation of the analyzers built at the University of Oregon is given in Tick [16]. The benchmark suite is summarized in Table 3. The programs include simpler, common benchmarks [15], as well as more complex ones: **rucs** is a rule check system (written in KL1 by R. Ezaki from the original Prolog by Y. Koseko); **bestpath** implements Dijkstra's (sequential) single-source best path algorithm with a heap data structure; **waves** (originally written in Strand by I. Foster) builds a multidimensional torus using an iterative technique; **waltz** is Waltz's constraint satisfaction algorithm, and **mandel** (also from Foster's Strand original) computes

program	proc	clause	symbols			broken paths			
			const	vars	total	totals			avg length†
msort	4	11	54	75	129	36†	30	26	1.7
queens	6	14	77	119	196	71	43	43	1.8
cubes	9	16	93	159	252	224	79	361	2.7
pascal	11	22	143	200	343	338	56	147	2.0
mandel	18	26	170	334	504	408	140	336	2.4
rucs	16	66	218	390	608	79	46	131	1.6
bestpath	20	44	279	492	771	507	207	492	2.5
waltz	20	54	333	630	963	329	‡	769	2.2
waves	20	45	352	690	1042	623	220	707	3.0
triangle	42	80	315	1226	1541	1155	648	1036	2.0

† average path length for graph analyzers.
‡ bug in finite domain analyzer prevents calculation.

Table 3: Benchmark Suite Characteristics

the Mandelbrot function.

The algorithms tend to have complexity related to the number of symbols in the source program, which we categorize as constants (including functor symbols) and variable instances. Because paths can be cyclic, to calculate the number of paths, we break the cycle, e.g., the *car* and *cdr* of a list will be counted, but not the *cadr* or *cddr*. We list, in order, the number of paths produced by the graph analyzers (either static or active), the finite-domain analyzer and the theorem prover.

We observe that the average path lengths are not long because the largest hard-wired data structures within the benchmarks are not long. Path length can be arbitrarily manipulated by such data structures, e.g., the size of the input graph in **bestpath**. None of the analysis techniques have an effective method of dealing with this problem. Methods that heuristically simplify such large data structures risk loss of completeness. We also observe that whereas finite domain analysis and MGTP analysis are both incomplete, for the larger benchmarks, the finite domain method generates significantly fewer paths. This indicates that the finite domain method is *more* incomplete, i.e., since fewer paths are derived, there is a greater chance that a mode conflict will be missed.

All the benchmarks are fully moded except for **wave**, which has a subtle mode conflict. All the analyzers, except the finite domain method, detect the conflict. However, to calibrate the timing measurements presented in the next section, we continue analyzing the program even when a mode conflict is detected.

7.1 Execution Measurements

The analysis tools were all implemented in KL1 and run on the PDSS (V2.52.19) compiler-based system, on a Sun Sparcstation 10/30. PDSS is a nonoptimizing compiler-based emulator for KL1, executing about 34,000 reductions per second for the analyzers described here. Figure 4

	static graph	**active graph**	**finite domain**	**theorem prover**
parser	787	484	484	1,265
reducer	1,558	752	1,983	268
misc.	442	297	858	635
total	2,787	1,533	3,325	2,168

Table 4: KL1 Source Lines of Each Analyzer

gives the code sizes of the analyzers. The "reducer" size of the theorem prover is the MGTP kernel prover: it does not count the meta-program being analyzed. One must excuse the larger sizes of the static graph and finite domain analyzers: they were written by novices, whereas the other analyzers were written by ourselves. The output produced by all the analyzers for all the benchmarks was verified as identical (modulo slight syntactic differences) by shell scripts written in UNIX.

It should be noted that no "early" termination is detected by any of the analyzers. Such early termination might be defined when all top-level variable nodes (i.e., variable occurrences appearing in the source program) have unique modes. Such an approach is problematic because a concurrently executing unification may contradict the known modes, resulting in global failure. In other words, our experiments pessimistically evaluate all necessary unifications to guarantee full modedness.

Table 5 shows the breakdown, by phase, of the execution times (msec) of the benchmarks for the static and active graph analyzers. Also given are the total number of KL1 reductions and abstract unifications executed. Both analyzers use phase I to parse the program. The static analyzer reduces tell unification nodes in phase II (a minor task), and reduces the rest of the graph in phase III. The active analyzer reduces the entire graph in one shot during phase II.

The vast difference in the performance of these analyzers is primarily because a novice wrote one and an expert the other. We believe a "memory leak" in the static analyzer (i.e., incremental garbage collection cannot be done because of sloppy naming) is causing much of this difference. Neither analyzer has been tuned for performance, and we expect both can be significantly optimized. For reasons discussed in Section 4, we expect the active analyzer to achieve better parallel speedup than the static analyzer, but we have not yet conducted multiprocessor experiments.

Table 6 shows the breakdown, by phase, of the execution times (msec) of the benchmarks for the finite domain and MGTP analyzers. For finite domain analysis, phase I is path creation and subsequent phases are applications of Ueda's mode rules (phase II applies rules 1 and 3; phase III applies rule 2; phase IV applies rule 4). For MGTP analysis, phase I translates the source program into a set of MGTP clauses and further, MGTP's pre-processor converts the MGTP clauses into a KL1 program with which the MGTP engine generates mode information. Phase II compiles the KL1 program (using the PDSS cross-compiler written in SICStus Prolog) and links it with the MGTP engine. Phase III performs

Static Graph

	execution time (msec)				red		red/
benchmark	phase I	phase II	phase III	total	x1000	unify	unify
msort	300 (39.5%)	50 (6.6%)	410 (53.9%)	760	33	60	550
queens	550 (45.5%)	10 (0.8%)	650 (53.7%)	1,210	54	75	720
cubes	620 (35.8%)	30 (1.7%)	1,080 (62.4%)	1,730	82	138	594
pascal	890 (36.9%)	210 (8.7%)	1,310 (54.4%)	2,410	110	124	887
mandel	1,480 (39.8%)	240 (6.5%)	2,000 (53.8%)	3,270	171	222	770
rucs	1,290 (20.3%)	270 (4.2%)	4,810 (75.5%)	6,370	302	319	946
bestpath	2,080 (18.5%)	180 (1.6%)	8,960 (79.9%)	11,220	526	431	1220
waltz	2,290 (19.5%)	180 (1.5%)	9,260 (78.9%)	11,730	573	404	1418
waves	3,540 (21.7%)	510 (3.1%)	12,270 (75.2%)	16,320	699	559	1250
triangle	6,190 (12.0%)	60 (0.1%)	45,190 (87.8%)	51,440	1488	1163	1279
arith mean	(28.9%)	(3.5%)	(67.5%)				884

Active Graph

	execution time (msec)			red		red/
benchmark	phase I	phase II	total	x1000	unify	unify
msort	320 (65.3%)	170 (34.7%)	490	19	60	317
queens	430 (69.4%)	190 (30.6%)	620	21	75	280
cubes	750 (69.4%)	330 (30.6%)	1,080	40	138	289
pascal	740 (68.5%)	340 (31.5%)	1,080	36	124	290
mandel	1,240 (70.9%)	510 (29.1%)	1,750	62	222	279
rucs	1,670 (71.1%)	680 (28.9%)	2,350	74	319	232
bestpath	2,070 (57.0%)	1,560 (43.0%)	3,630	142	431	329
waltz	2,830 (49.6%)	2,870 (50.4%)	5,700	241	404	597
waves	2,910 (62.0%)	1,780 (38.0%)	4,690	182	559	326
triangle	4,900 (50.7%)	4,760 (49.3%)	9,660	370	1163	318
arith mean	(63.4%)	(36.6%)				326

Table 5: Performance of Static and Active Graph Analyzers (KL1 on Sun Sparcstation 10/30)

Finite Domain

	execution time (msec)					red
benchmark	**phase I**	**phase II**	**phase III**	**phase IV**	**total**	**x1000**
msort	660 (61.1%)	50 (4.6%)	310 (28.7%)	60 (5.6%)	1,080	34
queens	1,350 (61.6%)	110 (5.0%)	660 (30.1%)	70 (3.2%)	2,190	64
cubes	1,730 (57.1%)	110 (3.6%)	720 (23.8%)	470 (15.5%)	3,030	80
pascal	3,120 (66.2%)	170 (3.6%)	1,230 (26.1%)	190 (4.0%)	4,710	123
mandel	8,210 (45.6%)	410 (2.3%)	3,430 (19.0%)	5,960 (33.1%)	18,010	572
rucs	6,410 (80.4%)	440 (5.5%)	1,020 (12.8%)	100 (1.3%)	7,970	208
bestpath	12,080 (58.2%)	700 (3.4%)	6,090 (29.4%)	1,870 (9.0%)	20,740	489
waltz	10,810 (56.1%)	350 (1.8%)	4,210 (21.8%)	3,910 (20.3%)	19,280	530
waves	22,600 (60.2%)	1,120 (3.0%)	12,150 (32.4%)	1,670 (9.0%)	37,540	853
triangle	29,440 (55.6%)	810 (1.5%)	16,670 (31.5%)	6,000 (11.3%)	52,920	1196
arith mean	(60.2%)	(3.4%)	(25.6%)	(10.8%)		

Model Generation Theorem Prover

	execution time (msec)				red
benchmark	**phase I**	**phase II**	**phase III**	**total**	**x1000**
msort	600 (6.3%)	8,750 (93.1%)	50 (0.5%)	9,400	28
queens	920 (7.0%)	12,170 (92.3%)	100 (0.8%)	13,190	45
cubes	1,560 (7.0%)	19,430 (86.8%)	1,400 (6.3%)	22,390	156
pascal	1,150 (7.4%)	13,840 (89.2%)	530 (3.4%)	15,520	87
mandel	2,700 (7.1%)	34,410 (90.8%)	800 (2.1%)	37,910	169
rucs	1,410 (8.6%)	14,770 (90.6%)	130 (0.8%)	16,310	66
bestpath	4,340 (6.3%)	52,080 (76.0%)	12,140 (17.7%)	68,560	1029
waltz	3,670 (8.4%)	38,140 (86.8%)	2,120 (4.8%)	43,930	303
waves	4,670 (7.7%)	51,940 (85.2%)	4,330 (7.1%)	60,940	511
triangle	14,060 (8.4%)	149,060 (89.3%)	3,810 (2.3%)	166,930	901
arith mean	(7.7%)	(86.7%)	(5.6%)		

Table 6: Performance of Finite Domain and MGTP Analyzers (KL1 on Sun Sparcstation 10/30)

benchmark	PDSS compile	static graph	active graph	finite domain	theorem prover
			exection time (msec)		
msort	760	760	490	1,080	9,400
queens	1,140	1,210	620	2,190	13,190
cubes	1,570	1,730	1,080	3,030	22,390
pascal	1,660	2,410	1,080	4,710	15,520
mandel	3,340	3,720	1,750	18,010	37,910
rucs	3,010	6,370	2,350	7,970	16,310
bestpath	6,160	11,220	3,630	20,740	68,560
waltz	4,510	11,730	5,700	19,280	43,930
waves	7,960	16,320	4,690	37,540	60,940
triangle	11,720	51,440	9,660	52,920	166,930
	PDSS	static graph	active graph	finite domain	theorem prover
			memory consumption (kbytes)		
msort	158	795	281	381	198
queens	244	1,444	340	692	261
cubes	320	2,070	571	1,057	491
pascal	343	3,272	574	1,362	383
mandel	616	6,561	880	15,204	650
rucs	699	8,224	1,343	1,838	374
bestpath	922	17,298	1,779	8,501	2,023
waltz	803	17,696	2,278	6,394	831
waves	1,204	28,371	2,318	9,098	1,323
triangle	1,865	51,656	4,437	17,339	1,845
analyzers	273	65	50	44	74

Table 7: Performance of Mode Analyzers (KL1 on Sun Sparcstation 10/30)

the MGTP proof and generates the mode information.[10]

Curiously compilation takes the lion's share of 89% of the MGTP execution time. Only 3% of execution time is attributed to doing the proof, compared to 40% of the finite domain execution time. If MGTP compilation could be sufficiently sped up, this technique would be faster than the finite domain method.

The performance of these runs, for all four analyzers, is summarized in Table 7. The table gives the execution time and memory (data and code) consumption for each input source program. Interestingly, MGTP is most space efficient, comparable with the PDSS compiler.[11] It competes favorably with the active graph analyzer for the reduction phase, but loses overall because of its long translation time. Although the active graph analyzer

[10]MGTP reductions counts for are for phases I and III only.

[11]The PDSS compiler is relatively fast considering that is does not do any static analysis. For example, the Monaco optimizing compiler [18] is considerably slower because it does data flow analysis. Thus because mode analysis can compare reasonably well with naive compilation, it has even higher utility with respect to more realistic optimizing compilation.

does not use memory effectively because of its process network, stream mergers, and numerous messages, it executes most quickly, comparable to PDSS compilation time. The MTGP analyzer slows down for larger benchmarks because the translation procedure is more complex with respect to program size than is parsing in the other analyzers. Clearly the greatest performance improvements lie in reducing compilation time for MGTP analysis, and reducing memory consumption for active graph analysis. Currently, the large memory requirements of the finite domain and static graph analyzers make them less desirable.

8 Summary and Conclusions

This paper described and compared four compile-time analysis algorithms, based on seminal work by Ueda and Morita [21], for deriving the *path modes* of concurrent logic programs. The analyses are based on constraint propagation over graphs, path partitioning, and model generation theorem proving (MGTP). The former two techniques are complete in the sense that they are guaranteed to detect a mode contradiction in the source program should one exist. The latter two techniques are incomplete, although to varying degrees. The MGTP analyzer is more robust than the simplistic finite domain analyzer because the limitations placed on axiom nesting can be incrementally relaxed. There is no corresponding fine-tuning for finite domain analysis as we have defined it.

We are currently working on a completeness proof for the graph analysis algorithm, which is clearly needed before it can be used in many applications. However, all the analyzers can be used when complete mode information is not required, for example in the Diadora model [10], a variant of "lazy task creation" which can break a deadlocking thread inadvertently created by faulty mode information. Considering that unrestricted use of array referencing will make any mode analysis technique incomplete, we may need to address the completeness issue both from restricted language semantics and novel runtime systems.

Future work is proceeding along several dimensions. We seek to engineer an analyzer that can pinpoint probable causes of a mode conflict more accurately than the current systems, for use in programmer debugging. We seek to design extended mode domains that capture more subtle nuances of producer–consumer relationships. Furthermore, the use of vector builtins, which cause notorious aliasing, and the recovery of partially correct mode information from a nonmoded program (by propagating "top") need to be addressed.

Acknowledgements

E. Tick was supported by a research grant from ICOT, and an NSF Presidential Young Investigator award, with matching funds generously provided by Sequent Computer Systems Inc. We thank B. Massey and P. Tulayathun for their excellent effort in implementing two of the mode analyzers. We also thank T. Fujise, K. Kumon and K. Ueda for their helpful discussions. The key idea in the process network implementation was due to K. Kumon.

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] K. L. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 84–139. MIT Press, Cambridge MA, 1987.

[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*, 5(3):207–229, September 1988.

[5] H. Fujita and R. Hasegawa. A Model Generation Theorem Prover in KL1 Using Ramified-Stack Algorithm. In *International Conference on Logic Programming*, pages 535–548. Paris, MIT Press, June 1991.

[6] A. King and P. Soper. Schedule Analysis of Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 478–492. Washington D.C., MIT Press, November 1992.

[7] B. C. Massey. Sequentialization of Parallel Logic Programs with Mode Analysis. Master's thesis, University of Oregon, September 1992. Also available as Technical report CIS-TR-92-18.

[8] B. C. Massey and E. Tick. Sequentialization of Parallel Logic Programs with Mode Analysis. In 4^{th} *International Conference on Logic Programming and Automated Reasoning*, number 698 in Lecture Notes in Artificial Intelligence, pages 205–216, St. Petersburg, July 1993. Springer-Verlag.

[9] B. C. Massey and E. Tick. Demand-Driven Execution of Concurrent Logic Programs. In *International Conference on Parallel Architectures and Compilation Techniques*, Montreal, August 1994. North-Holland. Submitted. Also available as University of Oregon Technical Report CIS-TR-94-03.

[10] B. C. Massey and E. Tick. The Diadora Principle: Efficient Execution of Fine-Grain, Concurrent Languages. In *Hawaii International Conference on System Sciences*, volume 2, pages 396–404, Maui, January 1994. IEEE Computer Society Press.

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

[12] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* MIT Press, Cambridge MA., 1989.

[13] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[14] Z. Symogyi. A System of Precise Modes for Logic Programs. In *International Conference on Logic Programming*, pages 769–787. University of Melbourne, MIT Press, May 1987.

[15] E. Tick. *Parallel Logic Programming.* MIT Press, Cambridge MA., 1991.

[16] E. Tick. Practical Static Mode Analyses of Concurrent Logic Languages. In *International Conference on Parallel Architectures and Compilation Techniques*, Montreal, August 1994. North-Holland. Submitted. Also available as University of Oregon Technical Report CIS-TR-94-02.

[17] E. Tick. The Deevolution of Concurrent Logic Programming Languages. *Journal of Logic Programming*, 1994. Invited for Tenth Anniversary Issue.

[18] E. Tick and C. Banerjee. Performance Evaluation of Monaco Compiler and Runtime Kernel. In *International Conference on Logic Programming*, pages 757–773. Budapest, MIT Press, June 1993.

[19] E. Tick, B. C. Massey, F. Rakoczi, and P. Tulayathun. Concurrent Logic Programs *a la mode*. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems.* Kluwer Academic Publishers, 1994.

[20] K. Ueda and M. Morita. Message-Oriented Parallel Implementation of Moded Flat GHC. In *International Conference on Fifth Generation Computer Systems*, pages 799–808, Tokyo, June 1992. ICOT.

[21] K. Ueda and M. Morita. Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, May 1994.