Practical Static Mode Analyses of Concurrent Logic Languages

Evan Tick

CIS-TR-94-02
February 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. One solution is *path analysis*. In this paper we first motivate the analysis with performance results from our experimental FGHC-to-C compiler that sequentializes entire concurrent FGHC programs into C programs. We then review Ueda and Morita's proposed rational-tree mode propagation method, and present three novel algorithms for realizing the technique. We present empirical measurements of the analysis times of a benchmark suite which indicate that the analysis can be comparable to the compilation time on a simple, non-optimizing compiler. This study presents the first empirical results concerning the practicality of mode analysis for CLPs in conjunction with a system using the information to achieve significant speedups of source programs.

A shortened version of this paper appears in the *International Conference on Parallel Architectures and Compilation Techniques*, Montreal, August 1994.

Department of Computer and Information Science
University of Oregon

1 Introduction

Concurrent logic programs, such as FGHC [12], as well as other dataflow languages such as Id, are collections of fine-grain concurrent tasks that synchronize implicitly when input data in not available to a procedure invocation. Naive compilation causes too many, too small tasks to be generated. Improving execution performance by static partitioning into threads is a currently developing technique e.g., [6, 16]. In contrast to functional languages, logic languages have an additional problem that data dependencies are implicit, i.e., the input/output characteristics of program variables are not explicitly declared or trivially derivable. Conservative knowledge of dependencies is necessary to avoid placing a cycle within a thread thereby causing erroneous deadlock.

There are numerous methods for automatic derivation of mode information from sequential logic programs, e.g., [1, 4, 5]. Another option is user declarations [3, 13], which we consider either incomplete or too much burden on the programmer. For CLPs, Ueda and Morita [17] proposed a mode analysis scheme 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. This analysis is subtlely different than classical abstract interpretation, which could also be used to generate identical information.

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 and is thus likely to be erroneous. In this regard, the analysis constitutes a type restriction on the language, called "moded" flat committed-choice logic programs by Ueda and Morita. These are programs in which a variable has at most a single producer and 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 can be transformed to moded form in a straightforward fashion.

To motivate the importance of the static analysis, consider Table 1 comparing the performance of our experimental FGHC-to-C compiler [8] using these techniques, to traditional multiprocessor and uniprocessor systems that are in common use. All measurements were made on a 20-processor Sequent Symmetry running DYNIX V3.2.0. PDSS [2] is the compilation-based FGHC system under which the analyzers described in this paper runs; the PDSS system emulates parallel scheduling and execution order on a single processor. Monaco [14] is a research compiler which produces native code executables capable of utilizing an arbitrary number of the processors of the Symmetry in a shared-memory multitasking fashion. The sequential ANSI C code generated by our sequentializer (labeled "sequential C") was compiled using GCC 2.2.2 at optimization level 2 but with no other special optimizations. Strand and JAM Parlog are other emulator-based compilers for CLP languages — the benchmark was slightly different under these implementations to accommodate mi-

System	PEs	Problem Size	Time (sec)
PDSS	1	125	1.4
		250	5.2
		500	20.5
Monaco	1	500	10.5
	4	125	0.20
		250	0.75
		500	2.9
	8	500	1.5
sequential C	1	125	0.14
		250	0.58
		500	2.2

System	PEs	Problem Size	Time (sec)
Strand	1	500	10.8
JAM Parlog	1	500	13.1
handcrafted C	1	125	0.10
		250	0.40
		500	1.5

Table 1: Performance of Sequentialized and Parallel QuickSort

nor language differences. The handcrafted C code was written to mimic the Quicksort benchmark algorithm while retaining a natural C style — in particular, parameters were passed with appropriate types whenever possible.

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 [11], is needed to retain multiple threads for parallelism, mode analysis is still required for safety.

In this paper we discuss three algorithms for implementing Ueda and Morita's technique: the first implementations yet built and evaluated.[1] We present empirical measurements of the analysis times of a benchmark suite which indicate that the analysis is comparable to compilation time on a simple, non-optimizing compiler.

2 Background: Paths and Modes

Ueda and Morita's [17] 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* (for a precise definition, see Ueda and Morita [17]). For example the `cadr` of the first argument of procedure q has an input mode specified as: $m(\{< q, 1 >, < ./2, 2 >, < ./2, 1 >\}) = in$.

[1] K. Koshimura is currently building an analyzer based on the Model Generation Theorem Prover (MGTP) [7].

All analyses presented in this paper exploit the rules outlined by Ueda and Morita [17]. For the purposes of following this paper, we grossly summarize these as three rules: §1 nonvariables and variables constrained in certain ways in the guard are immediately deduced as *in*; §2 corresponding paths among left and right side of body ("tell") unification goals have *opposite* modes, and §3 variables have at most one producer thus multiple occurrences have at most one *out* instance.[2]

3 Constraint Propagation Algorithm

In the constraint propagation algorithm [15], a graph[3] is constructed representing the entire program.[4] Analysis proceeds by unifying all roots with the same functor and arity. Termination, occurring when no further reduction is possible, is guaranteed because the mode lattice is finite and monotonic. Checking for program "well modedness," the analysis cannot terminate even if all modes are derived, in anticipation of a later contradiction.[5] Thus time complexity is simply a function of the size of subgraphs to be unified,[17] which are usually small.

3.1 Data Structures

A program graph is a directed, multi-rooted, (possibly) cyclic graph composed of two types of nodes. To illustrate the following definitions, Figure 1 presents a portion of the program graph for 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. □

> *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] This is not entirely precise: a variable produced in the body (*out*) can be exported in the head (also *out*).

[3] Throughout the paper we informally refer to rational trees as graphs. Note that our graph grammar is quite different than that of Ueda and Morita.

[4] It is straightforward to build an incremental analyzer; we do not go into details here.

[5] A program is "fully-moded" or "well-moded" if the modes of *all* paths are known, "moded" if the modes of *some* paths are known, and "non-moded" if there is a mode *contradiction*.

```
q( T, Y, Z ) :- T = [ X | Xs ] |
    U = [ X | Vs ],
    s( Xs, X, L, G ),
    q( L, Y, U ),
    q( G, Vs, Z ).
```

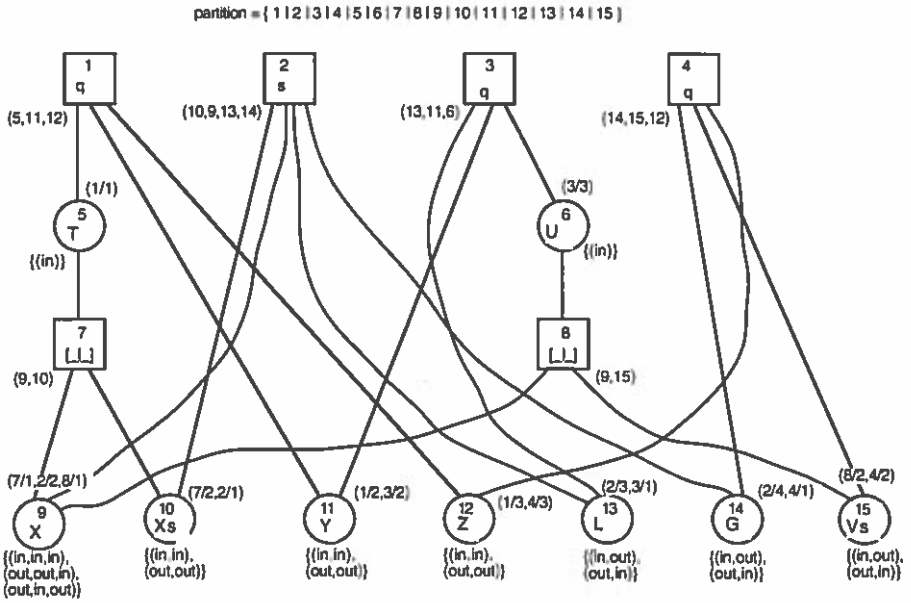

Figure 1: Initial Graph of Procedure q, After Phases I-II

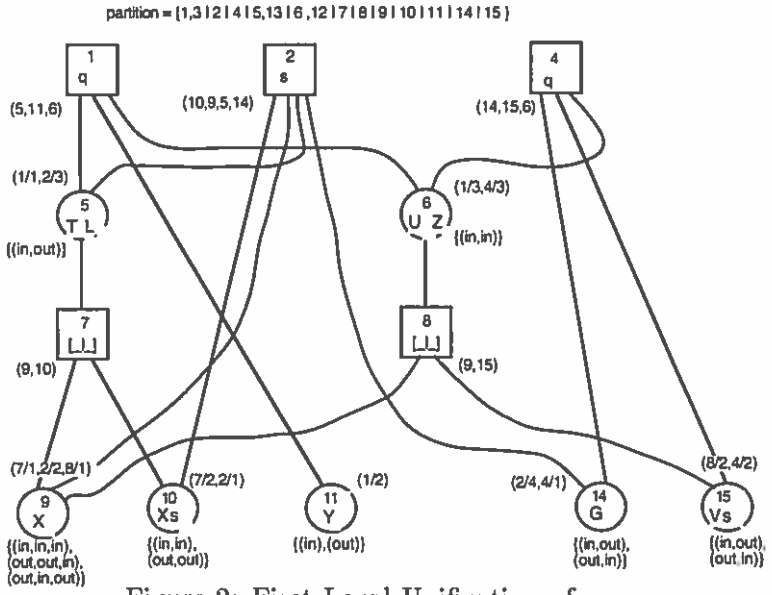

Figure 2: First Local Unification of q

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.[6] 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. □

Initial graphs, e.g., Figure 1, are multi-rooted directed acyclic graphs. The initial roots correspond to clause head functors, body goal functors, and body 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. Phases I and II (shared by the process network implementation) convert a flat committed-choice program into a normalized graph with roots named only by clause heads and user-defined body goals. Precise details of this transformation can be found in [15]. Phase III is described next.

3.2 Abstract Unification

Throughout this discussion of unification, consider the example shown in Figure 1 (initial) and Figure 2 (after unification of roots 1 and 3). See [15] for details. Figure 3 gives the graph unification algorithm. 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.

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.

[6] The size of m increases with the complexity of the rules, e.g., rule §3 (Section 2) can produce several vectors. By explicitly enumerating initial modes, we simplify the analysis by obviating the need to actively apply complex constraints implied by rule §3 throughout reduction of the graph.

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

sunify(a, b) {
      if a.fun ≠ b.fun
            return(failure)
      if mark({a,b}) clear then {
            set mark({a,b})                                            (2)
            ∀k ∈ [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 × b.modes
            p' = {(s,t) ∈ p  |  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)  |
                        (s,t) ∈ 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 ∪ b.out
            f_a = {s.fun  |  s ∈ a.out}
            f_b = {s.fun  |  s ∈ b.out}
            i = {s  |  fun ∈ f_a ∩ f_b,  s.fun ∈ u}
            c.out = a.out ∪ i                                          (7)
            — unify children with the same functor/arity
            ∀(x,y)  |  x ∈ a.out, y ∈ b.out  {
                  if x.fun = y.fun
                        if sunify(x, y) failure then                   (8)
                              return(failure)
            }
      return(success)
  }   }
```

Figure 3: One-Pass Rational-Tree Unification Algorithm

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. 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.

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.

Translating the previous algorithm (Figure 3) 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[7] 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.

For lack of space we do not discuss additional messages for performing output and termination.

The implementation shared phases I and II with the previous algorithm, and then spawns a node process network from the static graph definition. Similar to the static graph analyzer, 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.

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 3. 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.

[7] See Shapiro [12] for discussion of CLP programming techniques such as short circuits and back messages.

4.3 Implementation and Comparison

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 6). 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 [8]. 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 our 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.

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 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., $\{<=, 1 >\}\cdot s$).

In general, all interesting paths of a program are generated in a few sequential passes. For example, the 39 interesting paths of Quicksort are generated in two passes. The interesting paths could be generated from a depth-one traversal of the complete Quicksort graph, except for two paths which are "hidden" because they cannot be derived locally. 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. This is not always the case!

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, and 4) Repeatedly consider multiway relations (rule §3 Section 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 2–7% of total analysis execution time in simple programs, and 11–20% in complex programs.

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

program	proc	clause	symbols			broken paths	
			const	vars	total	total	avg length
qsort	2	5	31	40	71	19 (17)	1.6
primes	6	12	49	63	112	33 (28)	1.5
msort	4	11	54	75	129	36 (30)	1.7
queens	6	14	77	119	196	71 (43)	1.8
cubes	9	16	93	159	252	224 (79)	2.7
pascal	11	22	143	200	343	338 (56)	2.0
rucs	16	66	218	390	608	79 (46)	1.6
bestpath	20	44	279	492	771	507 (207)	2.5
waltz	20	54	333	630	963	329 ( )	2.2
waves	20	45	352	690	1042	623 (220)	3.0
triangle	42	80	315	1226	1541	1155 (648)	2.0

Table 2: Benchmark Suite Characteristics

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.

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. The 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 our analysis, and if the program being analyzed is known to be moded, all modes derived are correct. Assuming it can be made faster than safe analyses, unsafe analysis has utility for "lazy task creation" systems [9, 10] where cycles can be broken at runtime.

6 Performance Comparison

A benchmark suite of KL1 source programs (Table 2) was analyzed using the three mode analyzers. The analysis tools were all implemented in KL1 and run on the PDSS (V2.52.19)

benchmark	PDSS compile	finite domain	static graph	active graph	finite domain	static graph	active graph
	exection time (msec)				memory consumption (kbytes)		
qsort	410	460	390	330	205	695	321
primes	780	960	730	450	320	1,919	454
msort	760	1,100	940	570	381	2,358	519
queens	1,140	2,230	1,510	890	692	5,886	897
cubes	1,570	3,000	2,260	1,100	1,057	10,226	1,086
pascal	1,660	4,820	3,830	1,530	1,362	24,678	1,739
rucs	3,010	8,130	17,830	2,690	1,838	119,774	2,517
bestpath	6,160	21,110	32,780	4,280	8,564	237,866	5,074
waltz	4,510	21,070	36,240	3,930	6,683	279,132	5,704
waves	7,960	37,490	88,500	6,230	9,107	526,110	8,406
triangle	11,720	53,380	567,970	12,100	17,339	1,600,770	19,373
analyzers					44	54	70

Table 3: Performance of Mode Analyzers (KL1 on Sun Sparcstation 10/30)

compiler-based system, on a Sun Sparcstation 10/30. PDSS executes about 34,000 reductions per second for the analyzers described here. The analyses tend to have complexity related to the number of symbols in the source program [17], which we categorize as constants (including functor symbols) and variable instances. Because paths can be cyclic, we define the number of "broken" paths, e.g., the `car` and `cdr` of a list will be counted, but not the `cadr` or `cddr`. We list, in parentheses, the number of paths produced by the finite-domain analyzer, since it may differ from the other three algorithms.

The execution performance of these runs is summarized in Table 3. The table gives the execution time (an average over five runs) and data memory consumption for each input source program. The last row gives the static code size of the tools themselves. The broken path output of the analyses was verified as identical modulo the incomplete nature of the finite domain method.

There are several interesting observations supported from the empirical measurements:

- Programs such as cubes and waltz contain ground lists of data that increase the analysis complexity by lengthening the propagated paths. Although we can hope that a ground data list of length one holds as much information as length 100, there is always an outside chance that the 99^{th} element will cause a contradiction somewhere in the program. We are developing a method wherein we can cut ground terms and then do post-analysis to ensure that we did not miss a contradiction.

- Development of the analyzers by novice programmers indicated some weaknesses of the KL1 development environment. Notably, even after tuning, the finite domain analyzer is still generating excess suspensions and the static graph analyzer has a memory leak, albeit less significant than before. For example, although graph construction required 35% of total analysis time in the static graph analyzer, it was proportionally 68% in

the faster active graph analyzer (written by an expert). This leads us to believe that the analyzers can be further tuned.

- Finite domain analysis does not appear faster than constraint propagation and therefore its utility is questionable. Although the reduction in paths decreases memory consumption slightly, it can in some cases produce *more* paths than are necessary, and in other cases delay resolution of multiway relations.

- The active graph analyzer demonstrates execution times ranging from 58% to 103% of the PDSS compiler. The arithmetic mean over the benchmarks is 80%. For large programs, garbage collection remains problematic and thus we must throttle task creation.

7 Conclusions

We have described three alternative algorithms for rational-tree unification for the derivation of path modes in CLPs. We showed that mode analysis time was comparable to compilation time, which we consider reasonable, especially since Monaco [14], our native-code optimizing compiler (doing dataflow analysis, register allocation, etc.) has significantly slower compilation than PDSS.

One use of the analysis is for partitioning fine-grain programs into higher-granularity threads, as illustrated by the experimental FGHC-to-C compiler we developed [8]. The main alternative is abstract interpretation, for example as examined by King and Soper [6]. For our future work, we intend to extend the precision of the abstract domain to avoid spurious dependencies when logical variables are passed "through" a procedure invocation unbound (intuitively neither *in* nor *out*).

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with matching funds from Sequent Computer Systems Inc., and a grant from the Institute of New Generation Computer Technology (ICOT).

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, H. Sato, and T. Miyazaki. 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] 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.

[4] 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.

[5] S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, 5(3):207–229, September 1988.

[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] M. Koshimura and R. Hasegawa. A Mode Analyzer for FGHC Programs in a Model Generation Theorem Prover. In *Proceedings of the* 47^{th} *Annual Convention IPS Japan*, 1993. In Japanese.

[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. 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.

[10] E. Mohr, D. A Kranz, and R. H Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Computing*, 2(3):264–280, July 1991.

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

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

[13] 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.

[14] E. Tick. Monaco: A High-Performance Flat Concurrent Logic Programming System. In *PARLE: Conference on Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, pages 266–278. Springer Verlag, June 1993.

[15] 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.

[16] K. R. Traub and D. E. Culler. Global Analysis for Partitioning Non-Strict Programs into Sequential Threads. In *Conference on Lisp and Functional Programming*, pages 324–334. San Francisco, ACM Press, 1992.

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