Path Mode Analysis for Concurrent Logic Programs

Putthi Tulayathun
University of Oregon

CIS-TR-94-13
June 1994

Abstract

Mode analysis of logic programs is the derivation of the input/output relations of logic variables and their embedded terms. Mode information is critical in enabling many types of compiler optimizations, for example thread partitioning. It is therefore useful to derive modes automatically at compile time, without assistance from the programmer. In this thesis, an automatic mode analyzer is described, based upon a static-graph reduction technique. This extends the seminal work of Ueda and Morita by presenting an implementation algorithm and its empirical performance for a benchmark suite. It is shown that the first mode analyzer prototype produces correct results in reasonable time.

This thesis has been submitted to the University of Oregon in fulfillment of a Masters of Science degree in the department of Computer Science.

Department of Computer and Information Science
University of Oregon

Acknowledgements

Thanks to my loving mother Suganda Tulayathun, who make my study and life experience in the U.S.A possible. Thanks to my advisor Dr. Evan Tick. Without him I never would even have gotten involved in this topic, and much of his generous support has enabled me to continue with this project. Thanks to Barton Massey. Without him I would have very difficult time doing my research. Thanks Miho Eto for the first draft of all the pictures in the thesis.

My study expense was fully funded by Mrs. Suganda Tulayathun.

Contents

1 Introduction **1**
- 1.1 Execution Model of FGHC 1
- 1.2 Thesis Overview 2
- 1.3 Thesis Outline 2

2 Mode Analyzer Algorithm **3**
- 2.1 Definitions 3
- 2.2 Static Graph Reduction Mode Analysis 6
- 2.3 Static Graph Building 6
- 2.4 Unification and System-Defined Function Reducing (Graph Reducing Phase I) 7
- 2.5 User-defined functors reducing (Graph Reducing Phase II) 10
- 2.6 Mode Consistency Check 16
- 2.7 Interpretation of the Final Graph 17
- 2.8 Replaying the Whole Process 17

3 Non-moded Programs **23**
- 3.1 Mode Consistency Checking in General 26
- 3.2 Simple Non-moded Program 27
- 3.3 Non-moded Program without Mode Consistency Checking 40
- 3.4 Non-moded Program with Mode Consistency Checking 40

4 Analysis Cost and Performance Evaluation **52**
- 4.1 Cost of Analysis 53
- 4.2 Performance Evaluation 56
- 4.3 Execution Measurements 56

5 Conclusions and Future Work **58**

References **62**

1 Introduction

In recent years, efficient compilation of concurrent and parallel logic programming languages has been a popular research area in the logic programming community. Mode information is useful because it leads to many optimization techniques. We are interested in a family of concurrent logic programming languages which contains Flat Guard Horn Clauses (FGHC) [12], Strand [2], and similar languages [7]. Mode information can be used in optimization techniques, specifically in code generation for this family of languages, such as:

- Logical variables are overloaded to control synchronization in the execution of these programs. Mode information can help to generate code so that the number of suspensions is reduced [13]. Thus the execution times of the programs are reduced.
- Mode information enables pure demand-driven execution [5], which is quite a useful optimization technique. With pure demand-driven execution, program execution will consume less memory and less processor resources.
- Static partitioning of a concurrent logic program uses mode information to find sets of execution threads that give higher granularity, reduce inter-process communication, and thereby achieve more efficient multiprocessor usage [1].

Moreover, mode information can also be used to detect some bugs in the source program at compilation. These bugs are usually caused by unconventional usage of variable bindings in a program.

Mode information is useful and enables many types of compiler optimizations. It is important to automatically derive mode information during program compilation. Our purpose is to implement an automatic mode analyzer that can be a part of the front-end of an optimizing compiler. In the next section, we briefly introduce the semantics of FGHC languages.

1.1 Execution Model of FGHC

A Flat Guard Horn Clause (FGHC) program is a committed-choice logic program. An FGHC program consists of a collection of clauses. Each clause is in the form: "$H :- A_1, ..., A_m : T_1, ..., T_n \mid B_1, ..., B_p$." where $m, n, p \geq 0$. H is the clause head; A is an ask guard goal; T is a tell guard goal; and B is a body goal. The operator ':' separates two kinds of guards, and the operator ' | ', which is called the commit operator, separates guards from the body of the clause. A procedure is defined by a set of clauses which have the same functor and arity for H. Of course, all the guards must be "flat." That means that a guard can only be a system-defined built-in function. The ask guards perform passive unification, *i.e.*, input matching. The tell guards perform active unification, *i.e.*, output binding. A clause that has no body ($p = 0$) is called a *unit clause.*

During procedure invocation, a goal (query) commits to a clause of the procedure by matching to the head arguments, using passive unification and satisfying the guard goals. If the goal can commit to more than one clause of the procedure, the goal commits to one

of them *nondeterministically*. The other candidate clauses are discarded. If the goal fails to match the head or satisfy the guards of any of the clauses, the goal fails. If a particular argument (or variable) of the goal is determined to be insufficiently instantiated during the commit process (*i.e.*, goal-head matching and guard satisfaction), the execution of the the goal is suspended. The suspension may be resumed later when the variable associated with the suspended invocation of the goal becomes sufficiently instantiated. Notice that the variables which are already sufficiently instantiated will never lose their values since variables are single assignment.

After the goal commits to a clause, a process is created for each of its body goals. All of these new processes are put in the *process pool*, except the first body goal process which is executed immediately. The processes in the process pool will be picked to be executed concurrently by the scheduler. Again, the synchronization of program execution is controlled by means of overloading logical variables. The situation where the process pool is empty and at least one process is suspended is called *deadlock*. If the process pool is empty and no suspended process exists, the execution succeeds (returns *true*), and the program successfully terminates.

1.2 Thesis Overview

For this thesis, I have implemented the first mode analyzer using a static-graph reduction technique [11] following the abstract mode analysis method defined by Ueda and Morita [14]. I have shown that it is possible to implement a mode analyzer that has a reasonable analysis time. Also, I have conducted a performance evaluation of the mode analyzer and another mode analyzer which uses an active-graph reduction technique [10]. The results show that the analysis time is within a factor of four compared to the compilation time for the benchmarks analyzed. Memory consumption is more problematic; however I believe that improvements to the first prototype system will alleviate such inefficiencies.

1.3 Thesis Outline

Chapter 1 presents the purpose of the thesis with a brief explanation of the execution model of an FGHC program. Chapter 2 describes static-graph reduction mode analysis. Chapter 3 briefly demonstates the correctness of the analyzer by showing that the analyzer finds both correct and complete mode paths of moded programs, in addition to detecting mode conflicts in non-moded programs. In Chapter 4, Performance evaluation of the analyzer is compared to another mode analyzer that uses the active-graph reduction technique. We close the thesis with conclusions and suggestions for future work in Chapter 5.

2 Mode Analyzer Algorithm

In recent years, numerous mode analyzers have been implemented. Each of these mode analyzers uses a different approach in finding the mode path of a program. Examples of mode analysis techniques are:

- static-graph reduction mode analysis [11]
- active-graph process network reduction mode analysis [10]
- finite-domain mode analysis [4]
- model generation theorem prover mode analysis [10]

In this chapter, we introduce necessary definitions used in the thesis, and we introduce a mode-analysis algorithm using the static-graph reduction technique. First, we describe the algorithm step-by-step with small example codes. Then, we show how the algorithm works with the Quicksort benchmark program.

2.1 Definitions

Ueda and Morita's notion of "path" [14] 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 $\{\langle f, i\rangle, \langle a, j\rangle, \langle b, k\rangle, \ldots\}$ (where $\langle f, i\rangle$ 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 [13]. We use the Quicksort program as the example program in describing the definitions. The program is shown in Figure 1.

> *Definition* [14]: 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, $\langle q/3, 1\rangle\langle./2, 2\rangle\langle./2, 1\rangle \in P_{Atom}$ $\langle./2, 2\rangle\langle./2, 1\rangle \in P_{Term}$. □

> *Definition* [14]: 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 the Quicksort example, the *cadr* of the first argument of procedure $q/3$ has an input mode specified as: $m(\{\langle q/3, 1\rangle, \langle./2, 2\rangle, \langle./2, 1\rangle\}) =$ *in*. □

```
q(T0,Y0,Z0) :- T0 = [] :                                    (1)
    Y0 = Z0 |
    true.
q(T1,Y1,Z1) :- T1 = [X1|Xs1] :                              (2)
    U1 = [X1|Vs1] |
    s(Xs1,X1,L1,G1),
    q(L1,Y1,U1),
    q(G1,Vs1,Z1).

s(T2,_,L2,G2) :- T2 = [] :                                  (3)
    L2 = [], G2 = [] |
    true.
s(T3,Y3,L3,G3) :- T3 = [X3|Xs3], X3 < Y3 :                  (4)
    G3 = [X3|Ws3] |
    s(Xs3,Y3,L3,Ws3).
s(T4,Y4,L4,G4) :- T4 = [X4|Xs4], X4 >= Y4 :                 (5)
    L4 = [X4|Ws4] |
    s(Xs4,Y4,Ws4,G4).
```

Figure 1: Quicksort Program

Definition [14]: 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 $p = \langle q/3,1\rangle\langle./2,2\rangle$, m/p represents a function from a set of the paths from the *cdr* of the first argument of procedure $q/3$ to $\{in,out\}$. □

Definition [14]: A program is "moded" if the modes of all possible paths in the program are consistent, where each path may have one of two modes. □

Definition: A program is "non-moded" if the program is not a moded program. □

Definition: A mode vector matrix is a one-column matrix. Each element of the matrix is a possible mode vector. □

Example: Considering the first clause of the Quicksort example, there are two occurrences of the variable Z0 and the variable can be only input or output at these two positions. Thus, the mode vector matrix is $\{(in,in), (out,out)\}$. □

All analyses presented in this paper exploit the rules outlined by Ueda and Morita. Their axioms are restated in Figure 2 [9].

§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_g and $m(p_g) = in$

§2. Two arguments of a tell unification have opposite modes, for all possible p, or more formally: $m/\langle =/2, 1\rangle \neq m/\langle =/2, 2\rangle$, i.e., $\forall p\ m(\{\langle =/2, 1\rangle\} \cdot p) \neq m(\{\langle =/2, 2\rangle\} \cdot p)$

§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)

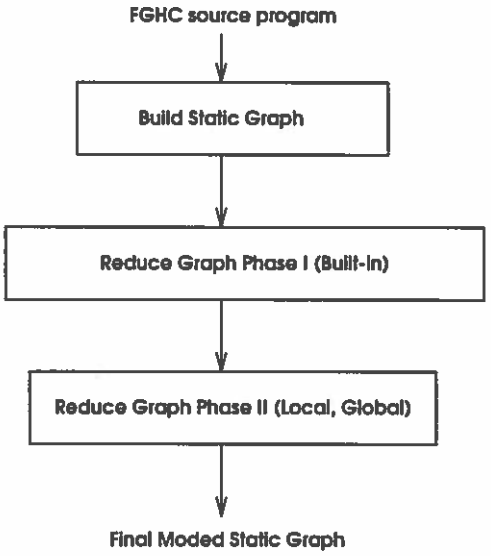


Figure 3: Structured Dataflow Graph Representation of the Sample Program

2.2 Static Graph Reduction Mode Analysis

The basic idea of static-graph reduction mode analysis is to use program graph reduction to propagate and to preserve the consistency of the mode information of the program. Mode analysis consists of three operational phases: program static graph building, system-defined functor (including unification) reducing (reducing graph part I), and user-defined functors reducing (reducing graph part II). The system dataflow is shown in Figure 3.

2.3 Static Graph Building

We introduce a static graph to represent a concurrent logic program. Each clause of a program is represented by a subgraph. The graph is a directed graph and consists of two types of nodes: variable nodes and structure (functor) nodes. A variable node is represented by a circle node, and a structure node is represented by a square node. Each child of a variable node represents a possible value binding of the variable it represents. Each child of a functor node represents an argument of the functor it represents. If a functor argument is a term (*i.e.*, not a variable), we introduce a dummy variable to substitute for that position and introduce a unification body goal to unify the argument term with the dummy variable. Notice that the program graph is a layered graph. That is, no two circle nodes nor two square nodes can be consecutive. From now on, we will use the term "variable node" and "circle node" interchangeably. For a program, we can create a collection of subgraphs, each representing a program clause. Each node of a graph is assigned a unique identification number, a node name (variable name or functor name), parent nodes' identification numbers, and child nodes' identification numbers. If the node

is a circle node, we introduce a *mode vector matrix* corresponding to the node. Each mode vector matrix consists of a number of mode vectors. Each mode vector consists of mode elements, corresponding to each parent of the node. That is, the length of the mode vector is equal to the number of parents of the associated circle node. Each mode vector represents a possible mode of the variable at each corresponding position in the source program. The possible values of a mode element are *"in"* and *"out"* (*i.e.*, input and output). Initially this mode information can be partially derived from the source program with Ueda and Morita's mode derivation axioms (Figure 2). After we create a graph and assign all the node information as described above, we create a partition set associated with the graph. Initially, each node of a graph has its own partition. That is, initially a graph with n nodes will have n partitions. Examples of program graphs are shown in Figure 4.

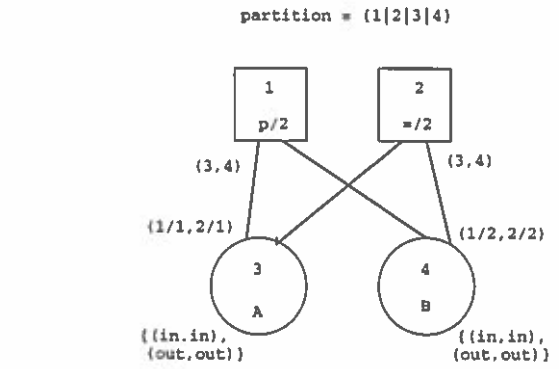
From the simple clause in Figure 4a, we create two square nodes, one for functor `p/2` and the other for functor `=/2`. Also, we number these two nodes with unique identifiers. The functor `p/2` has an arity 2. That is, its square node has two children, one for each argument. Notice that each argument of `p/2` is a variable, so we introduce a circle node for each of them. We do the same thing for the functor `=/2`. Moreover, each node contains information about its parent nodes (entry ports) and its child nodes. For example, the variable node which has the identifier 3 represents variable `A`. The variable appears in the first argument position of the functor `p/2`, which has identifier 1, and also appears in the first argument position (left-hand side) of the functor `=/2` which has identifier 2. Thus, the variable node has two parents and we keep the parents information in the form of *id/index* where *id* is parent identifier and *index* is the argument number of the parent. The parent information of variable node 3 is `(1/1,2/1)`. As mentioned, each variable node has a mode vector matrix associated with it. For variable `A`, there are two occurrences of the variable `A` in the clause. Using Ueda and Morita's rule §3.2, the initial mode vector matrix corresponding to the variable node is `{(in,in),(out,out)}`. For variable `B`, we assign its initial mode vector matrix by the same method.

In Figure 4b, we create a program graph with the same method that we just employed. The only difference is that we introduce dummy variable nodes `D1` and `D2` for each ground term (`a` and `b`), and insert dummies between the functor nodes and the ground term nodes. We can see that, as mentioned before, the program graph is a layered graph. The last step in building a program graph is to create the partition set. Each initial partition set of the graphs is shown in Figure 4.

2.4 Unification and System-Defined Function Reducing (Graph Reducing Phase I)

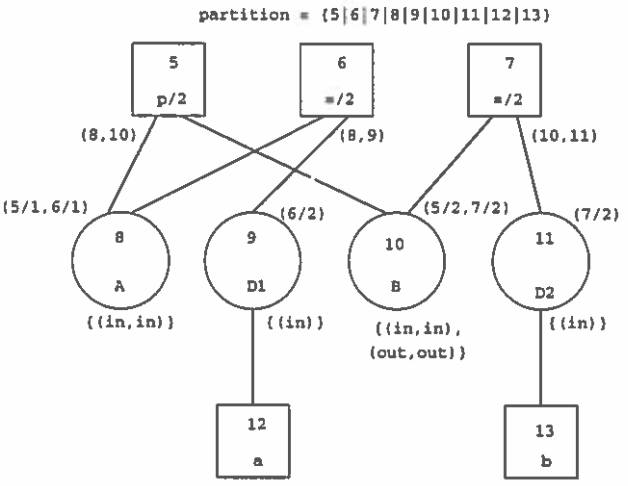
The input of this step is an initial graph which is built in the previous step. In this step, we will consider only unification (`=/2`) functor nodes and system-defined functor nodes (*i.e.*, arithmetic operators and built-in functions). We can divide these functor nodes into three categories:

- System-defined functions. Since the mode of each argument of system-defined functions is fixed, we can use this information as a constraint to reduce the size of the




```
p(A,B) :- true : A = B | true.
```

(a)




```
p(A,B) :- A = a : B = b | true.
```

(b)

Figure 4: Examples of Program Graphs

mode vector matrices corresponding to their argument nodes. However, our mode analyzer can handle only mathematic and comparison functors. We are not accounting for all system-defined functions. A proper way to handle all system-defined functors can be found in Massey [6].

- Passive unify (*i.e.*, a unification functor that appears in an ask guard). Each passive unification node is a root node and has two variable child nodes. These two child nodes can be merged (reduced) together. According to Ueda and Morita's mode rules, the modes of both arguments of a passive unify must be *in*. In order to reduce these two child nodes, a *cross-product* of their mode vector matrices is performed, and we keep only mode vectors that contain *in* mode elements at both entry-ports position from the passive unification root node. We discard the other mode vectors.

  The result of passive unify merging is a graph in which the two child nodes of the unification functor are reduced into one variable node. In our implementation, we discard the passive unification root node after the reduction and we also reduce the mode vector matrix by discarding the mode element in each vector corresponding to the unification functor parent node.

  An example of this reduction is shown in Figure 5. We reduce the `A` node with the `B` node since these two nodes are the children of a passive unification operator (root node). The cross-product of the mode vector matrices is performed. The mode vector matrices are {(in,in),(out,out)} and {(in,in),(out,out)}. The cross-product is:

$$\left\{ \begin{array}{ll} (\texttt{in}, & \texttt{in}) \\ (\texttt{out}, & \texttt{out}) \end{array} \right\} \times \left\{ \begin{array}{ll} (\texttt{in}, & \texttt{in}) \\ (\texttt{out}, & \texttt{out}) \end{array} \right\} = \left\{ \begin{array}{llll} (\texttt{in}, & \texttt{in}, & \texttt{in}, & \texttt{in}) \\ (\texttt{in}, & \texttt{in}, & \texttt{out}, & \texttt{out}) \\ (\texttt{out}, & \texttt{out}, & \texttt{in}, & \texttt{in}) \\ (\texttt{out}, & \texttt{out}, & \texttt{out}, & \texttt{out}) \end{array} \right\}$$

  We discard the second through the fourth mode vectors of the cross-product matrix since these three vectors contain mode elements that do not obey the mode rules. That is, the two arguments of the passive unification functors (2^{nd} and 4^{th} vector elements) must have mode *in*. Thus, the final result of the cross-product is {(in, in, in, in)}. Since we merge two variable nodes `A` and `B` together, each mode vector of the resulting matrix is reduced by removing mode elements which correspond to the unification node entry-port, and the final result mode is {(in, in)} as shown on the right side of Figure 5.

- Active unify (*i.e.*, a unification functor that appears in a tell guard). In this case, we perform almost the same action as in the case of passive unify. The only difference is the final result from the cross-product. According to the mode rules, the mode of the left-hand-side variable and the mode of the right-hand-side variable of the active unification operator must be opposites. An example is shown in Figure 6.

The outputs from this phase are program graphs which have no unification nor system-defined functor root nodes.

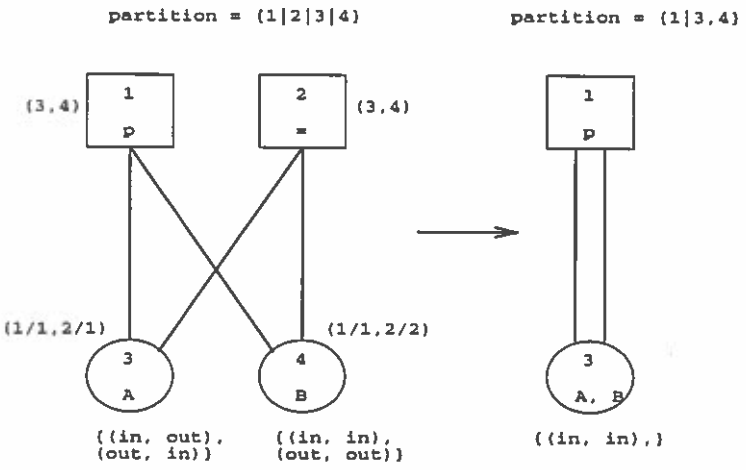


Figure 5: Passive Unify Merging

2.5 User-defined functors reducing (Graph Reducing Phase II)

In this phase, we take the graph produced by the previous phase and do more reductions until we get the *minimal graph*. The minimal graph is the program graph that has been reduced until all the root nodes are unique. That is, no two root nodes have the same functor name and arity. Assuming that the program is a moded program, intuitively, two functors that have the same name and arity must have the same data-flow. That is, the corresponding argument positions must have the same mode. For instance, consider the following program:

```
member(A,[B1|Bs],Result) :- A = B1 : Result = yes | true.
member(A,[B1|Bs],Result) :- A \= B1 : true | member(A,Bs,Result).
member(_,[],Result) :- true : Result = no | true.
```

For each occurrence of `member/3`, the first argument and its subterm must have the same mode (*in*), the second argument and its subterm must have the same mode (*in*), and the third argument and its subterm must have the same mode (*out*). This is the main concept behind a program graph reduction in this phase. We use an abstract unification algorithm to unify two functor root nodes which have the same name and arity. Figure 7 is the pseudo-code for this unification algorithm [11]. We denote a variable node (circle node) with v which has fields:

- *v.in*: The entry-port vector (identifying all parents. Each element of the vector has the form *id/index* where *id* is a parent identification number and *index* is the $index^{th}$ argument of the parent.

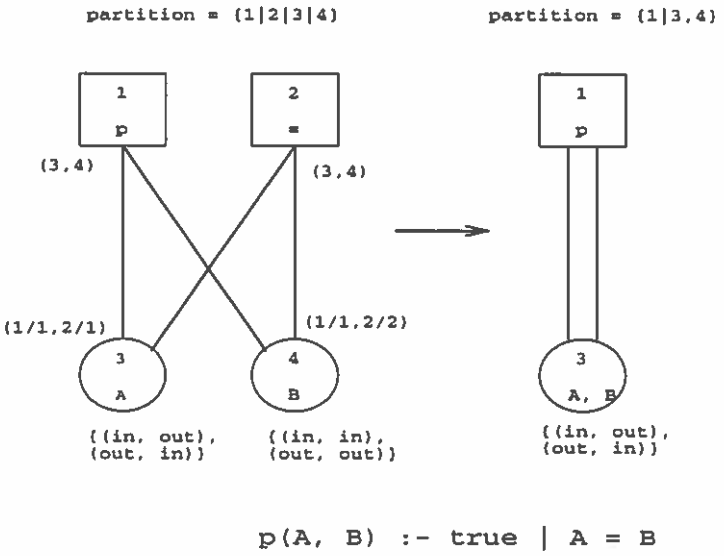


Figure 6: Active Unify Merging

- *v.out*: The exit-port vector (identifying all children), each element of the vector is the identification number of a child node.
- *v.modes*: The mode vector matrix.

We denote a functor node (a square node) with *s* which has the fields:

- *s.out*: The vector of child nodes (exit-ports). Each element of the vector is the identification number of a child node.
- *s.fun*: The form *name/arity*.

From Figure 7, unification is invoked as *unify(a,b)* where *a* and *b* are functor root nodes. The output of *unify(a,b)* is either failure or success with a new graph generated. The main idea of the algorithm is to traverse through the program graphs and find the mode information (mode vector matrix) of each pair of nodes which are supposed to have the same mode constraint. Then, we find a new mode vector matrix by multiplying these two mode vector matrices and filtering out the mode vectors that contrast with the mode constraint between the two nodes, and put the mode vector matrix in one of the two nodes. This action is called reduction. In our implementation, we will consider only the node that has the new mode constraint added for future reduction. Also notice that, in practice, the program graphs shrink after the *unify* process.

By keeping in track of the node that has a new mode constraint added (a new mode vector matrix) after a reduction, each graph has a disjoint node partition set associated with it. When two nodes are reduced, the node partition set is changed according to the

```
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 7: One-Pass Rational-Tree Unification Algorithm

reduction. The two disjoint partitions that contain these two nodes are glued together. In our implementation, the new mode vector matrix is always put in the node that has lower identifier of the two reduced nodes. Thus, in a node partition set, the node which has the lowest identifier will be the node that represents the whole node partition set when we reduce any node in the partition set with other nodes. Two important operations for the reduction are *union(x,y)*, and *find_set(x,y)*. The *union(x,y)* function unites two disjoint sets, where x belongs to the first disjoint set and y belong to the second disjoint set. The procedure *union* returns the least identifier of nodes of the united partitions. The function *find_set(x)* returns the least identifier of the disjoint set containing x. Notice that the *unify* function reduces the number of partition sets and increases the size of each set.

As mentioned earlier, a program graph might be cyclic. If we unify two cyclic graphs in a naive way, we might enter an infinite loop by unifying two nodes which we already unified. This action is clearly shown in Figure 8. Supposed we reduce node 1 with node 2, we unify node 1 with node 2, node 3 with node 4, node 5 with node 6, node 7 with node 8, node 1 with node 2, and so on. We can see that the unify procedure will continue forever and it will not give any better mode information because we are unifying the nodes which are already unified. We protect the analyzer from infinite unification by introducing a *mark table*. The mark table is simple a two-dimensional array of size $n \times n$ where n is the highest identifier in the graph. Initially, the mark table is clear. The mark table will be marked at (i, j) and (j, i) when we unify node i with node j. Notice that node unification is associative. We check the mark table every time before reducing two nodes. If the mark table has already been marked at the associated locations, unification succeeds immediately.

Now we consider how the *unify* function works. In Figure 7(1), we initially clear the mark table and we call *sunify* for unifying two root functor nodes. The function *sunify* checks whether the two functor nodes have the same name and arity. At (2), we begin the unification by marking the mark table at the position of the two unified nodes and start unifying their corresponding child nodes. Notice that all child nodes must be variable nodes.

We call *vunify* to unify each pair of child nodes. In the *vunify* function, before we unify two variable nodes, we call the *find_set* function in order to find the nodes that represent the partition sets that these two nodes are in. At (3), we mark the mark table to indicate that the two nodes have already been unified and we call *union* to combine the two partition sets together. Then, we multiply the mode vector matrices of the two nodes and filter out the mode vectors that have conflicting modes at the two parent entry-ports. In (4), the variable p' contains this mode vector matrix. Since the parents of the two unified nodes are reduced together, one entry-port from the parent that has the higher identifier can be ignored. We filter out the corresponding column from both the mode vector matrix and the entry-port vector in (5) and (6). Now we have reduced the two child variable nodes. Notice that these two child nodes could also have their own child nodes. We continue the process by unifying their child nodes together in (7) and do the recursive unification on their children in (8).

An example of *vunify* execution is shown in Figure 9. In Figure 9a, we unify node `A` with `B`. The arrows indicate the entry-ports to be merged. The portion of the algorithm that performs this merging is *vunify*. First, we find the two nodes that contain the most information about the partitions (this is the least identifier in the partition). The identifier can be found by *find_set(*`A`*)* and *find_set(*`B`*)*. Thus, we pick these two nodes for unification.

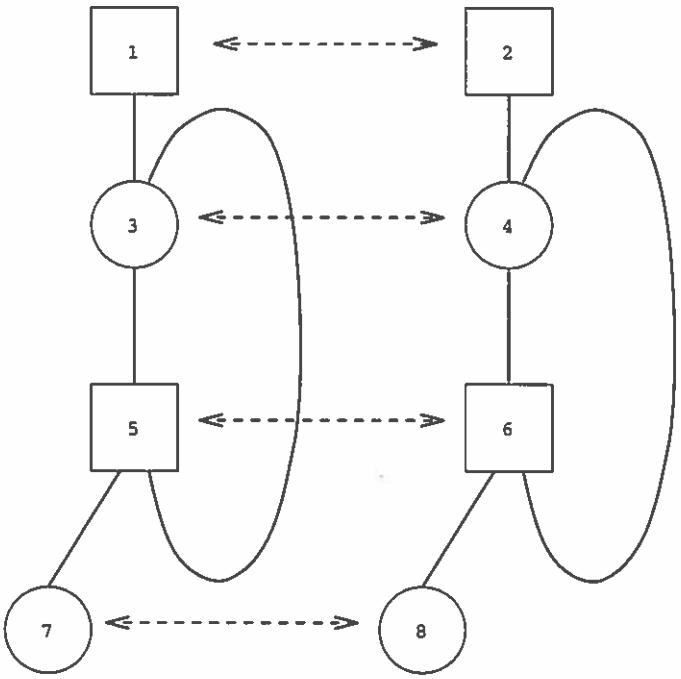


Figure 8: Unification of Cyclic Graphs

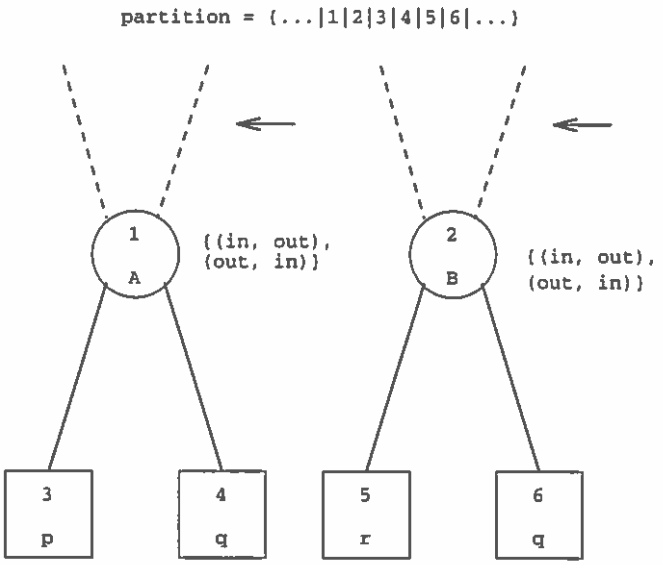
We compute the cross-product of the mode vector matrices as follows:

$$\left\{ \begin{array}{cc} (\ \texttt{in}, & \texttt{out}) \\ (\texttt{out}, & \texttt{in}) \end{array} \right\} \times \left\{ \begin{array}{cc} (\ \texttt{in}, & \texttt{out}) \\ (\texttt{out}, & \texttt{in}) \end{array} \right\} = \left\{ \begin{array}{cccc} (\ \texttt{in}, & \texttt{out}, & \texttt{in}, & \texttt{out}) \\ (\ \texttt{in}, & \texttt{out}, & \texttt{out}, & \texttt{in}) \\ (\texttt{out}, & \texttt{in}, & \texttt{in}, & \texttt{out}) \\ (\texttt{out}, & \texttt{in}, & \texttt{out}, & \texttt{in}) \end{array} \right\}$$

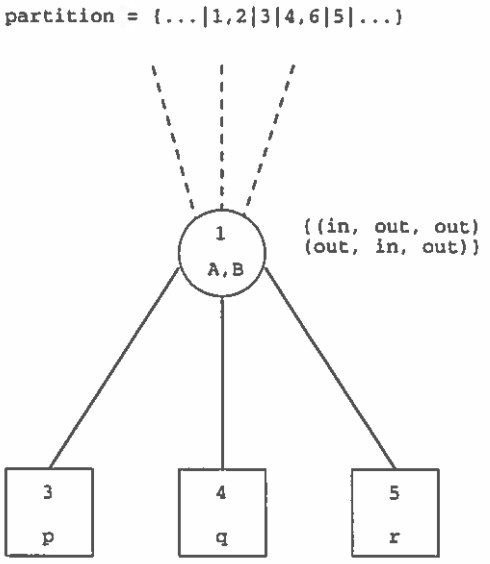
We take the product of the matrices and filter out the first and the third rows since the modes at the second and the fourth column cause a conflict. Also, we filter out the fourth column because we are reducing the graph and that entry-port has already been combined with the second entry-port. The final product is shown in Figure 9b.

In our mode analyzer implementation, there are two steps in unifying root nodes of a graph: *local* and *global* unification. In local unification, for each program clause, we call the *unify* function on matching functor root nodes in the clause. After we finish local unification, each procedure graph consists of unique functor root nodes (*minimal*) . In global unification, we take the output from local unification and follow the same procedure as in local unification, but this time we match functor root nodes across procedure subgraphs. Finally, at the end of global unification, we will get the minimal program graph with unique functor root nodes.

In general, the local and global unification described above have the same functionality. Furthermore, they actually are the same abstract unification procedure previously



(a)



(b)

Figure 9: Demonstration of *vunify* Procedure

state of A	state of B	unify(A,B)	comment
normal	*normal*	*normal*	standard unify
normal	*multiway*	*multiway*	standard unify
multiway	*normal*	*multiway*	standard unify
multiway	*multiway*	*shared*	standard unify
shared	any state	*shared*	standard unify + mode consistency check
any state	*shared*	*shared*	standard unify + mode consistency check

Table 1: Unification Variable Node with Status

memtioned. We will discuss about the complexity of the unification algorithm in Chapter 4.

2.6 Mode Consistency Check

The program graph generated by the mode analyzer is the minimal program graph and contains the final mode information. However, there are degenerate programs that give incorrect modes under this algorithm (Figure 7) because they lose the constraint of rule §4 in Figure 2. These programs will be examined closely in the next chapter. For now, we will describe a way to fix this problem.

To ensure that constraint §4 is enforced for a graph reduction, we divide variable nodes into three states: *normal*, *multiway*, and *shared* [10]. When we create a program graph, we initiate a variable node to be *multiway* if the node is moded by the rule §4; otherwise, we initiate the node to be *normal*. If during graph reduction a *multiway* node is unified with a *multiway* node, 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. The summary of node state changes is shown in Table 1.

With the mode consistency check [9], the unification of the variable nodes themselves is performed normally. However, the recursive descent is modified. If all mode vectors within the merged variable are *in* on the unification path, then this particular variable occurrence is a consumer in all possible scenarios. In fact, all variables within terms bound to that variable must be consumers also to avoid contradictions.

During the unification's descent from the merged node, recursive unifications of child pairs proceed normally. However, *singleton* children, for which unification does not take place, are no longer inherited. These children are not safe: they may erroneously be producers. Thus we descend to each such child, removing all mode vectors with *out* along the path. This removes all contradictory scenarios, potentially resulting in an empty mode for some nodes.

With mode consistency checking, the final graph is the minimal program graph and

contains correct mode information for all program paths. Examples of mode analysis with this mode consistency check are shown in Chapter 3.

2.7 Interpretation of the Final Graph

Ueda and Morita's *polarity rule* [14] suggests a way to interpret the final graph properly. If the path from a root passes through an odd number of *out* mode nodes, the final mode of the path must be interpreted oppositely from the final mode of the path in the graph; otherwise, the final mode path is interpreted normally. We illustrate the situation by showing a small program, its final graph, and its interpretation. The small example and its initial graph are shown in Figure 10. We follow the processes previously described. First, we do the unification with respect to the passive unification root node, and the graph is reduced to the one in Figure 11a. Then we do the unification with respect to the active unification root node, and the graph is further reduced to the one in Figure 11b. We can see that the graph is already in the minimal state since the graph contains only unique functor root nodes. We can interpret all the paths of the graph as in Figure 12. Notice that the paths from `=/2` functor root nodes are not shown in the mode paths

Considering all mode paths in Figure 12, the polarity rule is used when we want to find the mode of the path $\{\langle p/2,2\rangle\langle./2,1\rangle\}$ and the path $\{\langle p/2,2\rangle\langle./2,2\rangle\}$. The paths are reported to be *out*, whereas the graph shows both mode paths to be *in*, because both paths passing through one (odd number) *out* mode which is the path $\langle p/2,2\rangle$.

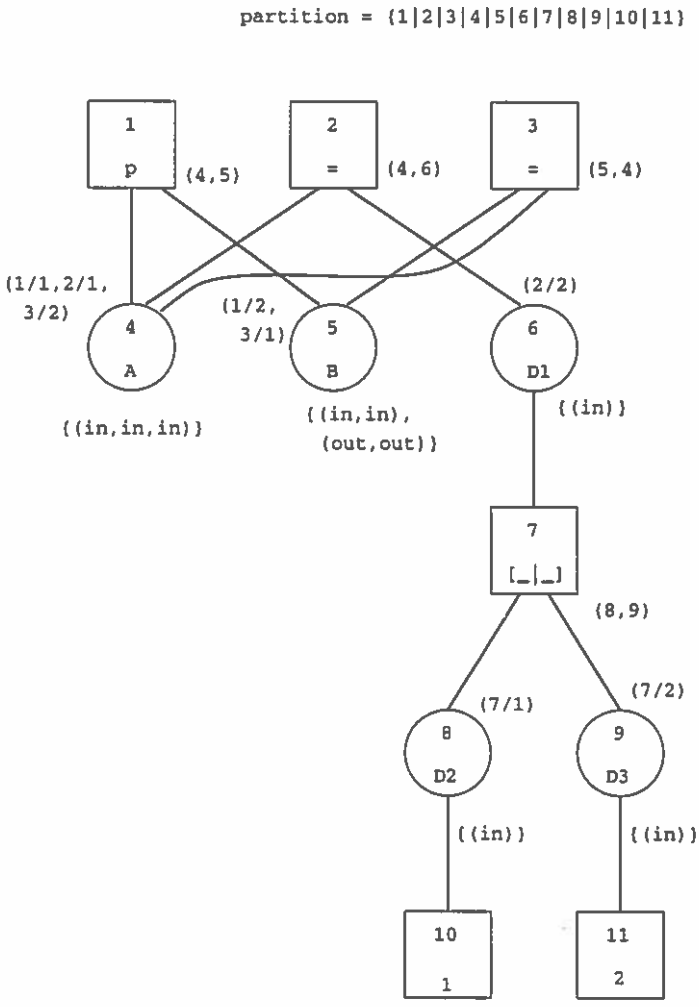
2.8 Replaying the Whole Process

We replay the whole process of mode analysis by illustrating the analysis of the quicksort program. The quicksort program is shown in Figure 13. The graph of clause (1) in Figure 13 after the ask unification root reduction is shown in Figure 14. Then we do tell unification reduction, and the resulting graph is shown in Figure 15. We cannot do any more reductions from Figure 15 because it is already the minimal graph.

Consider the second clause of the program. The graph of the clause after all unification functors are reduced is shown in Figure 16. The next phase reduces user-defined functors. We show the graph after the first user-defined functor reduction (root nodes number 1 and 3) in Figure 17. Then we show the minimal graph of the second clause in Figure 18 (after one more reduction step of nodes number 1 and 4).

We now have two minimal graphs for the first and second clauses of the program. We now do the global reduction of these two graphs and get the graph shown in Figure 19. Next, we consider the third clause of the program. The graph after reducing all unification functors is shown in Figure 20. Moreover, the graph is a minimal graph. Consider the fourth clause of the program. The initial graph is shown in Figure 21. The graph after all unification functors are reduced is shown in Figure 22. The minimal graph of the clause is shown in Figure 23. Consider the fifth clause of the program: the initial graph is shown in Figure 24. The graph after all unification functors are reduced, is shown in Figure 25. The minimal graph of the clause is shown in Figure 26.

Now, we consider reduction between clauses (4) and (5). The reduced graph is shown in Figure 27. The reduction among clauses (3), (4), and (5) is shown in Figure 28. Finally,




```
P(A,B) :- A = [1 | 2] : B = A | true.
```

Figure 10: Graph Interpretation Example (Step 1)

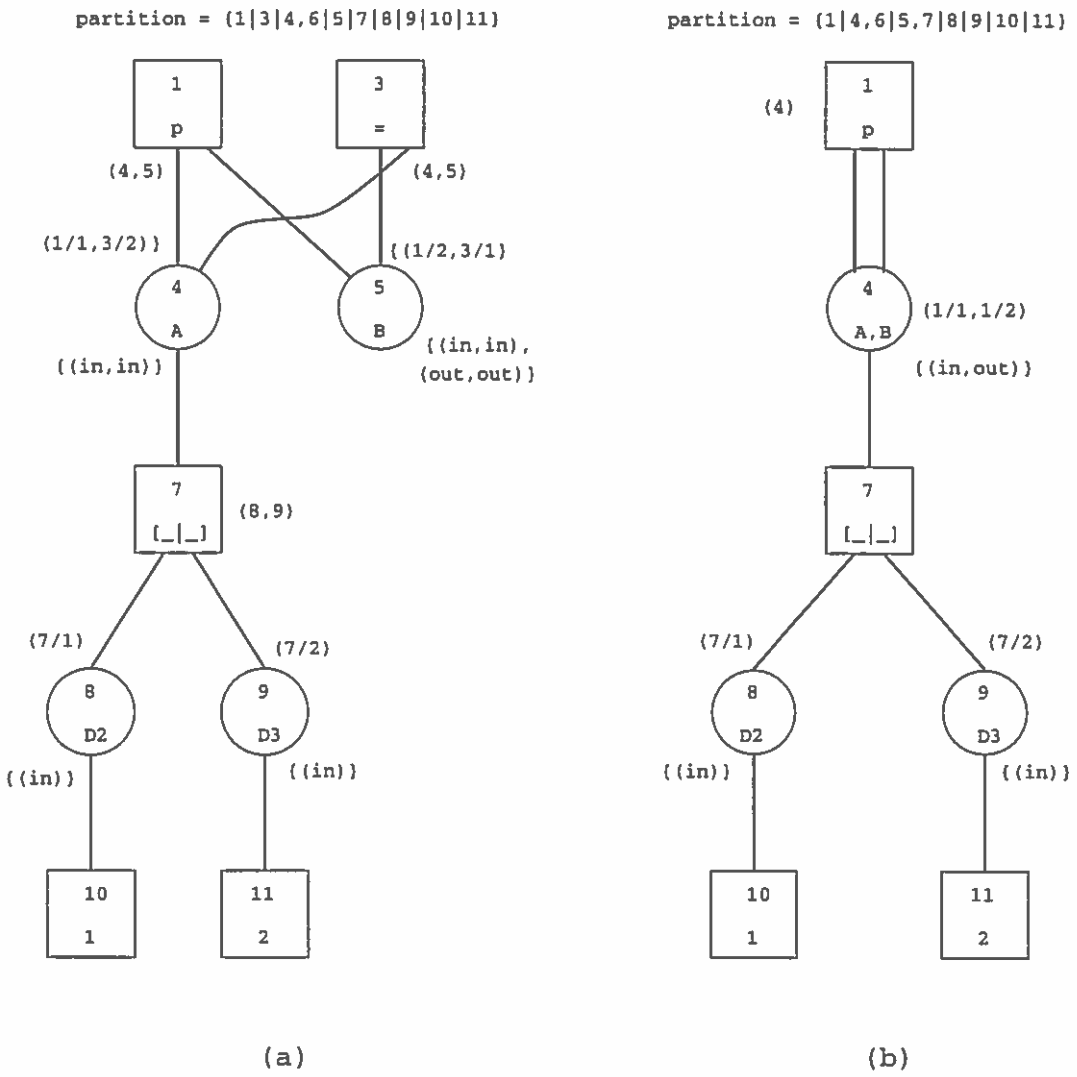


Figure 11: Graph Interpretation Example (Step 2)

$$
\begin{aligned}
&m(\{\langle p/2,1\rangle\langle./2,1\rangle\}) = in\\
&m(\{\langle p/2,1\rangle\langle./2,2\rangle\}) = in\\
&m(\{\langle p/2,1\rangle\}) = in\\
&m(\{\langle p/2,2\rangle\langle./2,1\rangle\}) = out\\
&m(\{\langle p/2,2\rangle\langle./2,2\rangle\}) = out\\
&m(\{\langle p/2,2\rangle\}) = out
\end{aligned}
$$

Figure 12: Path Modes of Example Program

```
q(T0,Y0,Z0) :- T0 = [] :                                    (1)
    Y0 = Z0 |
    true.
q(T1,Y1,Z1) :- T1 = [X1|Xs1] :                              (2)
    U1 = [X1|Vs1] |
    s(Xs1,X1,L1,G1),
    q(L1,Y1,U1),
    q(G1,Vs1,Z1).

s(T2,_,L2,G2) :- T2 = [] :                                  (3)
    L2 = [], G2 = [] |
    true.
s(T3,Y3,L3,G3) :- T3 = [X3|Xs3], X3 < Y3 :                  (4)
    G3 = [X3|Ws3] |
    s(Xs3,Y3,L3,Ws3).
s(T4,Y4,L4,G4) :- T4 = [X4|Xs4], X4 >= Y4 :                 (5)
    L4 = [X4|Ws4] |
    s(Xs4,Y4,Ws4,G4).
```

Figure 13: Quicksort Program

the reduction of the whole program (reduction of the graph in Figure 19 and the graph in Figure 28) is shown in Figure 29.

From the minimal final graph (Figure 29), we can see that the graph has no unification functors (=/2) or any system-defined functors because we reduced these functors in phase I. Actually, our implementation doesn't throw away this information. All the root functor nodes are still floating in the graph but the links to their children might not be correct compared to the minimal graph. We can add a module to our implementation that will fix all the links by making use of the disjoint partition set and the *find_set(x,y)* function. Then we can see the final graph which contains all unification functor root nodes and system-defined functor root nodes. In this thesis, we will consider only the final graph without this special module.

Considering the minimal final graph (Figure 29) again, we can see that the number of paths is infinite since we have two cycles in the graph. Notice that the cycle appears only at the list functor (node 7 and node 8). In order to display the mode paths of the graph, we break the cycles and display the mode down to only the level of the *car* and *cdr* because the mode of the *car* of a list is the same as the mode of the *cadr*, *caddr*, etc. By breaking the path, we can display a finite number of mode paths. The broken mode paths of the graph are shown in Figure 30.

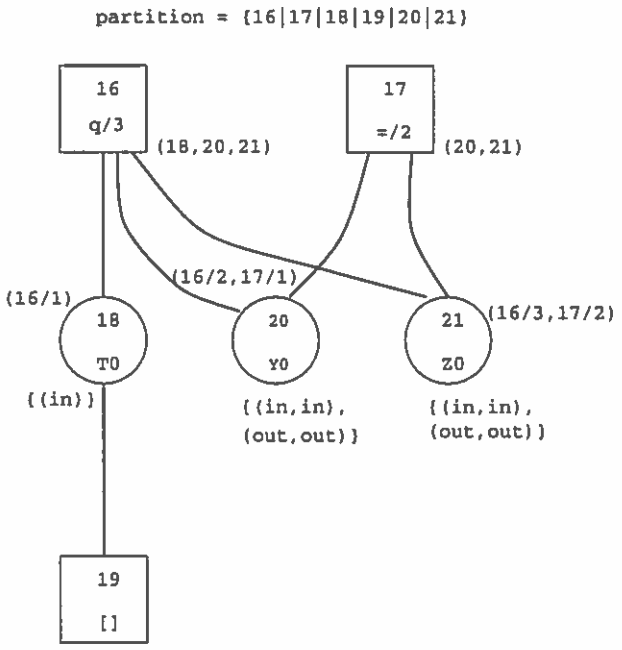


Figure 14: Quicksort Program Clause 1 Graph (Ask Unification Reduced)

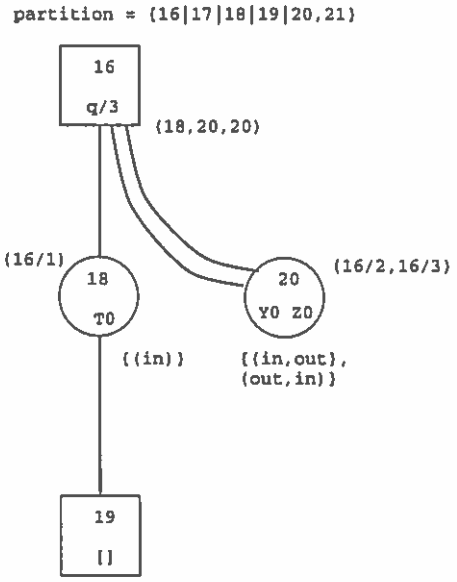


Figure 15: Quicksort Program Clause 1 Graph (Minimal)

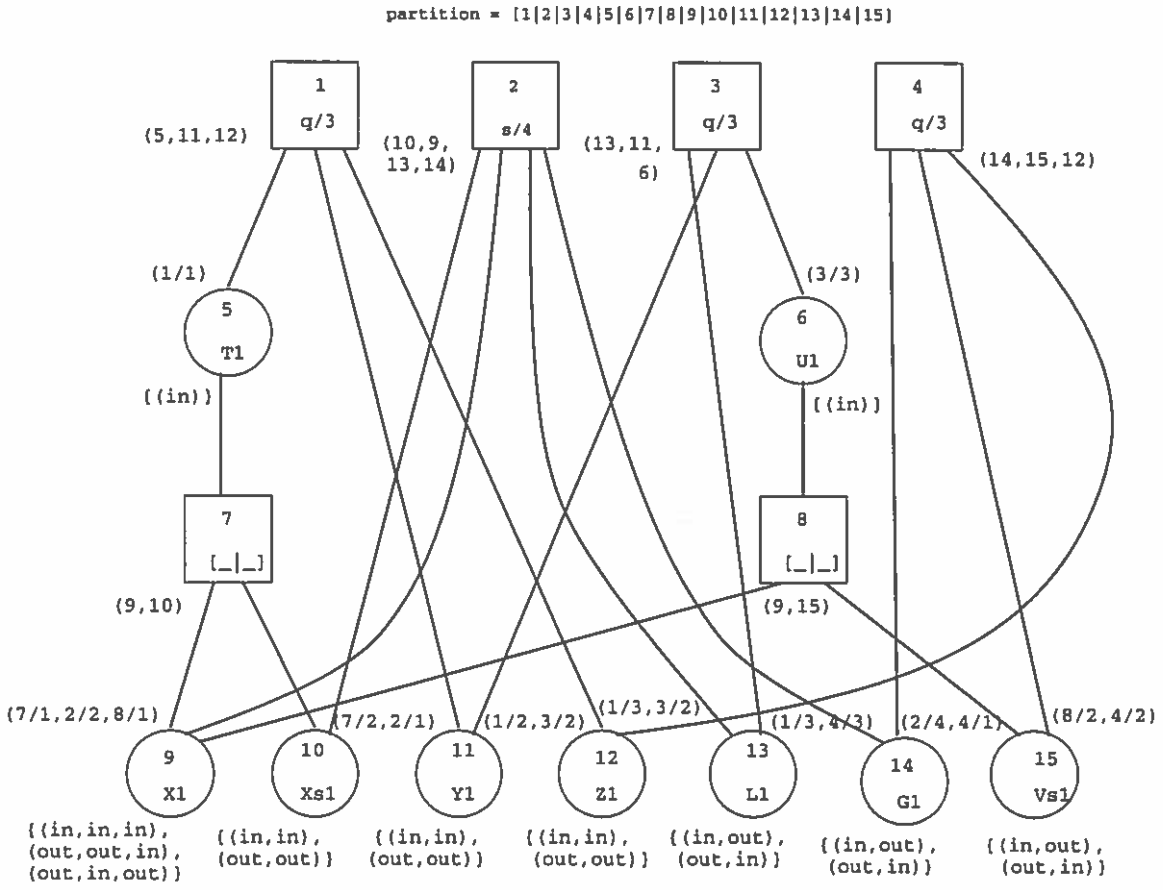


Figure 16: **Quicksort** Program Clause 2 Graph (Unifications Reduced)

3 Non-moded Programs

In the previous chapter, we demonstrated how to derive modes with our algorithm. For fully-moded programs, our mode analyzer will derive the correct mode paths of the program. This chapter is devoted to non-moded programs. We will argue that if a program is non-moded, our system will detect at least one mode path that has a mode conflict. First, we show the mechanism through which mode consistency checking detects the mode conflict in the program. Then, we show a simple non-moded program in which our analyzer detects the mode conflict without mode consistency checking (mentioned in the previous chapter). Finally, we show another non-moded program for which our mode analyzer detects the mode conflict with mode consistency checking.

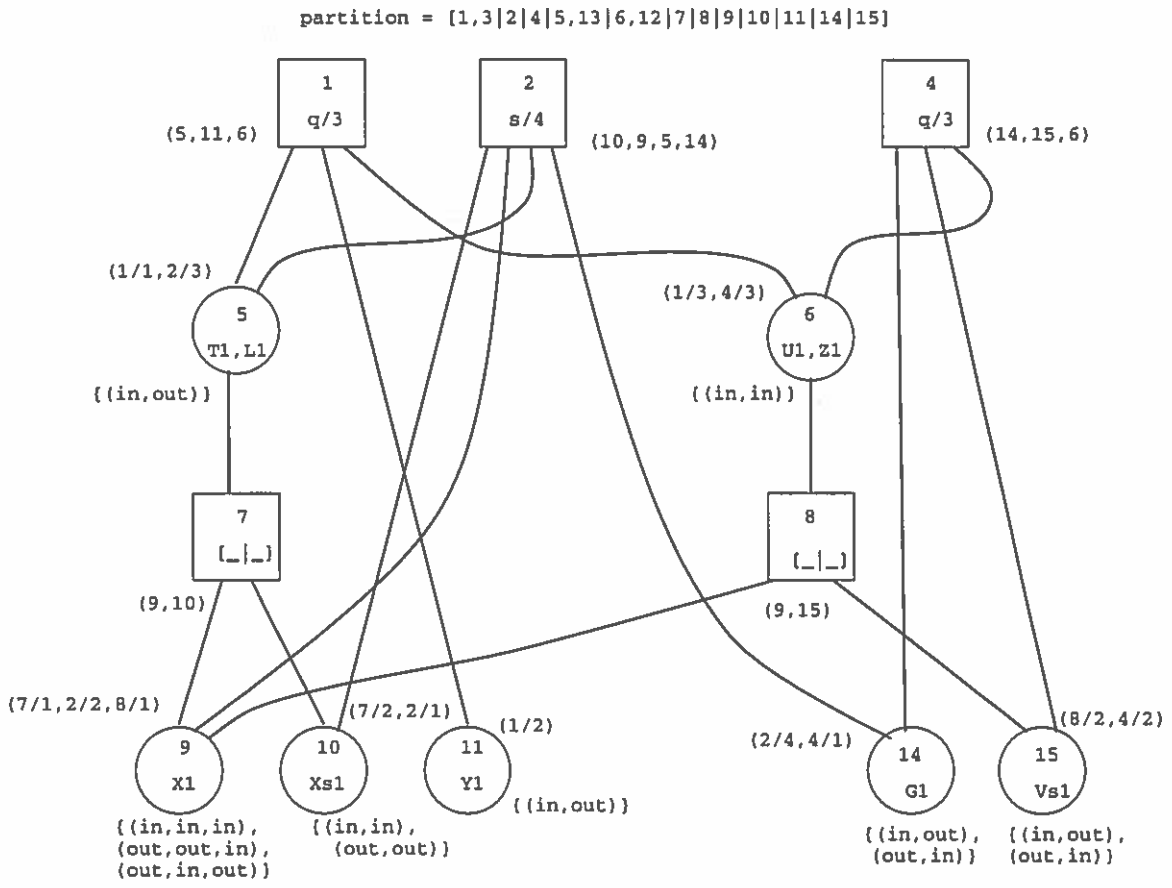


Figure 17: Quicksort Program Clause 2 Graph (After Reducing Nodes 1 and 3)

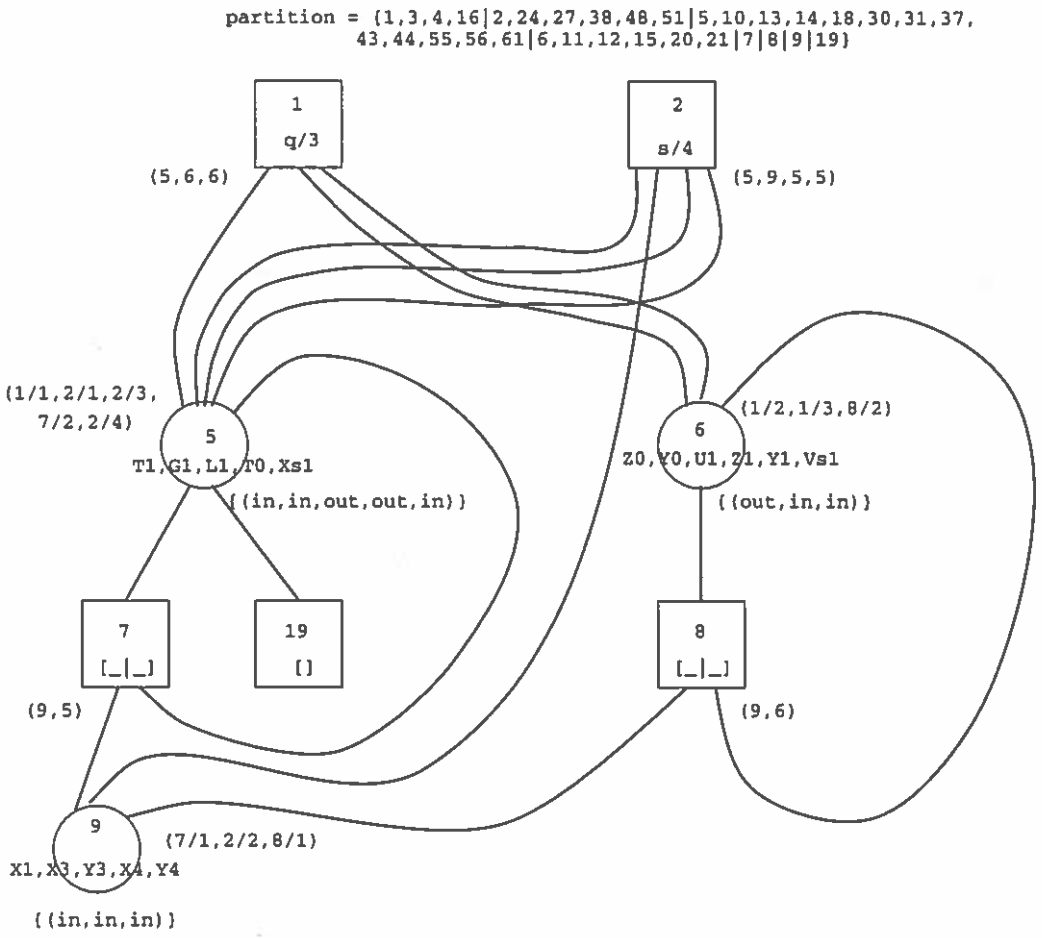


Figure 18: Quicksort Program Clause 2 Graph (Minimal)

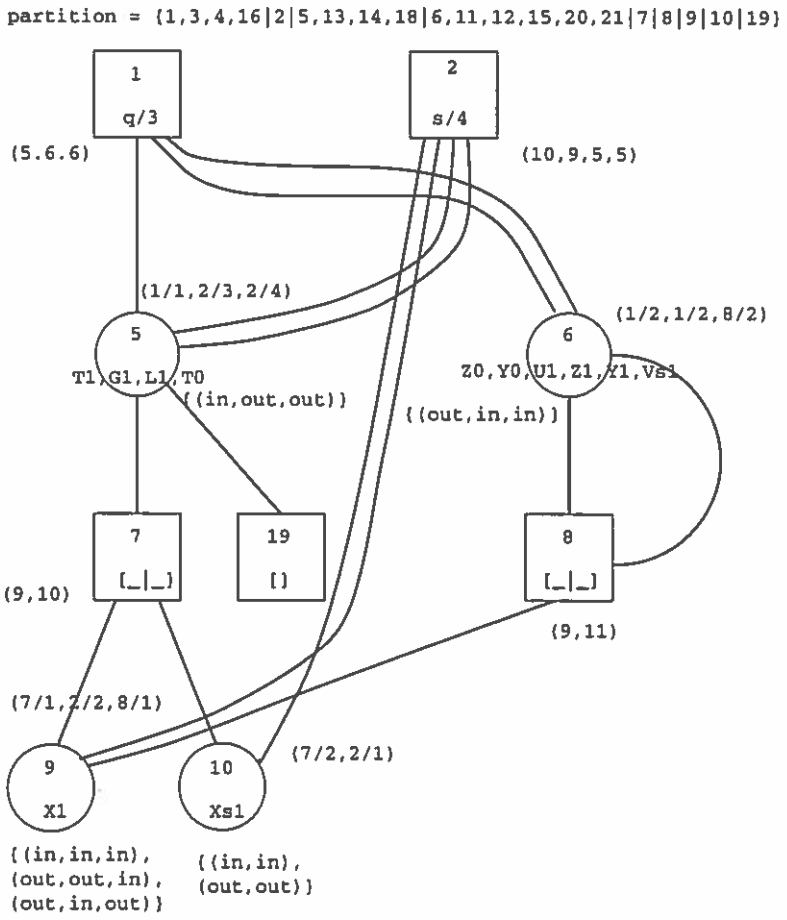


Figure 19: Quicksort Global Unification of Clause 1 & 2 Graph

3.1 Mode Consistency Checking in General

As mentioned in Chapter 2, the mode analyzer generates the complete and correct mode paths of a moded program. However, there are degenerate programs that give incorrect modes under the algorithm. These programs are a small subset of all non-moded programs. The problem occurs in programs that contain shared variables, according to mode rule §4 (refer to Figure 2). The algorithm might lose the constraint of the rule in some unification scenarios. To ensure that mode rule §4 is still enforced for the process of graph reduction, we introduce a mechanism that checks for the consistency of the rule.

Mode rule §4 is applied to shared variables. The problem occurs especially in a recursive program that has a shared variable as an argument in both the head and body. The purpose of the mode consistency check mechanism is to check that if the shared variable has *in* mode in all possible scenarios, then all the paths from the variable must have *in* mode also; otherwise a mode conflict occurs. That is, if the shared variable is constrained to be a consumer in all possible scenarios, all variables within the term bound to the variable must also be consumers. Examples in the following sections will help clarify this explanation.

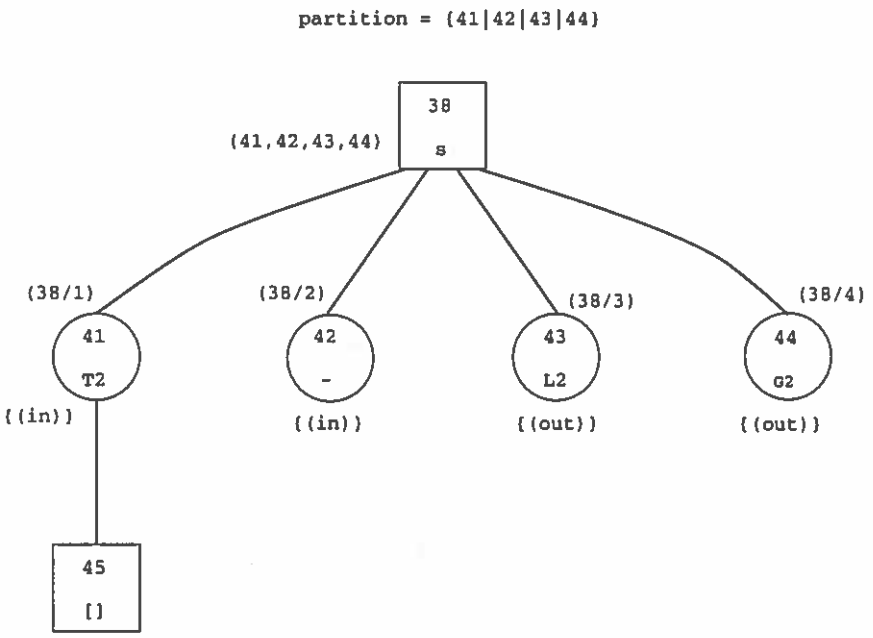


Figure 20: Quicksort Program Clause 3 Graph (Minimal)

3.2 Simple Non-moded Program

Consider a non-moded program,

```
p(A,B) :- A = 5 : B = A | true.
p(A,B) :- B = 5 : B = A | true.
```

We can see that the program is non-moded since the path $m(\{\langle p/1,1\rangle\})$ from the first clause has mode *in*, whereas the same path from the second clause has mode *out*. Moreover, the path $m(\{\langle p/1,2\rangle\})$ from the first clause has mode *out*, whereas the same path from the second clause has mode *in*.

Using our mode analyzer, Figures 31(a) and 31(b) show the initial program graphs for the first and the second clauses of the program, respectively. After we do unification reduction, the graphs are shown in Figure 32. Notice that the graphs are in minimal form. We now do global reduction of these two graphs, and the final graph is shown in Figure 33. In unifying the root functor p/2 of the first and the second clauses, we can see that the result of multiplying the mode vector matrices of variable node 4 and variable node 11 is empty. This means that a mode conflict appears in the graph. Intuitively, the mode conflict happens because the first argument of functor p/2 in the first clause must be input but it is output in the second clause. A mode conflict also happens for the same reason for the second argument of p/2. From the final graph, we can indicate that the mode conflict happened at the paths $m(\{\langle p/1,1\rangle\})$ and $m(\{\langle p/1,2\rangle\})$. Thus, mode analysis without mode consistency checking can detect a mode conflict for this non-moded program. This is not always possible, however, as illustrated in the next section.

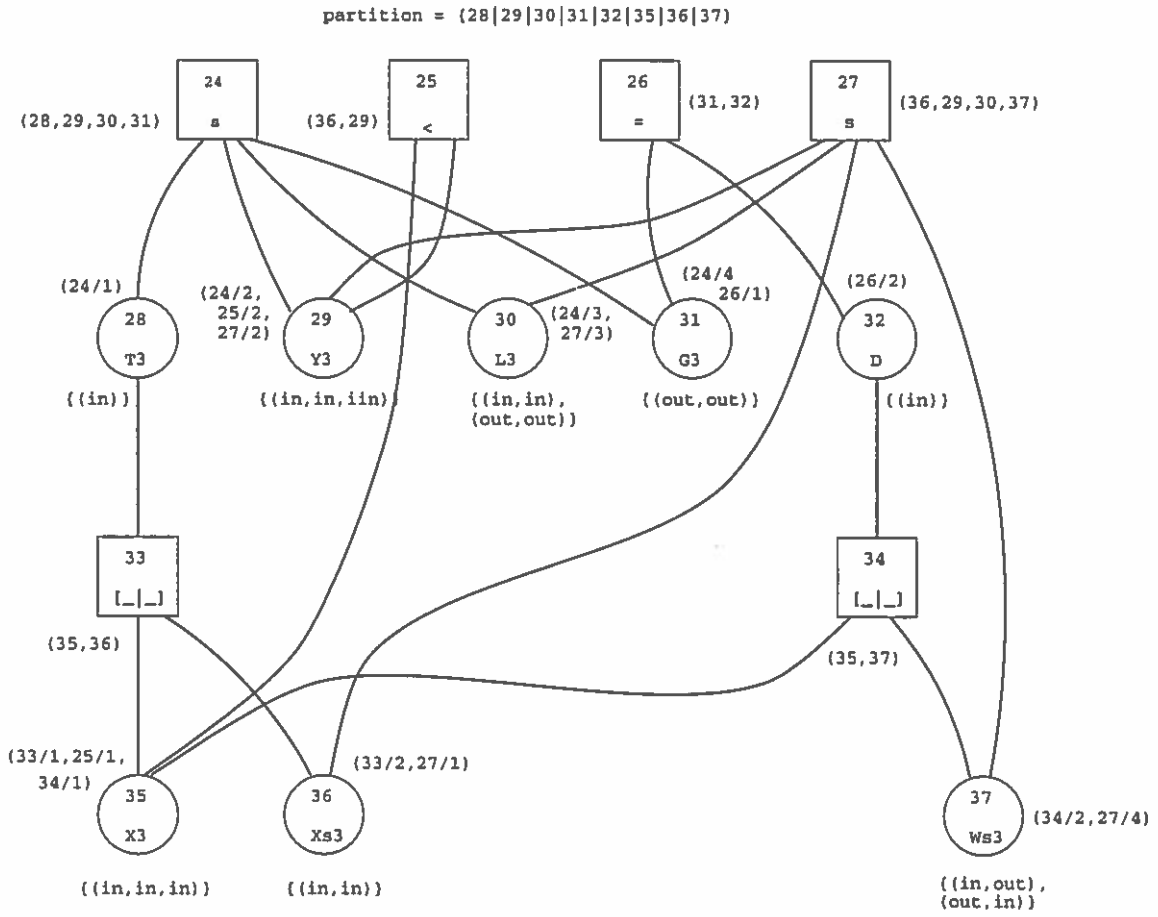


Figure 21: Quicksort Program Clause 4 (Initial Graph)

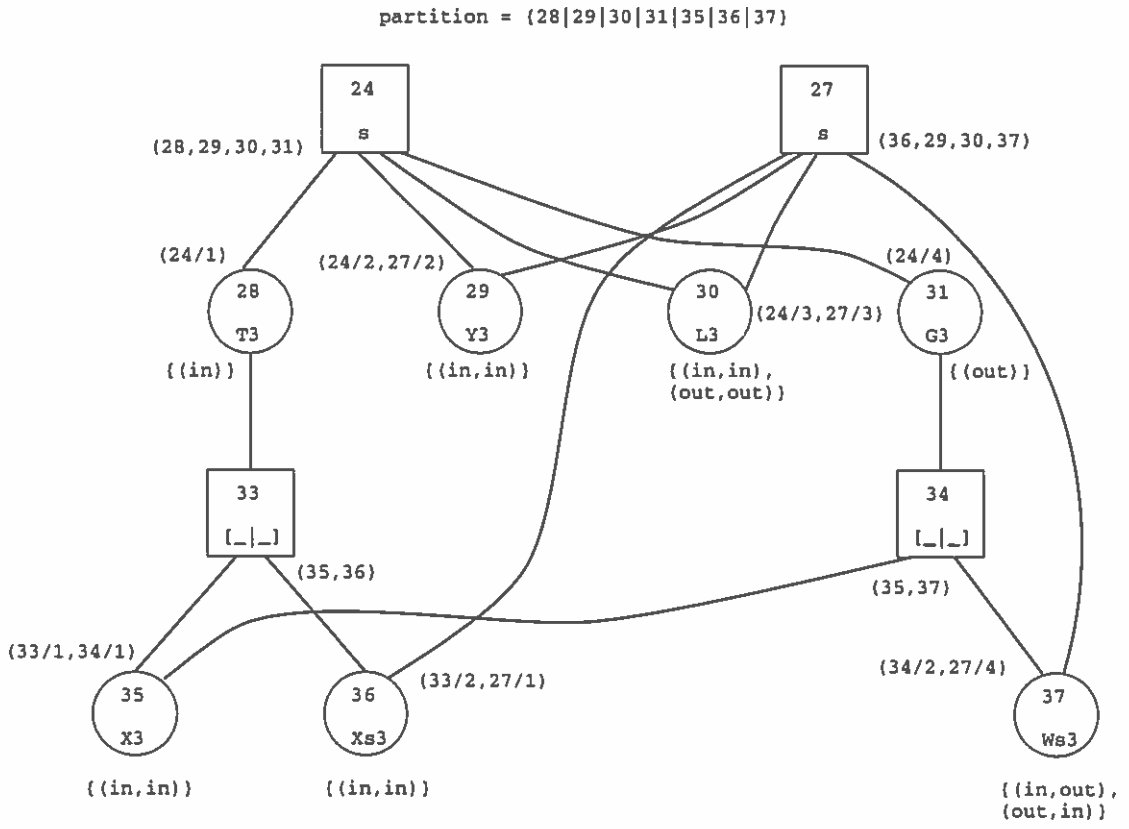


Figure 22: Quicksort Program Clause 4 Graph (All Unification Functors Reduced)

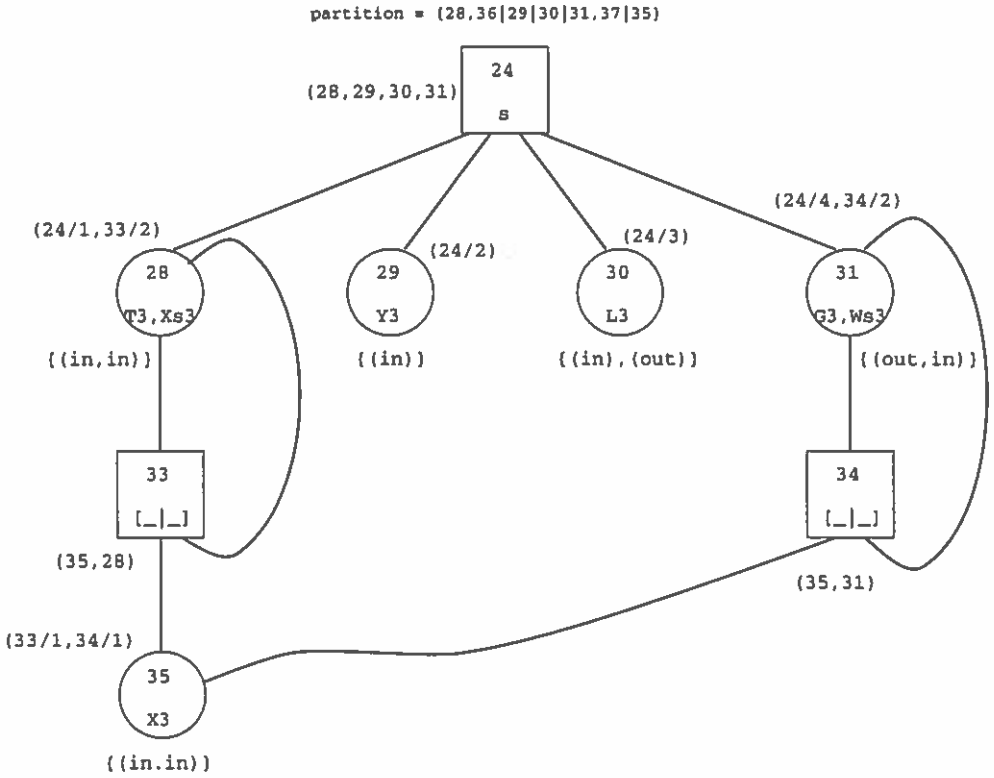


Figure 23: **Quicksort** Program Clause 4 Graph (Minimal)

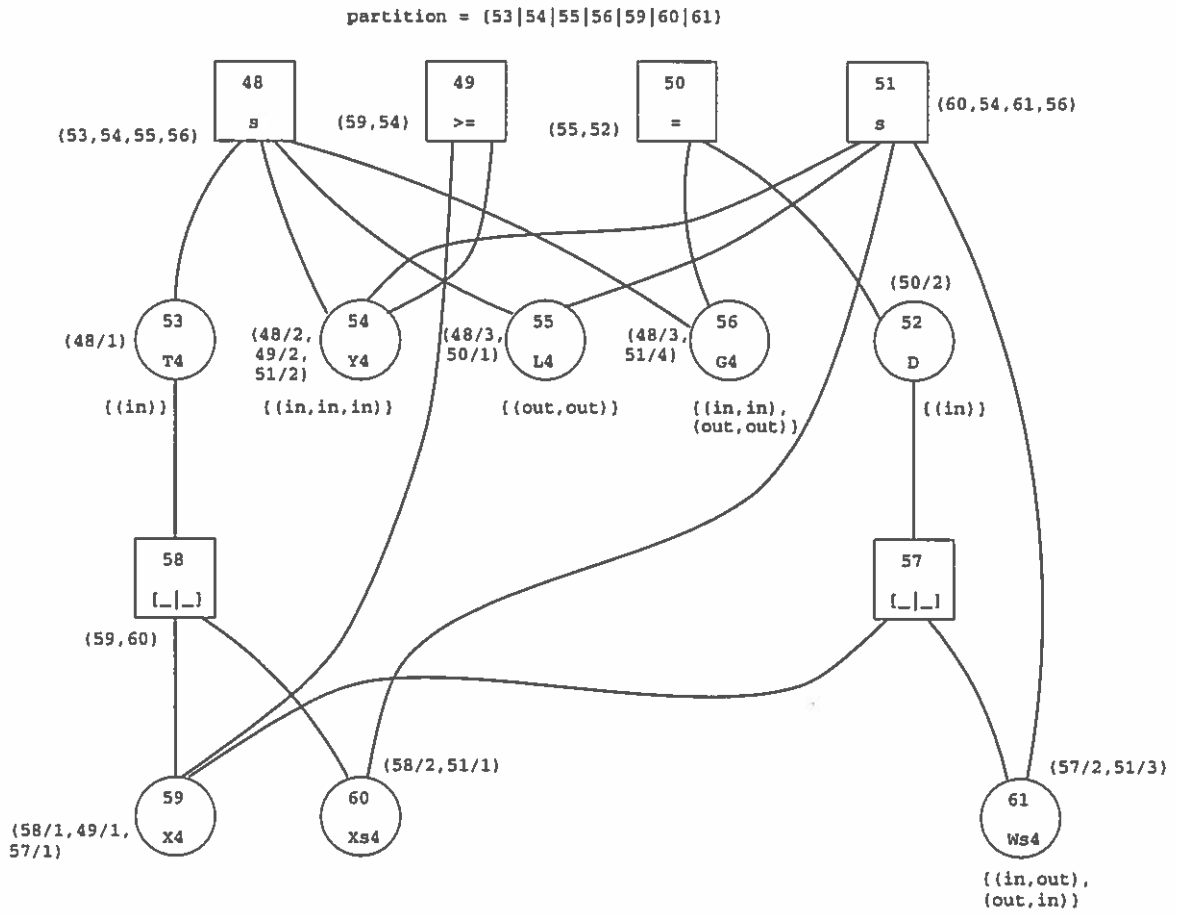


Figure 24: Quicksort Program Clause 5 Initial Graph

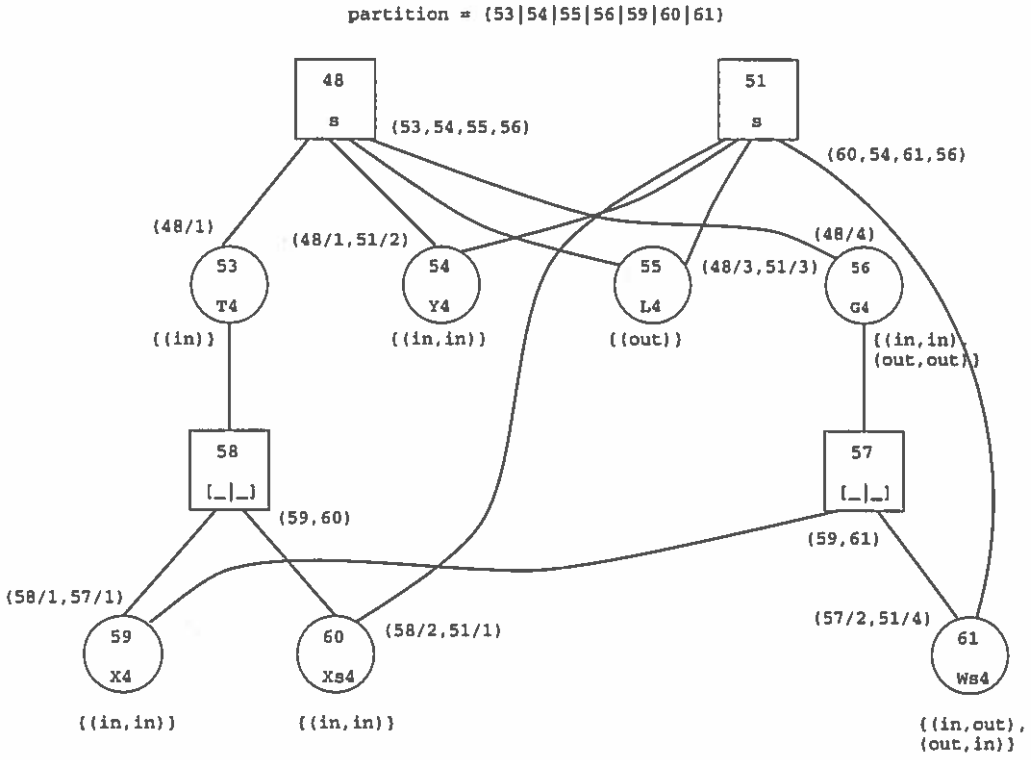


Figure 25: Quicksort Program Clause 5 Graph (All Unification Functors Reduced)

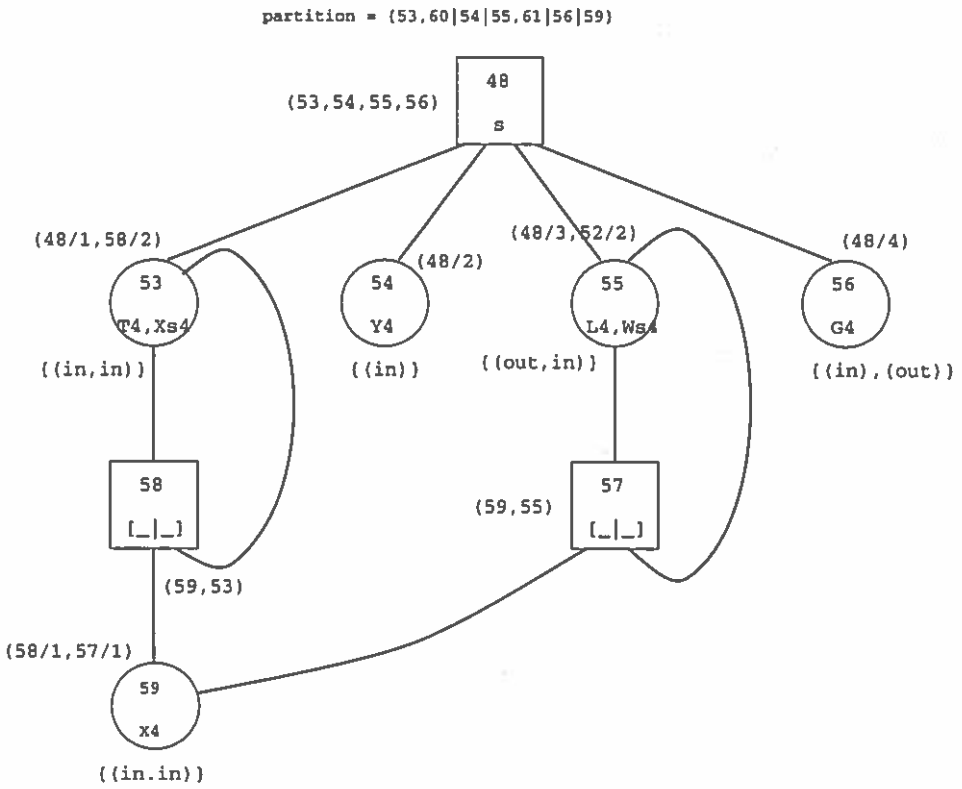


Figure 26: Quicksort Program Clause 5 Graph (Minimal)

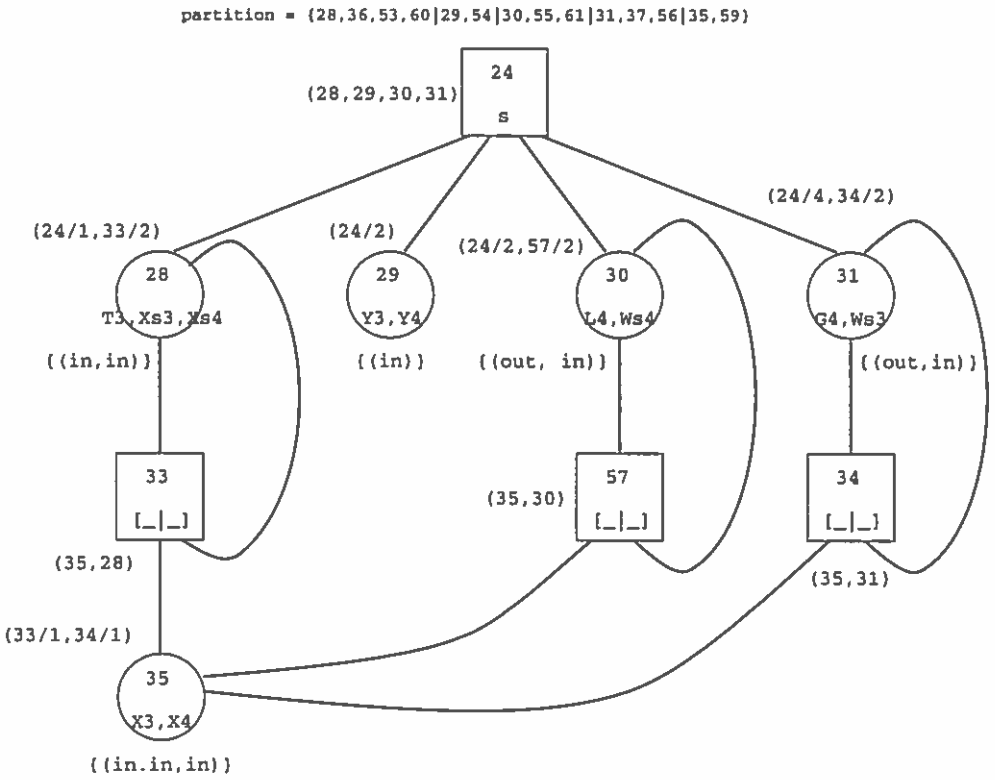


Figure 27: Quicksort Program Global Unification of Clauses 4 & 5

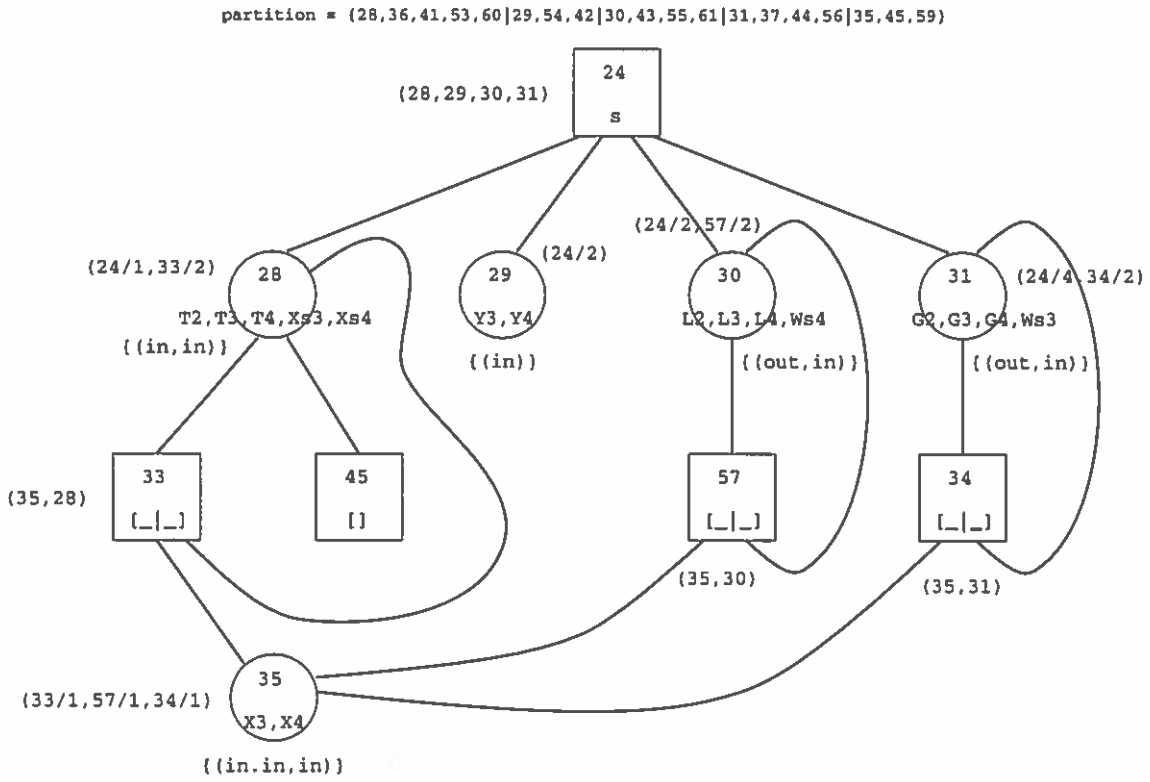


Figure 28: Quicksort Program Global Unification of Clauses 3, 4, and 5

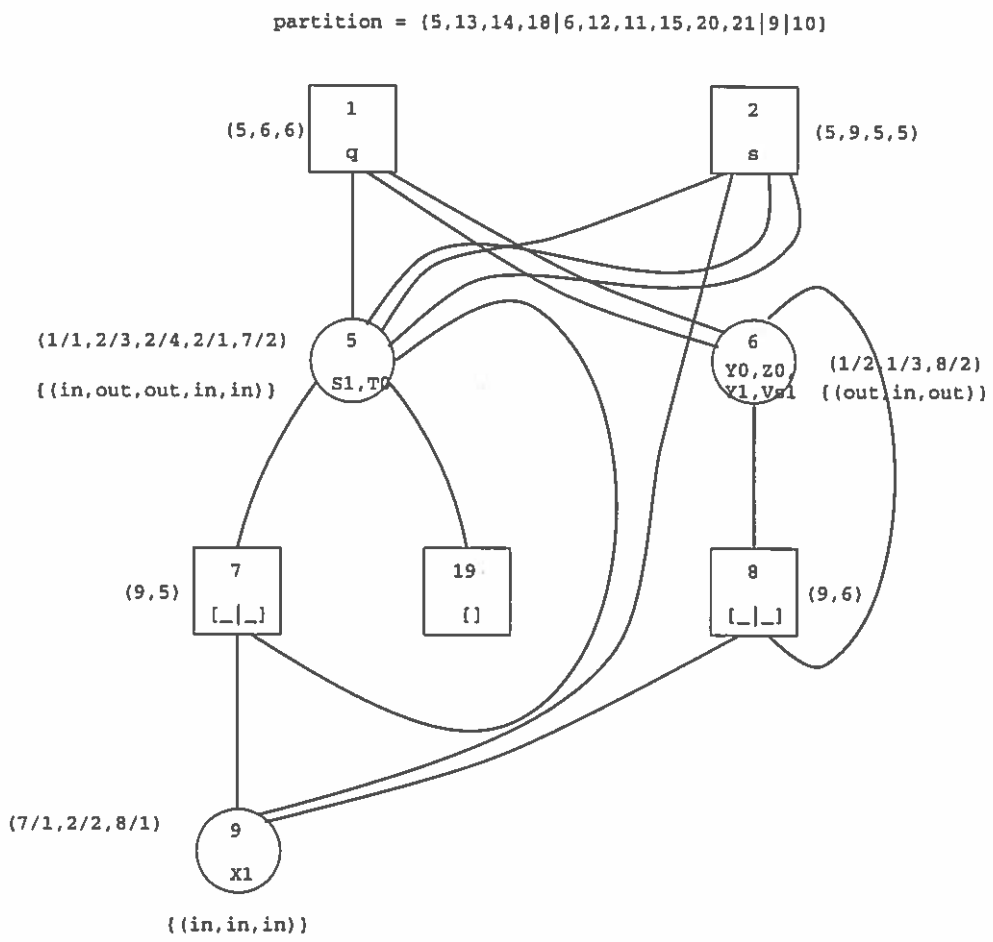
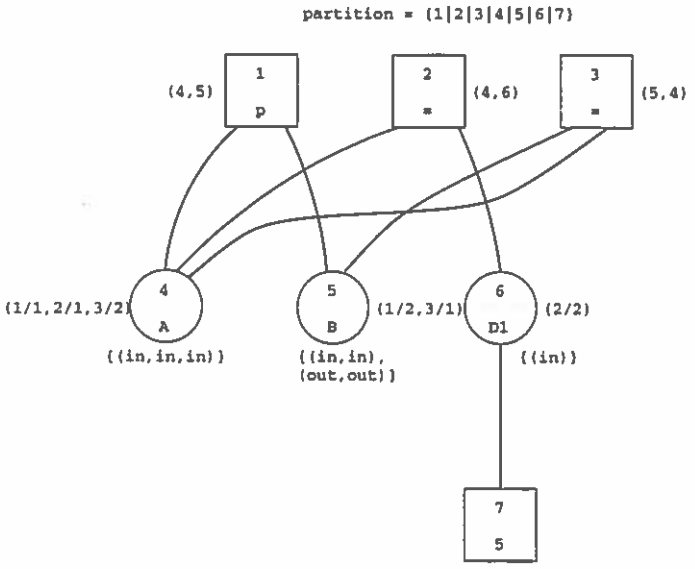


Figure 29: Quicksort Program Final Graph

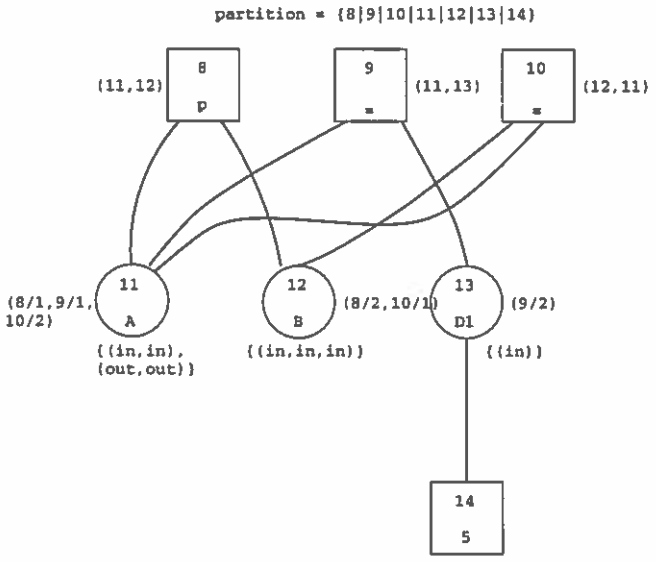
$m(\{\langle qsort/3,1\rangle\langle./2,1\rangle\}) = in$	$m(\{\langle split/4,1\rangle\langle./2,1\rangle\}) = in$
$m(\{\langle qsort/3,1\rangle\langle./2,2\rangle\}) = in$	$m(\{\langle split/4,1\rangle\langle./2,2\rangle\}) = in$
$m(\{\langle qsort/3,1\rangle\}) = in$	$m(\{\langle split/4,1\rangle\}) = in$
$m(\{\langle qsort/3,2\rangle\langle./2,1\rangle\}) = out$	$m(\{\langle split/4,2\rangle\}) = in$
$m(\{\langle qsort/3,2\rangle\langle./2,2\rangle\}) = out$	$m(\{\langle split/4,3\rangle\langle./2,1\rangle\}) = out$
$m(\{\langle qsort/3,2\rangle\}) = out$	$m(\{\langle split/4,3\rangle\langle./2,2\rangle\}) = out$
$m(\{\langle qsort/3,3\rangle\langle./2,1\rangle\}) = in$	$m(\{\langle split/4,3\rangle\}) = out$
$m(\{\langle qsort/3,3\rangle\langle./2,2\rangle\}) = in$	$m(\{\langle split/4,4\rangle\langle./2,1\rangle\}) = out$
$m(\{\langle qsort/3,3\rangle\}) = in$	$m(\{\langle split/4,4\rangle\langle./2,2\rangle\}) = out$
	$m(\{\langle split/4,4\rangle\}) = out$

Figure 30: Quicksort Program Broken Path Modes




```
p(A,B) :- A = 5 : B = A | true.
```

(a)




```
p(A,B) :- B = 5 : B = A | true.
```

(b)

Figure 31: Initial Graphs of Simple Non-moded Program

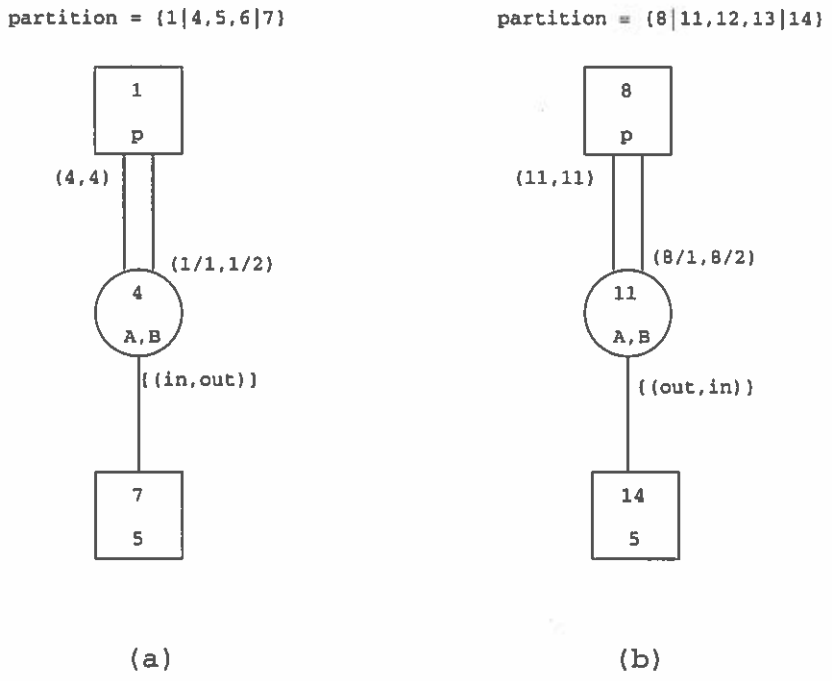


Figure 32: Graphs of Simple Non-moded Program (Unification Reduced)

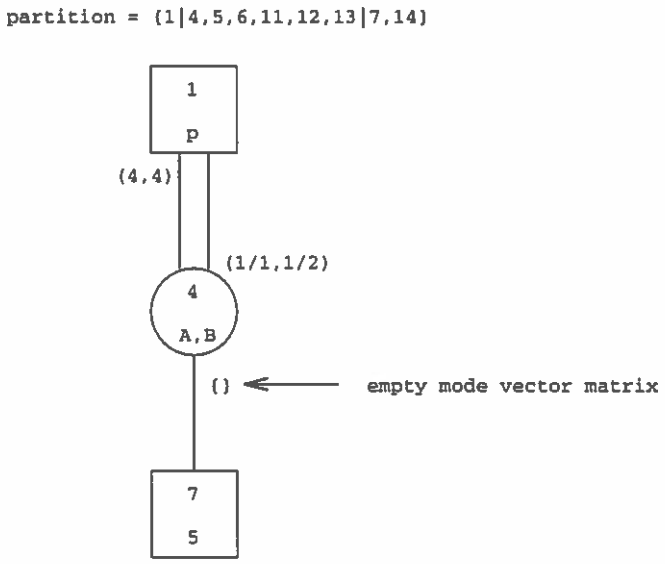


Figure 33: Graph of Simple Non-moded Program (Final)

3.3 Non-moded Program without Mode Consistency Checking

Consider the non-moded program [9]:

```
f(_,Z) :- true : true | h(Z).                (a)
f(X,Y) :- Y = b(X) : true | f(X,_).          (b)
h(b(s(W))) :- true : W = a | true.           (c)
```

The program is non-moded, since if the program were moded, variable `X` within the `b(X)` term in the ask guard of procedure `f/2` could not produce output for the program. If that variable occurrence did produce output, it would be sent *out* through the head occurrence and *into* the self-recursive call occurrence: a contradiction.

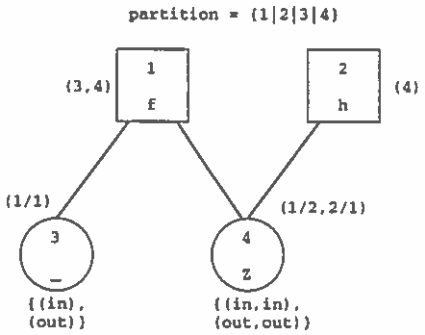
Now, we consider the program graphs without the mode consistency checking feature added. The initial graphs are shown in Figure 34 and 35. After we finish local reduction, the graphs are changed as shown in Figure 36 and 37. Finally, we do global reduction and the final graph is shown in Figure 38. From the final graph, we can see that no mode conflict has been detected, since there is no variable node in the graph having an empty mode vector matrix, and we can derive a mode for each mode path of the program.

3.4 Non-moded Program with Mode Consistency Checking

We now add the feature of mode consistency checking. The initial graphs of the program are changed from the previous section since we divide variable nodes into three categories: *normal, shared,* and *multiway.* The initial graphs are shown in Figure 39 and Figure 40. We follow the unification rules given in the previous chapter (Table 1) for doing local reduction. The first and the third clauses are reduced normally since there is no *shared* or *multiway* variable node in the graphs. However, the second clause reduction needs the mode consistency check because we have one *multiway* variable node. Although we need mode consistency checking for the second clause reduction, no new actions occur. This is because the *multiway* variable node has no child. Notice that when reducing the second clause graph, the *multiway* node must be changed to a *shared* node after reducing the node with itself. The minimal graphs for each clause of the program are shown in Figure 41 and Figure 42.

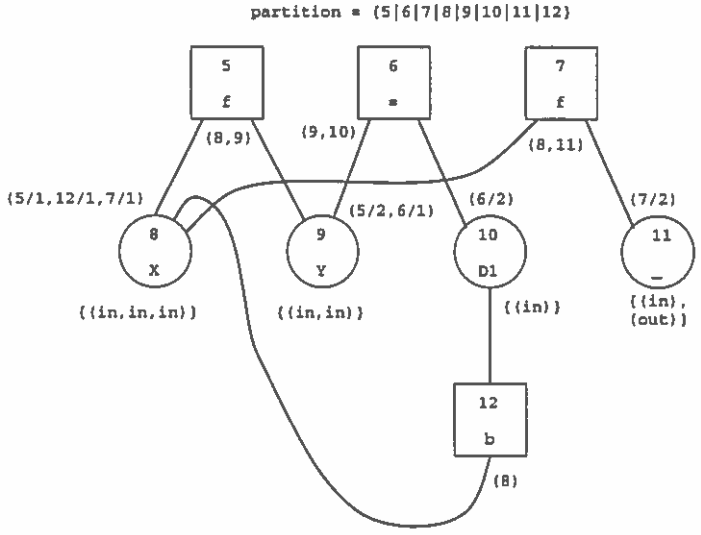
Finally, we follow the same reduction rules for doing global reduction. In reducing the first and the second clauses (graph (a) and (b)), standard reduction is performed (shown in Figure 43), because the *shared* node has no child. However, when we reduce the minimal graph of the third clause (graph (c)), we have to account for the mode consistency check. This is because the node 8 has *in* mode in all scenarios, and all its offspring must have the *in* mode. Thus, with the mode consistency check, the variable node 19 (`W`) has an empty mode vector matrix, and a mode conflict is detected. The final minimal graph of the program is shown in Figure 44.

Let's consider the simple non-moded program that we introduced at the beginning of this chapter. We see that none of the variable nodes in the initial graphs has type *shared* or *multiway.* Thus, mode consistency checking will never be used in analyzing the program. Therefore, the mode paths derived from the program will not change, and the mode conflict is still detected.




```
f(_,Z) :- true : true | h(Z).
```

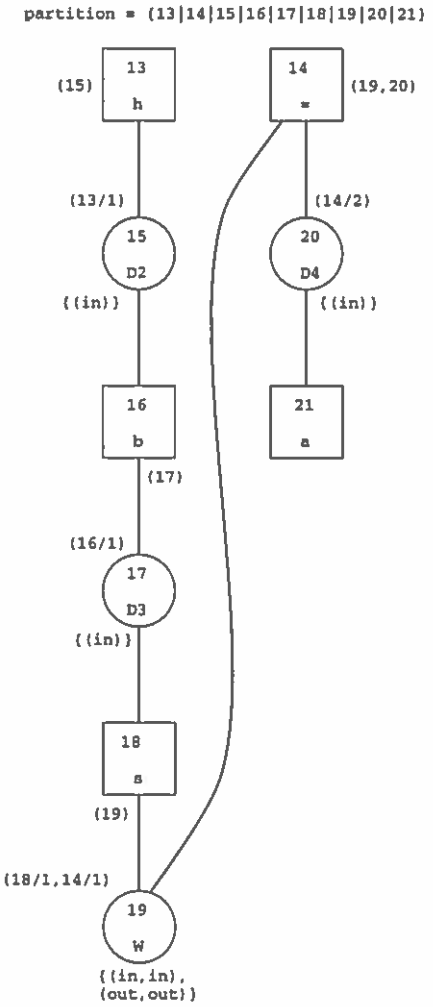
(a)




```
f(X,Y) :- Y = b(X) : true | f(X,_).
```

(b)

Figure 34: Non-moded Program Initial Graphs Part I




```
h(b(s(W))) :- true : W = a | true.
```

(c)

Figure 35: Non-moded Program Initial Graphs Part II

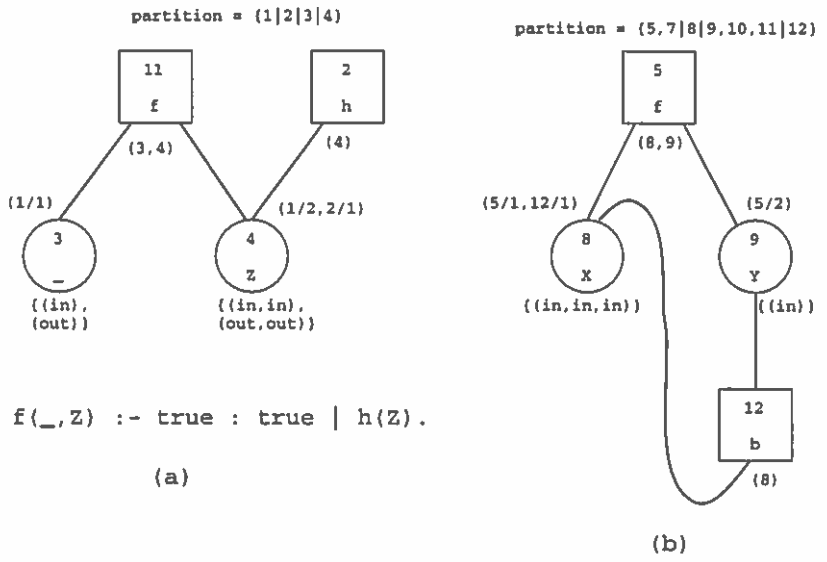
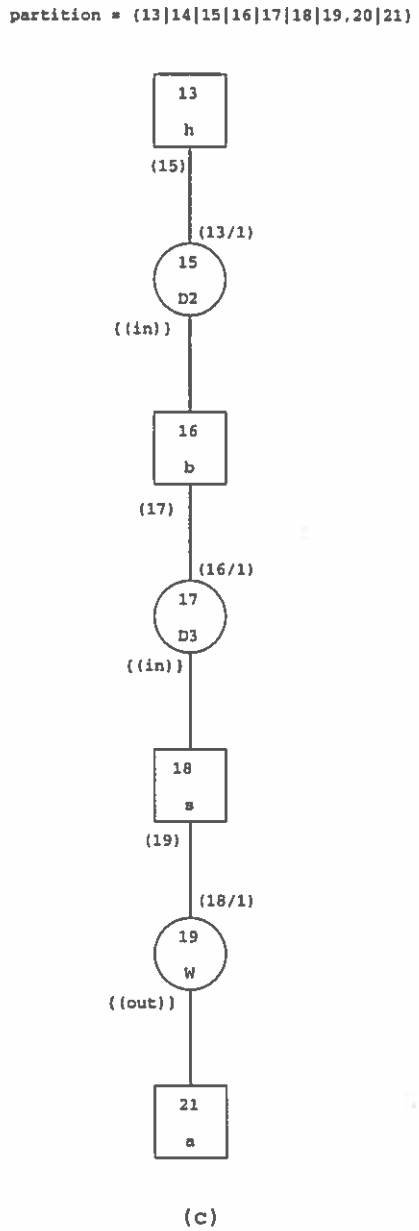


Figure 36: Minimal Non-moded Program Graphs Part I (Flawed Analysis)



(c)

Figure 37: Minimal Non-moded Prcgram Graphs Part II (Flawed Analysis)

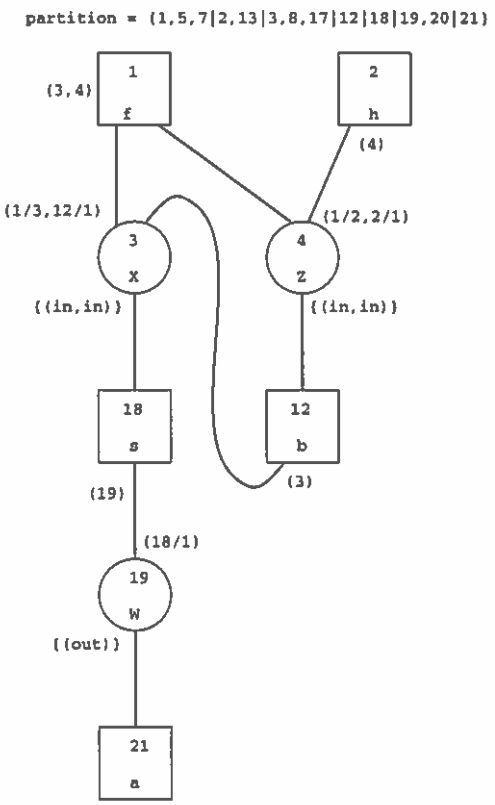
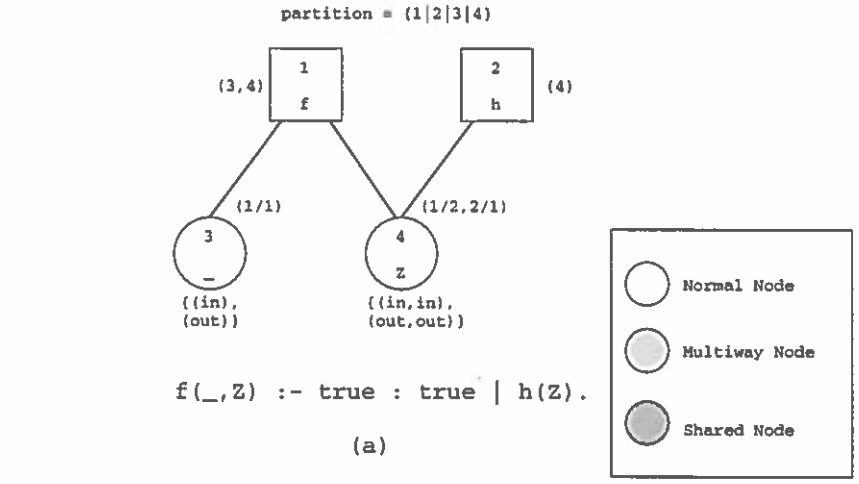
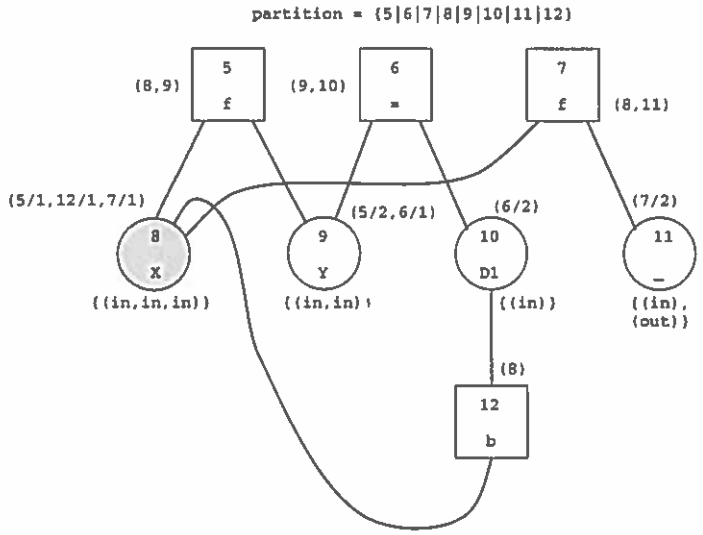


Figure 38: Final Non-moded Program Graph (Flawed Analysis)




```
f(_,Z) :- true : true | h(Z).
```

(a)




```
f(X,Y) :- Y = b(X) : true | f(X,_).
```

(b)

Figure 39: Non-moded Program Initial Graphs (Nodes marked) Part I

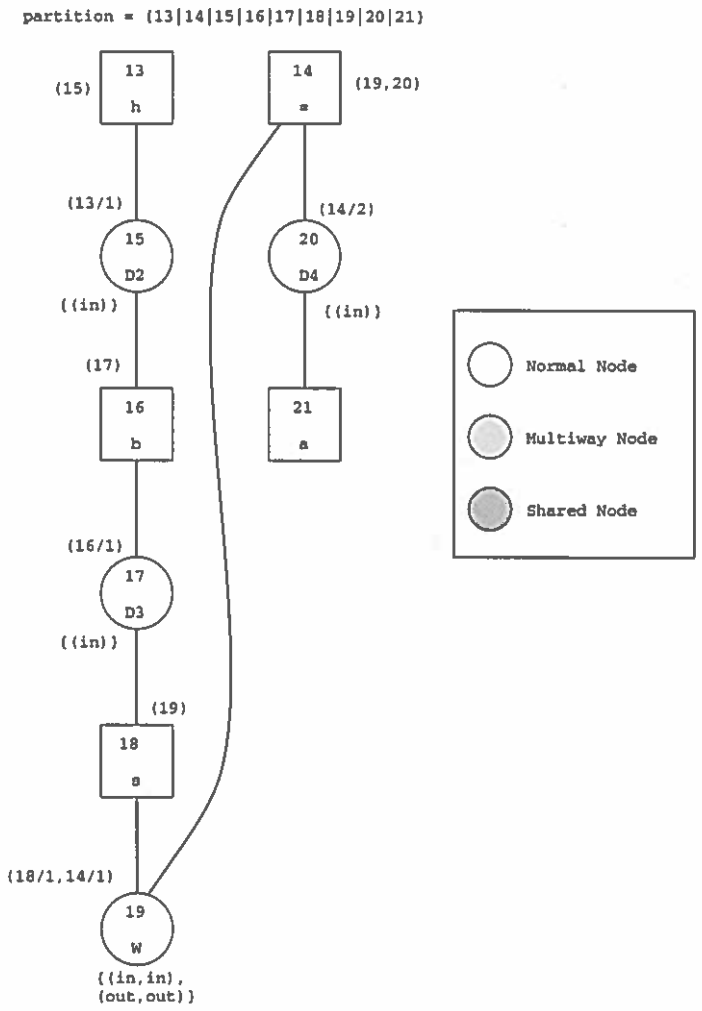


Figure 40: Non-moded Program Initial Graphs (Nodes marked) Part II

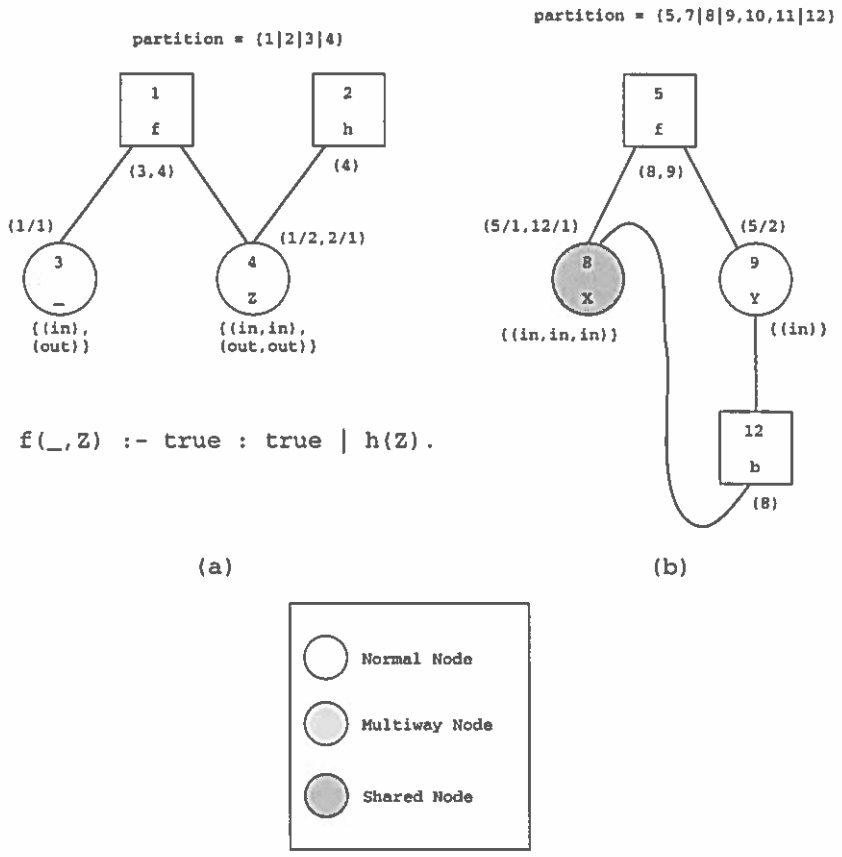


Figure 41: Minimal Non-moded Program Graphs (Nodes marked) Part I

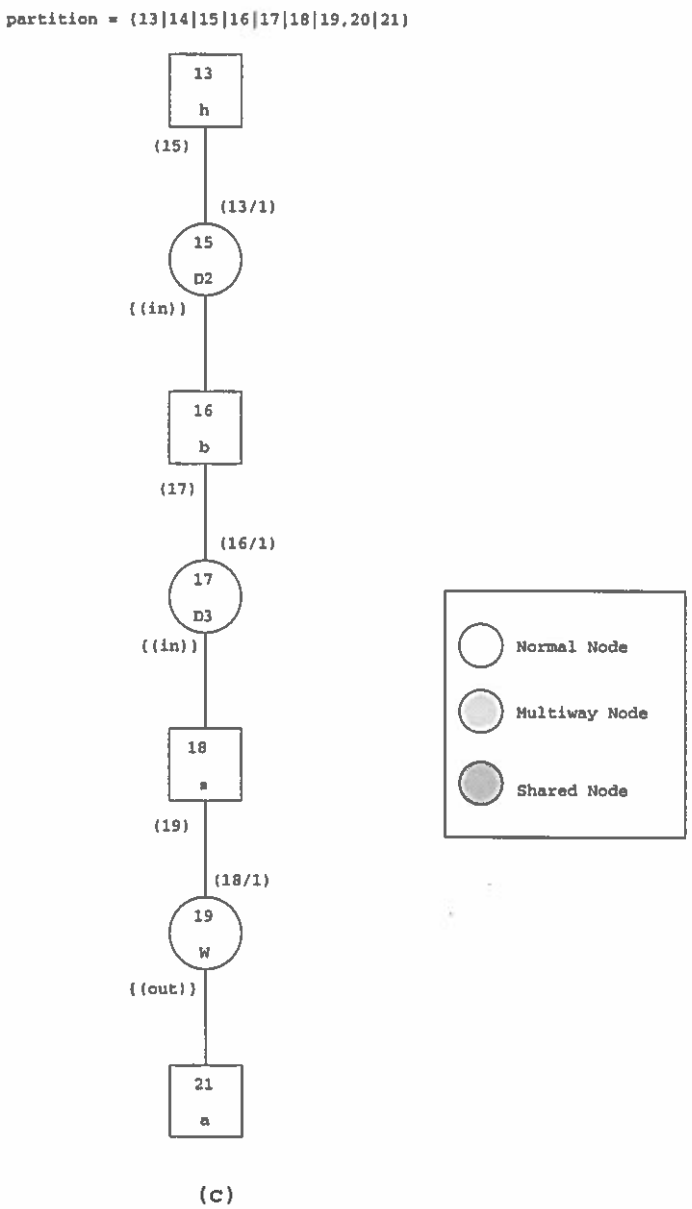


Figure 42: Minimal Non-moded Program Graphs (Nodes marked) Part II

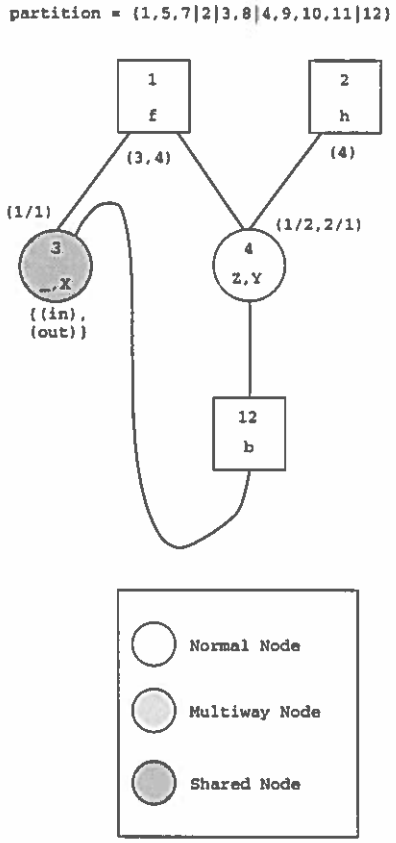


Figure 43: Global Unification of Non-moded Program Clause 1 & 2 (Nodes marked)

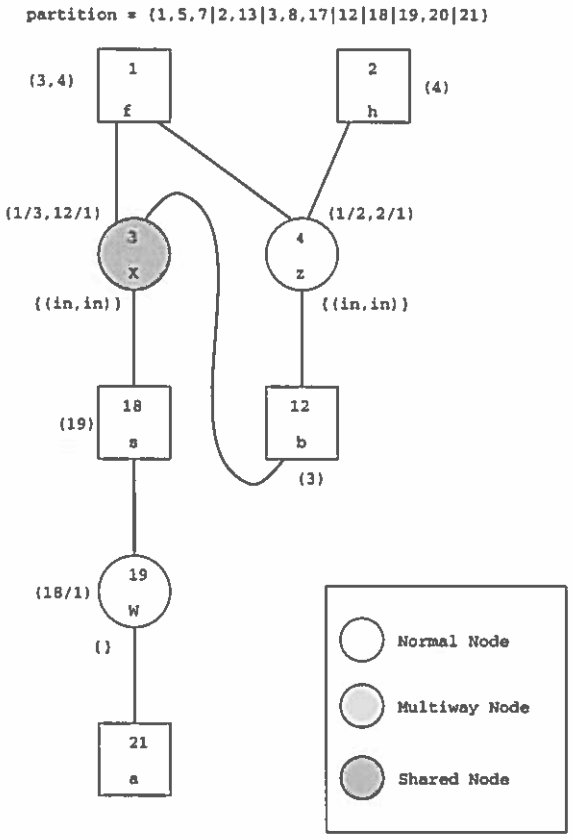


Figure 44: Final Non-moded Program Graph (Nodes marked)

Figure 45 shows some sample programs for which the mode analyzer detects mode conflicts. We believe the analyzer with mode consistency checking is correct, *i.e.*, it will always find a conflict if one exists and will never find one if none exist.

```
Non-moded Program I
-------------------

  s(X,Y) :- Y = c | t(X), s(X,Y).
  s(X,Y) :- Y = b | u(X).

  t(X) :- X = h(A) | A = c.
  u(X) :- X = h(c) | true.

Non-moded Program II
--------------------

  s(X,Y) :- Y = c | t(X), s(X,Y).
  s(X,Y) :- Y = b | u(X).

  t(X) :- X = h(A) | A = c.
  u(X) :- true | X = h(c).

Non-moded Program III
---------------------

  s(X) :- true | u(X), s(X).
  u(X) :- true | v(X), u(X).
  v(X) :- X = a(Y) | Y = a.
```

Figure 45: Sample of Non-moded Programs

4 Analysis Cost and Performance Evaluation

In this chapter, we discuss the complexity of the analyzer by comparing our program graph representation and reduction method with the one used in Ueda's abstract mode system [14]. We also examine the characteristics of two mode-analysis algorithms (static-graph and active-graph reduction) by evaluating the timing of a benchmark suite on the implementation of each algorithm. An extended version of the performance evaluation (including other mode analyzers) can be found in Tick [9].

4.1 Cost of Analysis

In this section, we show the mapping between our program graph representation and the program graph used in Ueda's abstract mode analysis [14]. We use a producer-consumer stack program [14] as the example program in this section. The Stack program is shown in Figure 46.

```
drive(M,S) :- M =:= 0 :
     S = [] |
     true.
drive(M,S) :- M =\= 0 :
     S = [push(M),pop(N)|S1] |
     subtract(N,1,N1),
     drive(N1,S1).

stack(S,D) :- S = [] :
     true |
     terminate(D).
stack(S,D) :- S = [push(X)|S1] :
     true |
     stack(S1,p(X,D)).
stack(S,D) :- S = [pop(X)|S1], D = p(Y,D1) :
     true |
     X = Y,
     stack(S,D1).

terminate(D) :- true :
     true |
     true.
```

Figure 46: A Producer and Consumer Stack Program (From Ueda)

Ueda's mode analysis produces a minimal graph for the drive procedure as in Figure 47. The minimal graph of the procedure produced by our analyzer is shown in Figure 48. These two graphs represent the same minimal graph. The differences between these two graphs are syntactical. We can define a one-to-one mapping between each notation used in these two graphs. A structure node in our graph is denoted by a branch in Ueda's graph. That is, our node 1 is denoted by a branch at the top of Ueda's graph. Our variable node is denoted by a circle node in Ueda's graph. Ueda has a notion of *input ground term*. We explicitly represent the term by a subgraph where in each node of the subgraph has *in* mode. The list construction structure node ([_|_]) is represented by "." in Ueda's graph. Notice that Ueda's has no explicit notion of a mode vector matrix. He uses a circle node with a plus sign inside to represent input for all entry-ports and uses an inversion node

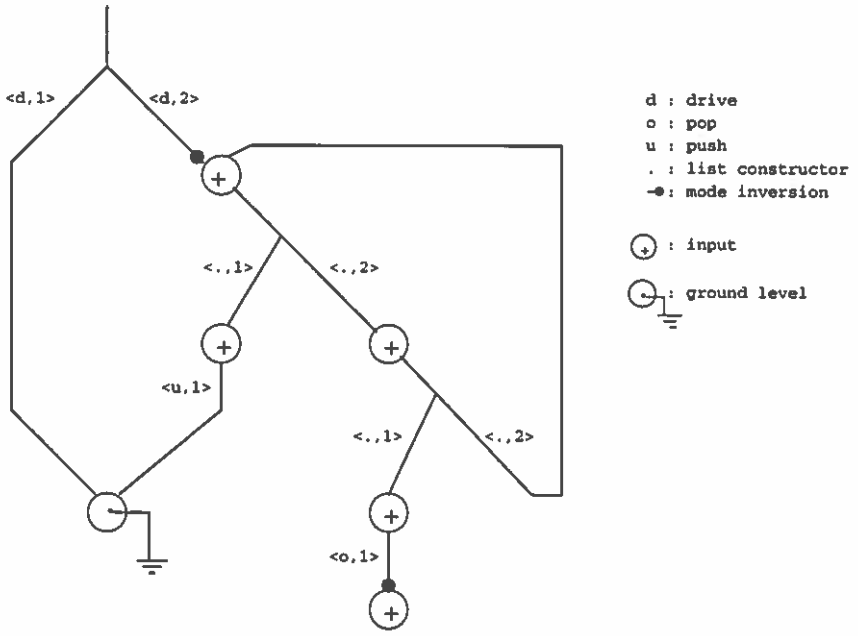


Figure 47: Ueda's Final Graph of **drive** Procedure (From Ueda)

to cast an output mode. In our final graph, we represent the same path modes but they are presented in the form of a mode vector matrix that has one vector in the matrix. The mode inversion is denoted implicitly in our graph. Effectively, mode inversion is done when we derive all modes of the paths using the mode polarity rule when we output the modes (explained in Chapter 2.) Actually these two graphs are isomorphic. Moreover, the modes of paths derived from these two graphs are identical, as shown in Figure 49.

The time complexity derived by Ueda for his algorithm is $O(l \cdot \alpha(l))$,where l is the size of the graphs to be unified, and α is the inverse of the Ackermann function [14, 3]. Since our program graph is isomorphic to Ueda's program graph, and we both use an abstract unification algorithm to unify (reduce) subgraphs, we claim that our mode analyzer has the same time complexity as Ueda's algorithm. However as mentioned earlier Ueda has no explicit notion of mode vector matrix. We use these mode vector matrices, however, as the key in finding modes of paths, and the cross-product operation of mode vector matrices is the major complexity of our algorithm, which is $O(n^3)$. This fact might destroy our claim about the time complexity above. However, the result of our experiment shows that in unifying graphs, the mode vector matrix usually contains only one or two mode vectors. Thus, the cross-product of two mode vector matrices has constant time complexity, and our claim still holds. However, for some particular programs and order of subgraph unification, the mode vector matrix size might get larger during the subgraphs unification process, and our claim will not be true. A solution to this problem is the mechanism of local and global unification that we introduced in chapter 2. The mechanism guarantees that after local unification, the mode vector matrix will be small in terms of the number of vectors in the matirx. Also, when we do global unification, the size of the mode vector matrix will never

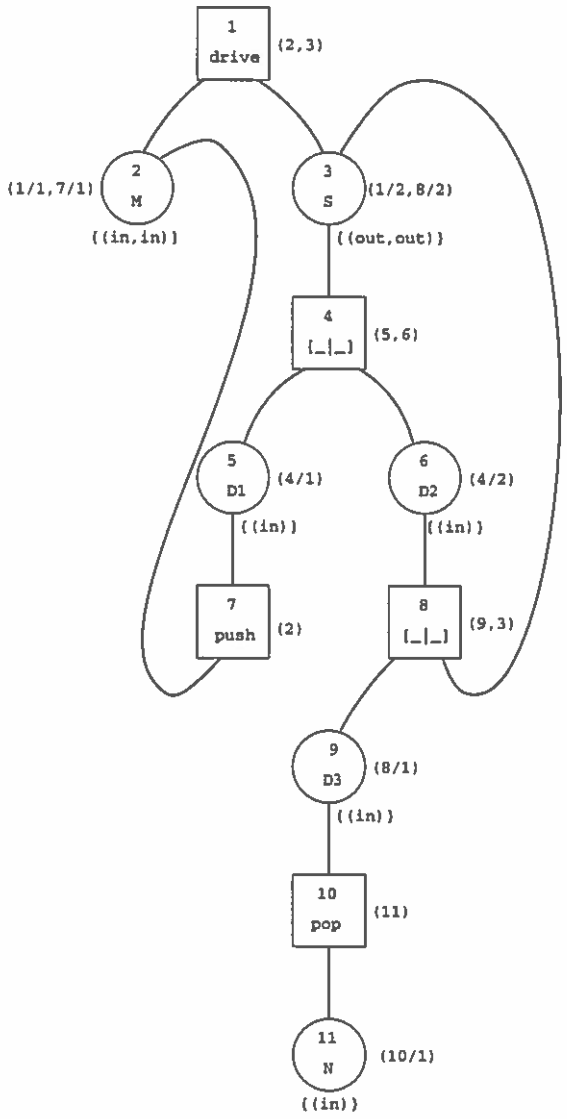


Figure 48: Final Graph of `drive` Procedure

$$
\begin{aligned}
&m(\{\langle drive/2,1\rangle\}) = in\\
&m(\{\langle drive/2,2\rangle\langle./2,1\rangle\langle push/1,1\rangle\}) = out\\
&m(\{\langle drive/2,2\rangle\langle./2,1\rangle\}) = out\\
&m(\{\langle drive/2,2\rangle\langle./2,1\rangle\langle pop/1,1\rangle\}) = in\\
&m(\{\langle drive/2,2\rangle\langle./2,1\rangle\}) = out\\
&m(\{\langle drive/2,2\rangle\langle./2,2\rangle\}) = out\\
&m(\{\langle drive/2,2\rangle = out
\end{aligned}
$$

Figure 49: Path Modes of the Stack Program

get larger in term of the vector in the matrix. Thus, our claim still holds and our mode analyzer has the same time complexity as Ueda's algorithm.

4.2 Performance Evaluation

In this section, we examine the performance characteristics of two mode analyzers, implementing the same mode analysis algorithm. The static mode analyzer uses a static data structure for graph representation, and the reductions are done explicitly by manipulating the data structure. The active mode analyzer uses a process network to represent the graph. Each node is represented by a concurrent process. The edges are represented by streams upon which messages pass between processes. We will discuss the differences between these two implementation techniques in terms of execution time and memory usage. The benchmark suite is a collection of KL1 programs. KL1 [15] is a language in the FGHC family. The programs vary from simple, common programs to complex programs. A description of the benchmark suite is shown in Table 2. The benchmark suite characteristics are shown in Table 3.

The complexity of the algorithms is proportional to the number of symbols and variable instances in the source program, as shown in Table 3. In calculating the number of paths, and since paths can be cyclic, we break the cycle (as described in chapter II), *e.g.,* the *car* and *cdr* will be counted, but not the *cadr* and *cddr*. The number of paths and average path length generated by the two analyzers are the same, since their output is identical. All the programs in the suite are fully moded except for waves which has a mode conflict. Both mode analyzers detect this conflict.

4.3 Execution Measurements

Both mode analyzers were implemented in the KL1 language. The systems run on the PDSS (V2.52.19) compiler-based system. PDSS is a nonoptimizing compiler-based emulator for KL1. The machine that we used for timing the execution is a Sun Sparcstation 10/30. The size of both mode analyzers is shown in Table 4.

Table 5 shows the breakdown, by phase, of the execution times (msec) using the benchmark programs for the static-graph and active-graph reduction mode analyzers. For the

Benchmark	Description
stack	a producer-consumer stack program [14].
qsort	an implementation of the Quicksort algorithm.
primes	a prime number finding program [8].
msort	a merge sorting program.
queens	the implementation of N-Queens problem [8].
cubes	Instant Insanity (candidates/noncandidates) program [8].
pascal	the Pascal's triangle program [8].
mandel	a program that computes the Mandelbrot function. Originally, the program was written in Strand, [2] and it was translated into KL1 by E. Tick.
rucs	a rule check system. It was written in KL1 by R. Ezaki from the original Prolog by Y. Koseko.
bestpath	the implementation of Dijkstra's single-source best-path algorithm with a heap data structure [8].
waltz	Waltz's constraint satisfaction algorithm [8].
waves	builds a multidimensional torus, using an interactive technique [8] (originally, written in Strand by I. Foster).
triangle	peg board game [8].

Table 2: The Benchmark Suite Description

			symbols			broken paths	
program	proc	clause	const	vars	total	totals	avg length†
stack	4	7	31	35	66	21†	1.8
qsort	2	5	31	40	71	19	1.6
primes	6	12	49	63	112	33	1.5
msort	4	11	54	75	129	36	1.7
queens	6	14	77	119	196	71	1.8
cubes	9	16	93	159	252	224	2.7
pascal	11	22	143	200	343	338	2.0
mandel	18	26	170	334	504	408	2.4
rucs	16	66	218	390	608	79	1.6
bestpath	20	44	279	492	771	507	2.5
waltz	20	54	333	630	963	329	2.2
waves	20	45	352	690	1042	623	3.0
triangle	42	80	315	1226	1541	1155	2.0

† average path length for graph analyzers.

Table 3: Benchmark Suite Characteristics

	static graph	active graph
parser	787	484
reducer	1,558	752
misc.	442	297
total	2,787	1,533

Table 4: KL1 Source Lines of Each Analyzer

static-graph reduction mode analyzer, each phase corresponds to the phases defined in earlier chapters. For the active-graph reduction mode analyzer, the first phase is used to parse the source program, and the whole process of graph reduction is done in the second phase. Notice that the two analyzers have the same number of calls to the *unify* function, as shown in the "unify" column. That is the analyzers implement essentially the same algorithm. The differences in data structure representation and programming style make the number of reductions differ between the two analyzers. We can see that the number of reductions of the static-graph reduction mode analyzer is about one to four times greater than the number of reductions of the active-graph reduction mode analyzer.

Table 6 shows a comparison of both mode analyzers' execution times. The garbage collection time is included in the execution time of both mode analyzers. From Table 6, we can see that the static-graph analyzer has greater analyzing time and memory consumption than the active-graph analyzer. One of the reasons is the difference in data structures used. Consider for some large benchmarks such as **rucs**, **bestpath**, **waltz**, **waves**, and **triangle**, the static-graph analyzer has an execution time of one to four times higher than both PDSS and the active-graph analyzer. We believe that the static-graph analyzer, which was our first prototype, has a memory leak problem. The memory leak problem causes the execution of the analyzer to slow down dramatically because of frequent garbage collection.

5 Conclusions and Future Work

Mode information is useful information which can lead to a number of optimization techniques in the compilation of concurrent logic programs. In this thesis, we implemented a static mode analyzer which can be a part of the front-end of an optimizing compiler. The implementation can derive correct and complete mode information from source programs. However, the use of the analyzer for analyzing large programs is still impractical. There are a number of ways that we can make changes in order to improve the performance of the analyzer, such as:

- Redesigning the data structures used. The data structures used in the analyzer are an important factor that dominates the performance of the analyzer. If we redesign the data structures in such a way that we can keep the graph in compact memory and can access information in the graph quickly and require less copying, the memory usage and the garbage collection required will be reduced. This will dramatically improve

Static Graph

benchmark	execution time (msec)				red x1000	unify	red/ unify
	phase I	phase II	phase III	total			
stack	150 (51.7%)	10 (3.4%)	130 (44.8%)	290	13	24	541
qsort	200 (52.6%)	30 (7.9%)	150 (39.5%)	380	16	26	615
primes	250 (41.0%)	50 (8.2%)	310 (50.8%)	610	28	43	651
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

benchmark	execution time (msec)			red x1000	unify	red/ unify
	phase I	phase II	total			
stack	180 (78.3%)	50 (21.7%)	230	7	24	292
qsort	180 (78.3%)	50 (21.7%)	230	8	26	308
primes	270 (73.0%)	100 (27.0%)	370	12	43	279
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 Reduction Mode Analyzers (KL1 on Sun Sparcstation 10/30)

benchmark	PDSS compile	static graph	active graph	static graph	active graph
	exection time (msec)			normalized by PDSS	
stack	470	290	230	0.62	0.49
qsort	410	380	230	0.93	0.56
primes	780	610	370	0.78	0.47
msort	760	760	490	1.00	0.64
queens	1,140	1,210	620	1.06	0.54
cubes	1,570	1,730	1,080	1.10	0.69
pascal	1,660	2,410	1,080	1.45	0.65
mandel	3,340	3,720	1,750	1.11	0.52
rucs	3,010	6,370	2,350	2.12	0.78
bestpath	6,160	11,220	3,630	1.82	0.59
waltz	4,510	11,730	5,700	2.60	1.26
waves	7,960	16,320	4,690	2.05	0.61
triangle	11,720	51,440	9,660	4.39	0.82
	PDSS	static graph	active graph	static graph	active graph
	memory consumption (kbytes)			normalized by PDSS	
stack	108	264	132	2.44	1.22
qsort	111	373	138	3.36	1.24
primes	164	616	204	3.76	1.24
msort	158	795	281	5.03	0.35
queens	244	1,444	340	5.92	0.24
cubes	320	2,070	571	6.47	0.28
pascal	343	3,272	574	9.54	1.67
mandel	616	6,561	880	10.65	1.43
rucs	699	8,224	1,343	11.77	1.92
bestpath	922	17,298	1,779	18.76	1.93
waltz	803	17,696	2,278	22.04	2.84
waves	1,204	28,371	2,318	23.56	1.93
triangle	1,865	51,656	4,437	27.70	2.38
analyzers	273	65	50	0.24	0.18

Table 6: Performance of Mode Analyzers (KL1 on Sun Sparcstation 10/30)

the execution time of the analyzer.

- Adding incremental mode analyzing. Currently the analyzer requires a large memory space to analyze a large program. This is a major cause of the long execution times of the analyzer. One way we can solve this is to divide the source program into a number of smaller programs and analyze each piece separately. Eventually, we add all the information that we get from analyzing each piece and get the complete information of the whole program.

- Adding heuristics in graph reduction. As we described in the Chapter 2, the mode analyzer will terminate whenever the graph is reduced into its minimal graph. Reducing the graph is done locally in program clause subgraphs, and then globally across program clause subgraphs. Actually, in analyzing a program, we might get all the mode information that we need (that is, every mode vector matrix now contains only one element, *i.e.*, one vector) before the graphs are reduced into the mininal graph. Furthermore, the order of reduction is a factor that may cause this situation to happen earlier. One idea is to investigate and to invent a heuristic that we can use to schedule the order of reduction in analyzing a program. Also, we can implement a mechanism that we can use to stop the reduction as soon as we have the information we need in the graph. However, this approach might not be safe because we might stop reducing program graphs before a mode conflict can be detected.

- Reusing of complex ground term subgraphs. As mentioned in Chapter 4, the large amount of memory used is one of the reasons why the static-graph analyzer has greater execution time than the active-graph analyzer. We can reduce the amount of memory used by the static-graph analyzer by reusing (sharing) subgraphs. A graph of the input ground term can shared because the mode of the term is fixed and will never be changed.

References

[1] Z. M. Ariola, B. C. Massey, M. Sami, and E. Tick. Compilation of Concurrent Declarative Languages. Technical Report CIS-TR-94-05, Dept. of Computer Science, University of Oregon, Eugene, OR 97403, March 1994.

[2] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[3] J. Jaffar. Efficient Unification Over Infinite Terms. *New Generation Computing*, 2(3):207–219, 1984.

[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. Demand-Driven Execution of Concurrent Logic Programs. In *International Conference on Parallel Architectures and Compilation Techniques*, Montreal, August 1994. North-Holland.

[6] B. C. Massey and E. Tick. Modes of Comprehension: Mode Analysis of Arrays and Array Comprehensions. In *International Logic Programming Symposium.* Cornell University, MIT Press, November 1994. Submitted.

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

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

[9] E. Tick. Practical Static Mode Analyses of Concurrent Logic Languages. In *International Conference on Parallel Architectures and Compilation Techniques*, Montreal, August 1994. North-Holland.

[10] E. Tick and M. Koshimura. Static Mode Analyses of Concurrent Logic Languages. *Journal of Programming Language Design and Implementation*, March 1994. Submitted. Also available as University of Oregon Technical Report CIS-TR-94-06.

[11] 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.

[12] K. Ueda. Guarded Horn Clauses. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 140–156. MIT Press, Cambridge MA., 1987.

[13] 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.

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

[15] K. Wade, M. Kohata, and D. Dure. *PDSS Manual (Version 2.52e)*. ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, December 1989.