Concurrent Logic Programs *a la Mode*

E. Tick, B. Massey, F. Rakoczi, and P. Tulayathun

CIS-TR-93-11
June 1993

Abstract

This paper describes and compares two compile-time analysis algorithms for deriving the *path modes* of a moded concurrent logic program. A path describes a subterm of a procedure argument. Deriving all path modes is a way to perform certain valuable optimizations, such as sequentialization of too-fine concurrent tasks, and scheduling to reduce suspension. We describe our own variation of Ueda and Morita's original constraint propagation scheme, which includes our novel circular unification algorithm. We also describe an alternative method of finite domain analysis that we developed. The two methods are critiqued and we show the relationship between them.

This paper appears in the *Proceedings of the Workshop on Practical Implementations and Systems Experience in Logic Programming*, Budapest, June 1993.

Department of Computer and Information Science
University of Oregon

1 Introduction

Mode information has been shown to be quite useful in the efficient compilation of logic programming languages. 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, 2, 3, 7]. 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.

We are interested in concurrent logic programs in the FCP(:, |) language family [8] 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 sample program used throughout the paper to illustrate the analysis techniques discussed herein. A sample query would be: ?- $q([2,1,3], Y, [])$, returning $Y = [3,2,1]$.

Ueda and Morita [11] 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.

As an alternative algorithm, we developed a projection of the previous work to a *finite domain of paths*. This has the nice property of obviating the need for unification over circular terms, thereby gaining some efficiency. We represent the relationships of a finite set of paths in such a way that all mode information directly available about this set of paths in a program may be efficiently derived. Again, moded programs are required.

The purpose of this paper is to describe and compare both algorithms in detail. As far as we know, this is the first document describing practical implementations for these mode analysis schemes. Section 2 describes our implementation of the constraint propagation algorithm. Section 3 describes our finite domain algorithm. In both sections we present examples to illustrate the concepts. The algorithms are compared in Section 4 and conclusions are summarized in Section 5.

2 Constraint Propagation Algorithm

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

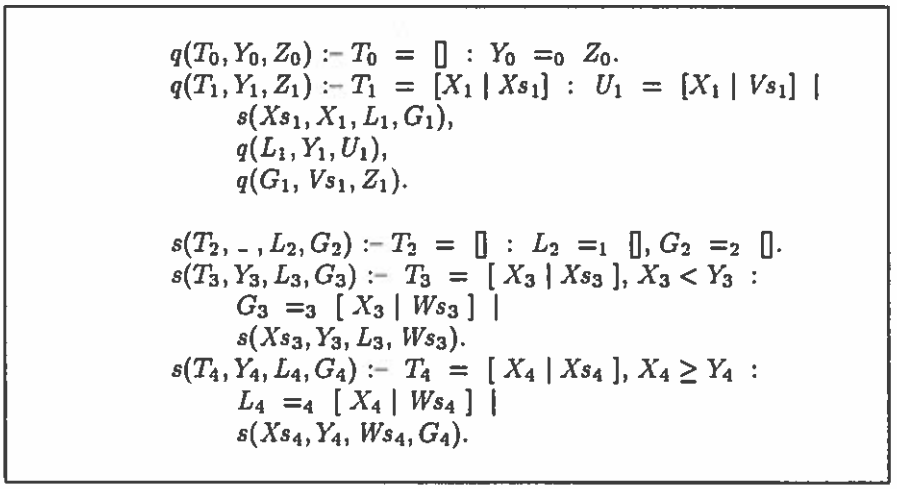

Figure 1: Quicksort FCP(:, |) Program: Normalized Form (Clauses 1–5)

consistent, where each path may have one of two modes: *in* or *out* (for a precise definition, see Ueda and Morita [11]).

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

Both 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). In the constraint propagation algorithm, a graph is constructed representing the entire program. 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 2.3. 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 2.3).

2.1 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).

> *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

§1. For some path p in a clause, $m(p) = \text{in}$, 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 $m(p_h) = \text{in}$

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

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

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

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

1. If the body contains more than two occurrences of the shared variable and the head has no occurrences, then one of the modes is 'out,' and the others are 'in.' 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)

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*).[1]

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

[1] 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 2.3).

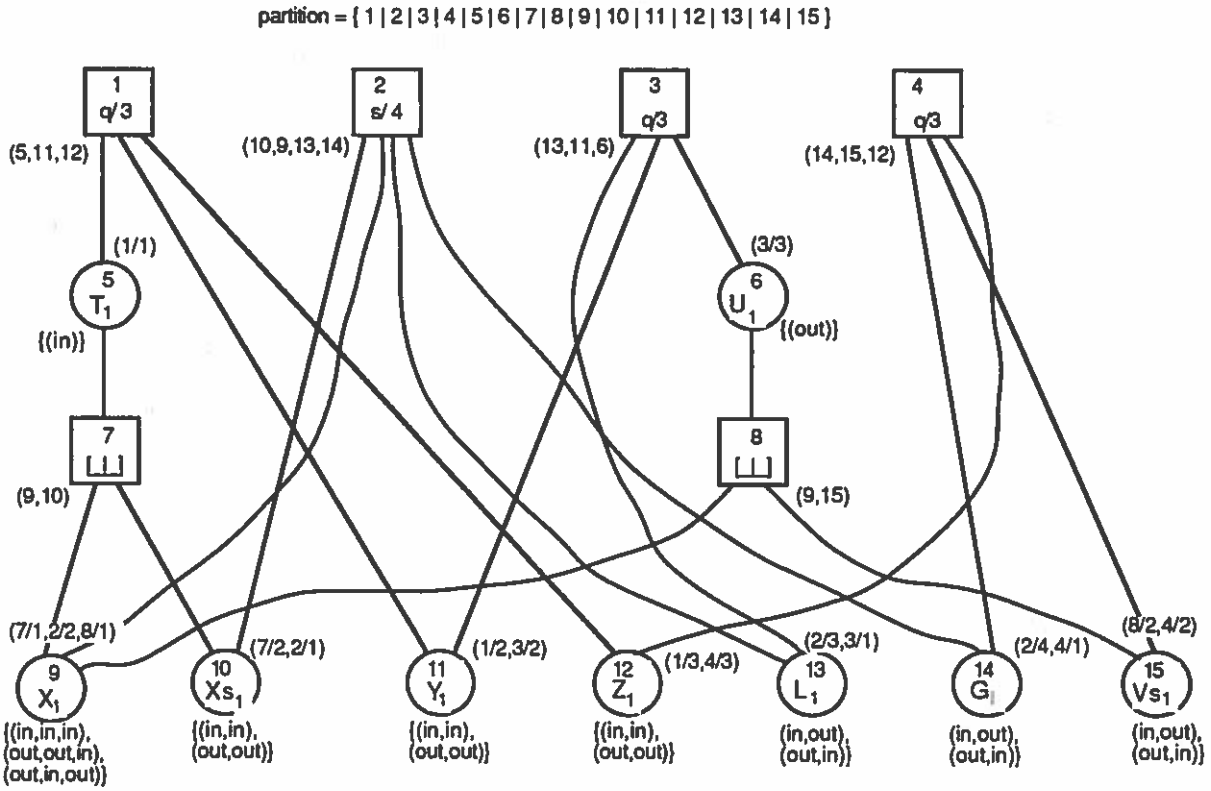

Figure 3: Initial Graph of Procedure $q/3$ (Clause 2), After Phases I-II

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.[2] 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 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. These are described in the following sections.

[2]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.

2.2 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 avoid this exception thereby facilitating the next phase, we introduce a *dummy* variable node as an intermediary between these two structure nodes. This is effectively a nameless placeholder that will be removed in phase II.

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. Here we describe only reduction of (both active and passive) unification operators because they are the most frequent.

First consider a (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.

Now consider a tell (active) 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). After phase II, builtin predicate nodes are no longer present in the graph. The resulting graph contains roots named only by clause heads and user-defined body goals.

> *Example*: Phase II analysis of $q/3$ clause 1 is illustrated in Figure 4. The modes of the tell unification operator must be opposite. Thus we remove the unification operator node, combining its operands, resulting in the graph shown in Figure 4b. That concludes the analysis for clause 1 because there are no body goals.
>
> Another example is shown in Figure 3, where the topology of node groups {5,7,9,10} and {6,8,9,15} result from the introduction of dummy nodes in phase I, followed by phase II analysis. □

2.3 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

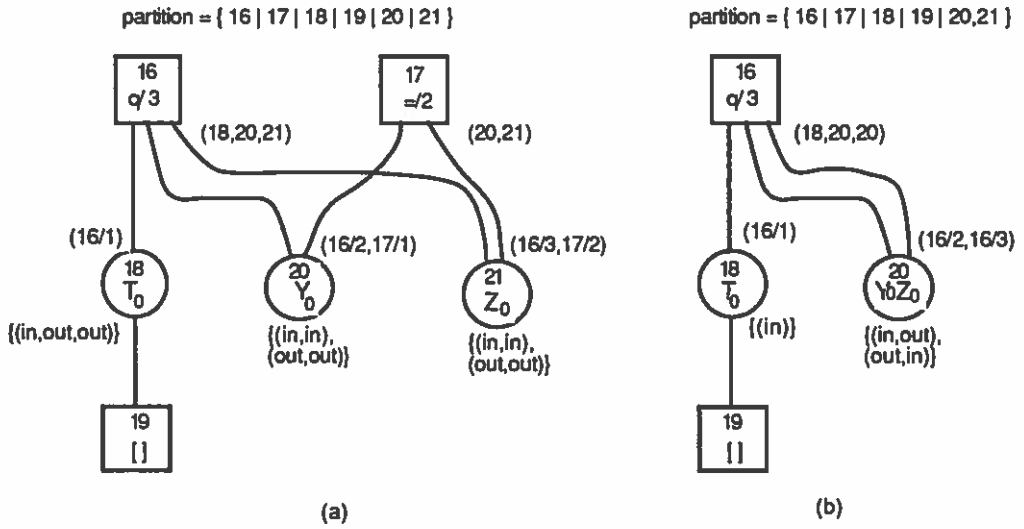

Figure 4: Builtin Tell Unification of Quicksort ($q/3$, clause 1)

different procedures. The abstract unification algorithm is the same, however, for any two (node) arguments, local or global.

Figure 5 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.[3]

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.

> *Example*: Consider two graphs representing $m(\{<\text{`.'}/2, 2>, p\}) = m(p)$ and $m(\{<\text{`.'}/2, 2>, <\text{`.'}/2, 2>, p\}) = m(p)$. The former has a 1-period cycle and the latter a 2-period cycle. Suppose that node 1 corresponds to the structure node in the shorter cycle, and nodes 2 and 3 are in the longer cycle. Unifying the roots, we first mark pair {1,2}, then mark {1,3}, then terminate when pair {1,2} is found to already be marked. □

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

[3]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 ≠ 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 5: One-Pass Graph Unification Algorithm

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.[4] 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 6.[5] That is we unify all three 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)} × {(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

[4] 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.

[5] By convention we remove nodes 3, 12, and 13 from the illustration to clarify that they represent redundant information. In the implementation, these nodes might be kept in the graph if space was not a critical resource.

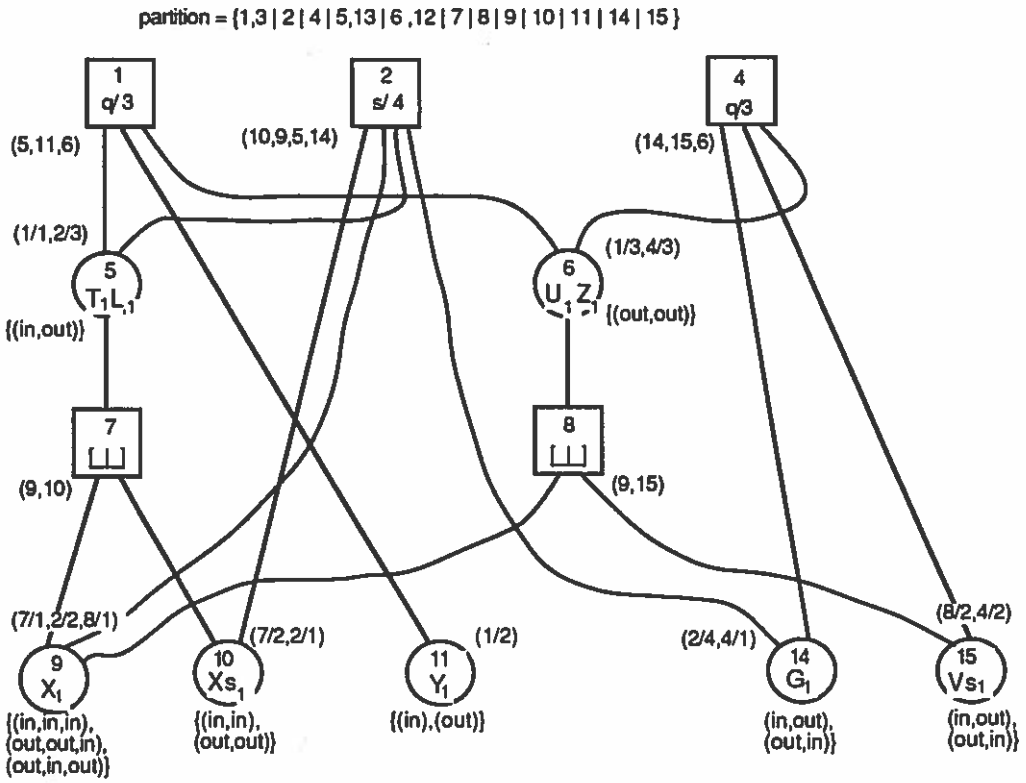

Figure 6: First Local Unification of $q/3$ (clause 2)

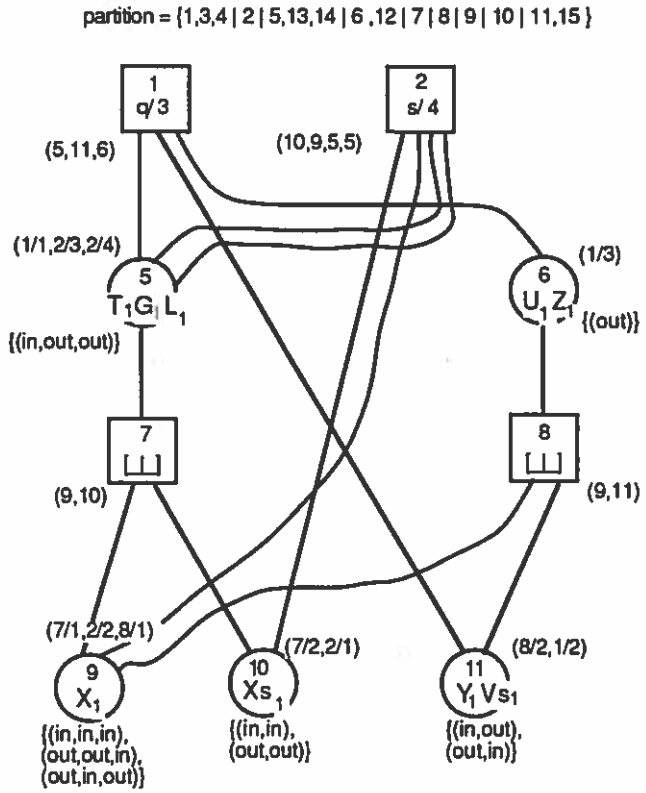

Figure 7: Second Local Unification of $q/3$ (clause 2)

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 6. Node 1 and node 4 are then unified, resulting in the graph shown in Figure 7. This is the final local unification possible for clause 2 since all roots now have unique functors.

We now do inter-clause analysis of $q/3$, unifying the first (root) nodes from the graphs represented in Figures 7 and 4b, resulting in the graph of Figure 8. This represents the most information that can be derived from $q/3$ alone. Note that the modes of X_1 and Xs_1 are still ambiguous. In normal operation, we would locally derive the modes for $s/4$, and then do global analysis, 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 9. □

3 Finite Domain Analysis

The first stage of the 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. 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 9. The interesting paths could be generated from a depth-one traversal of the graph, except for $\{< q/3, 2 >, < ., 1 >\}$ and $\{< q/3, 2 >, < ., 2 >\}$. These two paths are "hidden" because they cannot be derived from

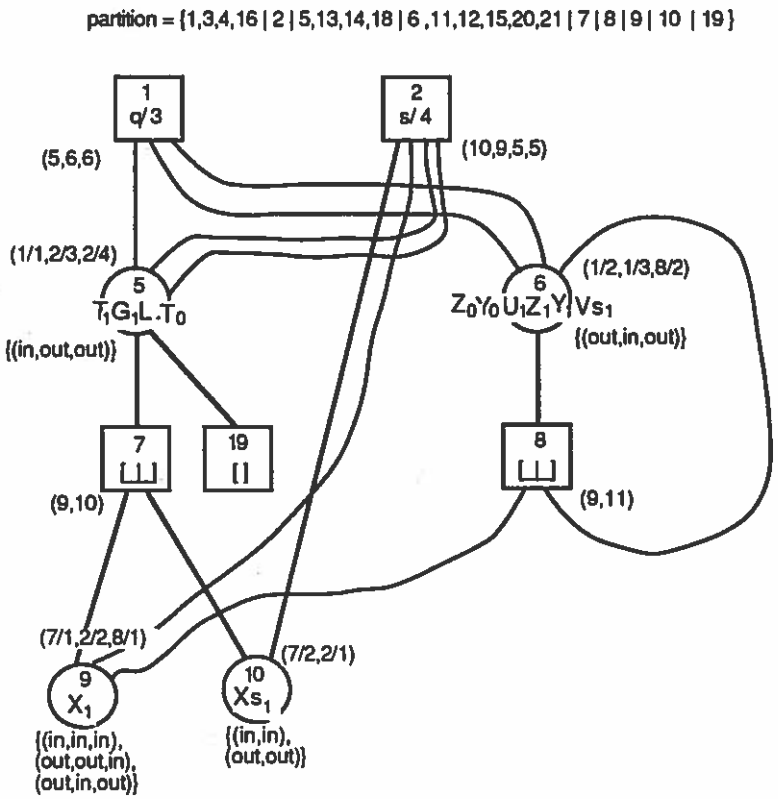

Figure 8: Global Unification of Quicksort $(q/3)$

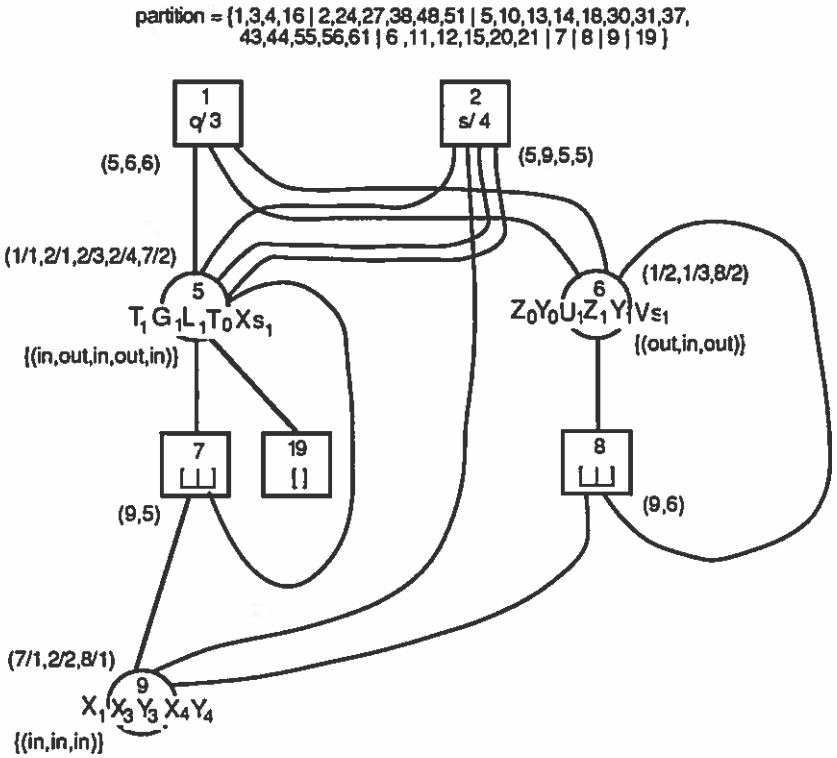

Figure 9: Minimized Graph of Quicksort

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

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

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, as discussed below, there is good reason to believe that some fundamentally important paths may not be generated.

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 2–7% of total analysis execution time in simple programs, and 11–20% in complex programs [4].

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. (We say that 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 our analysis, and if the program being analyzed is known to be moded, all modes derived are correct. Thus, our mode analysis algorithm is a practical tool for many compiler optimizations.

4 Critique

We believe that our algorithmic specification and implementation of the *constraint propagation method* has contributed in the following areas:

- The algorithm to be used is specified precisely. In particular, the data structures used by the algorithm and the method of unification of subtrees are specified in sufficient detail that the algorithms for encoding the program and determining its modes are always well-defined.
- Our implementation of the algorithm is undergoing benchmarking and appears to behave correctly.
- The algorithm seems amenable to a proof of completeness and correctness via structural induction, as it is specified in a relatively denotational style.

Nonetheless, there is still work to be done. Most importantly, a formal proof of the correctness of the algorithm should be given, and the implementation should be tested with a variety of real-sized programs.

The *finite domain method* avoids circular unification by sacrificing completeness. Our initial belief was that real programs have very short paths, so that the construction of "interesting" paths would cover most programs. From analysis of small benchmarks, we note that programs with deep paths are not altogether rare. In our system, the programmer must declare the modes of these deep paths that the analysis cannot derive on its own, if completeness is desired. In addition, the finite domain method may fail to note a moding contradiction, and thus silently produce unsound output for erroneous inputs.

To compare the utility of the two methods for a set of benchmark programs, we plan to generate paths from depth-one traversals of the complete graph and feed these sets to the finite path analyzer. For this we can compute the percentage of all syntactic variable modes determined unambiguously. For fully-moded programs this indicates how well the finite domain method approximates the constraint propagation method.

We believe that the graph algorithm is consistent, complete, and safe, although as noted above this remains to be proved. Thus, in applications requiring these properties, we recommend the use of the graph algorithm, as we currently know of no way to repair the finite domain algorithm, although it may be possible.

Note that if either algorithm reports a contradiction, there is no obvious way to automatically correct it, or even to determine the minimal subset of paths involved in the contradiction. Thus,

we restrict our attention to the language of fully-moded (and thus non-contradictory) programs, and check this constraint with the mode analysis algorithm. The current implementation will report any contradiction, ignore the contradictory assertion, and proceed with the derivation. This allows the user to examine the final modes produced by the analysis and determine which might be incorrect. In our experience, this is usually sufficient to correct the problem. In practice, in the absence of user intervention, this also allows the modes of most of the remaining paths to be determined.

We have recently developed an parallel execution model [6] which requires only a fairly precise approximation to the modes of a program to optimize the performance of the program — in particular, the technique requires neither complete nor correct mode information. We believe that our finite domain analysis may be an adequate choice for this case because it is simple to implement, efficient, and reasonably precise.

5 Conclusions

This paper informally describes two alternative algorithms for deriving, at compile time, the path modes of flat committed-choice languages. One contribution of the paper is to refine Ueda and Morita's proposed method of constraint propagation into an implementable and efficient algorithm. This includes our novel abstraction of unification over variable and structure nodes in the graph. In addition, we describe an alternative method based on a finite path domain, with the motivation of decreasing analysis time by avoiding cyclic unification. This comes however at the expense of completeness, because we cannot guarantee that all paths are constructed.

Future work in this area includes integrating these algorithms into the front end of our experimental FGHC compiler, Monaco [9, 10]. We have already experimented with code sequentialization via mode analysis [4, 5] and continue to explore this application. Another important problem to be solved is making the mode analysis algorithm *constructive* in the sense that it can point out probable causes (in the source code) of illegal moding.

Acknowledgements

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

References

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

[2] 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.

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

[4] 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.

[5] 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*, St. Petersburg, July 1993.

[6] B. C. Massey and E. Tick. The Diadorra Principle: Ėfficient Execution of Fine-Grain, Concurrent Languages. Dept. of Computer Science, University of Oregon, June 1993. *In progress.*

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

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

[9] E. Tick. Monaco: A High-Performance Flat Concurrent Logic Programming System. In *PARLE: Conference on Parallel Architectures and Languages Europe*. Springer Verlag, June 1993.

[10] E. Tick and C. Banerjee. Performance Evaluation of Monaco Compiler and Runtime Kernel. In *International Conference on Logic Programming*. Budapest, MIT Press, June 1993. Also available as University of Oregon CIS-TR-92-21.

[11] K. Ueda and M. Morita. Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, 1993. In press.