Sequentialization Of Parallel Logic Programs With Mode Analysis

B. Massey

CIS-TR–92–18
August 1992

Abstract

The family of concurrent logic programming languages based on Flat-Guarded Horn Clauses has proven to be a great asset to programmers seeking to quickly construct efficient programs for highly parallel shared-memory machines. If these languages are to be implemented efficiently for other architectures, however, language-specific compile-time analysis techniques must be improved. This work describes a technique and implementation of automatic "mode analysis" (identification of input and output parameters) for a large subset of *FGHC* programs, and some possible techniques for the automatic "sequentialization" (ordering of body goals) of a subset of the fully-moded programs.

Department of Computer and Information Science
University of Oregon

Acknowledgements

Praise God from whom all blessings flow.

Thanks to my loving wife Joanie, who made this whole experience possible. And thanks to my advisor, Dr. Evan Tick. Without him I never would even have gotten involved in this topic, much less figured any of it out, and his generous support has enabled me to continue with this project.

This project was partially funded by the University of Oregon. Additional funding was provided by a National Science Foundation Presidential Young Investigator Award with matching funds generously provided by Sequent Computer Systems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 *Prolog*

It has been some years now since the idea of "Logic Programming" using "Horn-clause" logics first came upon the computer-scientific scene. The original work on theorem provers led quickly to the introduction of *Prolog*, a language containing key features of imperative programming and functional programming, but really based upon the idea of "resolution" in Horn-clause logics. Good accounts of the origins of *Prolog* were given in a recent issue of the Communications of the ACM [7, 20].

Prolog semantics are relatively straightforward, although a natural-language description, as given here, is perhaps less than satisfactory. A *Prolog* program consists of a number of "clauses." Each clause has a "head," consisting of zero or more "terms," and a "body," which consists of zero or more "body goals." The body goals are executed by trying to match them against identically named clause heads with the same number of terms in such a way that corresponding terms successfully "unify." A term is composed of "constants" and "variables" using various structuring features such as lists and vectors. Two terms are said to unify if variables can be bound to subterms in a consistent way so that the two terms are equal. If a body goal fails to match any clause head, it is said to "fail," and execution "backtracks" searching for a previous choice of bindings which would allow the program to proceed. A more formal semantics is available in Lloyd [21].

As a consequence of the above definition, a *Prolog* program may be read in three distinct ways: as an imperative program, as a functional program with "logical variables," and as a "declarative logic program" in which the program is viewed as requesting a binding of variables to values such that the Boolean disjunct of conjuncts of the body goals is "true."

1.2 *FGHC*

It is the declarative reading of *Prolog* programs which inspires the observation [8] that there is great potential for parallel execution of logic programs, since execution of body goals can be though of as subprogram execution which can proceed independently of execution of the rest of the program. One hindrance to this in *Prolog* is the potential for "backtracking," which requires that program state be unwound in the event of subprogram failure. Other important hindrances include dependent goals, task scheduling overhead, and the costs of maintaining logical variables.

The overhead involved in backtracking leads one to wonder whether it would be possible to simply insist that if a body goal fails, the whole program fails. Indeed, this is the basic idea behind "committed choice" logic programming languages [26]. The family of committed choice languages which has received the greatest attention is the "Guarded Horn Clause"

(*GHC*) languages [29]. The "guarded" refers to the "guard" constructs of the language, introduced because it is not easy to program without greater control than is provided by simple unification over which clause is chosen in case of a conflict.

The guard in generalized *GHC* languages may be an arbitrary construction of the programmer, but it proves convenient for efficient implementation to limit the programmer to so-called "flat" guards which are built into the language. This family of languages is known as "Flat Guarded Horn Clauses," and is the principal concern of this thesis. The formal semantics of *FGHC* for the purposes of this thesis are those of [26].

The *FGHC* language definition strongly implies fine-grained concurrent execution of *FGHC*; once each clause has been "committed to," its body goals may be executed independently on separate processors, and probably *must* be scheduled at runtime to avoid deadlock. The ability to execute goals in parallel is a useful property for execution of an *FGHC* program on a multiprocessor, since programs will run faster in direct proportion to the number of processors available. Unfortunately, the inability to schedule goals at compile time, the fact that there are typically many more body goals than processors, and the expense of scheduling goals at runtime combine to make it very difficult to utilize each processor efficiently.

1.3 Efficient *FGHC* Execution

The class of *FGHC* languages is capable of efficient parallel execution, but there are some potentially serious problems. Foremost is the idea of low granularity: the only useful computation in an *FGHC* program is done when selecting an appropriate clause for execution, when doing arithmetic, and when creating structure within a clause. These costs of computation may be swamped by the cost of expanding body goals and binding variables, limiting the potential efficiency of the language. Worse, each body goal must be assigned to a processor, which involves significant work by some sort of "scheduler" process — thus it is entirely possible for the scheduler to become the limiting factor in execution performance.

This thesis suggests a two-step approach to alleviating these problems, along lines explored earlier by others including Ueda and Morita [30, 31], Tick and Korsloot [19], and King and Soper [15, 17, 16]. First, the program is statically analyzed to discover where variables will be bound. This analysis allows traditional compiler optimizations to be applied to the compilation of *FGHC* programs, and also provides a framework for a second stage of analysis, which attempts to find body goals which may be executed sequentially without affecting the program semantics.

The hope is that eventually one will be able to write programs in a high-level-language similar to *FGHC* without regard to their parallel execution properties, and then execute them efficiently without source changes in uniprocessor, coarse-grained multiprocessor, and fine-grained multiprocessor environments. The author believes that he has illustrated how the combination of mode analysis and sequentialization may allow *FGHC* programs to be executed more efficiently in all of these environments, greatly increasing the ability of those unfamiliar with traditional parallel programming techniques to write efficient concurrent applications.

1.4 Outline of the Thesis

Chapter 1 gives some flavor of the research to those perhaps less sophisticated in the area of *FGHC* programming.

Chapter 2 covers "mode analysis" of *FGHC* programs, which attempts to discover where in a program variables are bound to values. Emphasis is placed on results of safety and completeness, and on the empirical results of an experimental implementation of this analysis.

Chapter 3 covers "sequentialization analysis" of *FGHC* programs, which, given mode information, discovers which body goals of a program may safely be executed in a sequential order determinable at compile time. Several possible approaches are explored, and their safety, completeness, and efficiency are discussed. Finally, empirical results of an experimental implementation of one of these techniques are given.

Chapter 4 summarizes the thesis, offers conclusions, and suggests future work in the thesis area.

Chapter 2

Mode Analysis

Mode information has been shown to be quite useful in the efficient compilation of logic programming languages. Primarily, mode information facilitates the strength reduction of unification operators into matches and assignments. There are numerous methods for automatic derivation of mode information from logic programs, e.g., [9, 10, 3, 22]. In committed-choice logic programs [25], the logic variable is overloaded to perform synchronization. Mode information can thus be used to optimize code generated for argument matching. We are particularly interested in mode analysis because it enables us to do static "sequentialization" analysis [19]. This analysis determines if automatic sequentialization of body goals is safe, i.e., cannot result in deadlock. The potential advantage of such a scheme is vastly improved efficiency in register allocation and procedure call protocol.

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

Ueda and Morita [30] outlined a mode analysis technique that showed great elegance and potential for efficiently deriving rich mode information. Essentially they suggested propagating a set of mode constraints around a control-flow graph representing the program. The analysis was restricted to "moded" flat committed-choice logic programs, although, for reasons explained below, this is not regarded as a major drawback. This work is a clarification of their method, a description of an algorithm and its implementation, and an evaluation of the method's performance. This paper is the first reported implementation of this kind of mode analysis technique, although we do not use graph propagation. Empirical results are also presented concerning the mode characteristics of a set of *FGHC* benchmarks.

This chapter is organized as follows: The path concept is reviewed and the procedure by which interesting paths are generated is described. The fundamental data representation used in mode analysis is described. The mode analysis axioms and their incorporation into an algorithm is described. The moding algorithm's complexity is discussed, as is its consistency, completeness, and safety, and the effect of the restriction to moded *FGHC*. The relationship to other work is summarized. Empirical measurements characterizing the performance of the algorithm are presented. Finally, conclusions are summarized and future research is suggested.

2.1 Path Generation

The first stage of the algorithm generates a finite set of paths whose modes are to be considered. Ueda and Morita's notion of "path" is adopted as follows: A path p "derives" a subterm s within a term t (written $p(t) \vdash s$) iff for some predicate f and some functors $a, b, \ldots$ the subterm denoted by descending into t along the sequence $\{< f, i >, < a, j >, < b, k >, \ldots\}$ (where $< f, i >$ is the i^{th} argument of the functor f) is s. A path thus corresponds to a descent through the structure of some object being passed as an argument to a function call. f is referred to as the "principal functor" of p. A program is "moded" if the modes of all possible paths in the program are consistent, where each path may have one of two modes [30]:

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

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

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

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

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

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

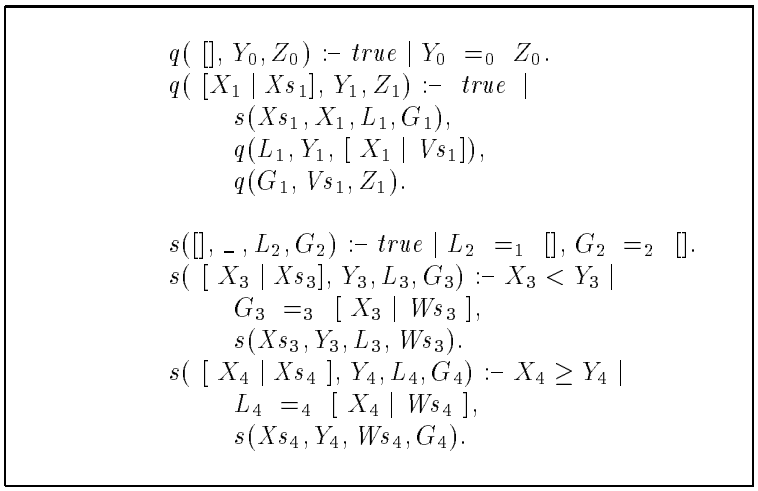


Figure 2.1: Quicksort *FGHC* Program: Weak Canonical Form

As an example of path generation, consider a Quicksort program[1] written in *FGHC*, shown in Figure 2.1. The program shown is the result of applying some syntactic transformations to produce a "weak canonical form" required by the subsequent algorithms. The left-hand side of every unification operator $=/2$ has been made a variable. No "extra" variables appear in unification body goals: all variables appearing in a unification body goal must also appear in some non-unification body position or in the head. All variables are qualified by the clause number in which they appear, in order to retain scoping information for subsequent passes. In addition, each unification operator is labeled with a unique integer.

In general, all interesting paths of a program are generated in a few sequential passes. The 39 interesting paths (23 input, 16 output) of Quicksort shown in Table 2.1 are generated in two passes[2]. In this example the unification body goals provide four additional interesting paths during the first pass, and no additional interesting paths during the second pass.

An important question is whether this set of paths represents a *minimal and complete* set of paths for the mode analysis. The answer to this may depend upon the use to which the mode analysis is put — nonetheless, as discussed on pages 14–15 of this thesis, there is good reason to believe that some fundamentally important paths may not be generated. However, in the benchmarks examined here the paths generated by the above algorithm prove to be almost entirely sufficient.

2.2 The Partition Table

The mode information for paths is derived from knowledge of the relationships between paths. Thus, our algorithm requires a systematic and efficient means of keeping track of these relationships as analysis proceeds. The computation of path relations is optimized

[1] This program is used as an example throughout the chapter.

[2] Note that $< ., 1 >$ and $< ., 2 >$ are respectively the *car* and *cdr* of a list structure.

Table 2.1: Interesting Paths of Quicksort

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

by partitioning the paths such that all the paths in each partition have the same mode. First, each path is placed in a unique partition, indicating that its mode is unknown and that its relationship with the modes of other paths is unknown. There are also two special partitions which are initially empty: the *input* and *output* partitions. Various relationships between the partitions are then asserted by predicates as we proceed through the analysis.[3] In order of decreasing precedence, the predicates are:

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

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

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

[3] This does *not* imply that the logic programming implementation uses an `assert` builtin!

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

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

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

§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.'[a]
2. If the head contains one (or more) occurrences of the shared variable (so the body has two or more occurrences), then the modes are as follows:
   (a) If the mode of the head occurrence is 'in,' the modes of all body occurrences are 'in' as well.
   (b) If the mode of the head occurrence is 'out,' then *one* of the body occurrences is 'out,' and the other body occurrences are 'in.'

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

Figure 2.2: Ueda and Morita's Mode Derivation Axioms

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

2.3 Mode Analysis

The second stage in the algorithm is to derive the modes of paths generated by the first stage. This is accomplished by finding absolute modes for a small number of paths and then examining relationships between the modes of paths. This mode analysis exploits the rules outlined by Ueda and Morita. Tick and Korsloot's [19] formulation of their axioms is given in Figure 2.2 (in the figure, $m(p)$ denotes the mode of path p). Again the algorithm repeatedly scans sequentially through the program, this time deriving modes of paths. The critical insight is that given a variable v, the modes of all paths deriving v must be related

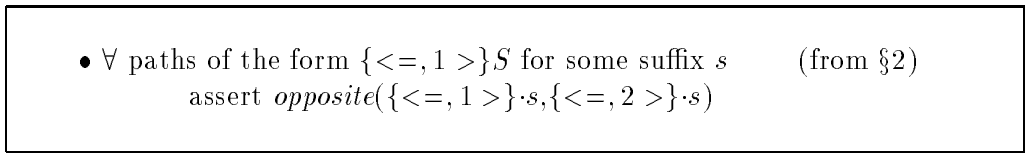


Figure 2.3: Unification Analysis Algorithm

via the mode axioms. The mode analysis algorithm proceeds in four steps:

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

2.3.1 Syntactic Analysis

During the first step in mode analysis a single syntactic pass is made over the program, noting paths which lead to constants, variables, and guard arguments. All occurrences of each variable in the module being analyzed which are derivable from each interesting path are recorded; information which will be used by all succeeding steps of the algorithm. Whether the variable occurrence was in the head, guard, or body of its clause is also recorded. Constants in interesting path positions are noted and the partition table is updated according to §1.1 of Figure 2.2. This will generally be a rich source of information about *in* paths. In Quicksort, this fixes the modes of paths such as $\{< q, 1 >\}$ as *in*. The modes of paths leading to non-unification guard arguments in the partition table are then asserted according to §1.3 of Figure 2.2. In Quicksort, this fixes the modes of paths such as $\{<\text{`}\geq\text{'}, 1 >\}$ as *in*.

2.3.2 Body Unification Analysis

This step simply asserts that corresponding paths on opposite sides of a body unification goal have opposite modes. This relationship corresponds to §2 of Figure 2.2, and is implemented according to the algorithm shown in Figure 2.3. It is generally efficient to assert these relationships early, since it allows greater propagation of information asserted by later steps. For example, in Quicksort clause #0 $opposite(\{<=_1, 1 >\},\{<=_1, 2 >\})$ is asserted.

Note the universal quantifiers in the algorithms of Figures 2.3, 2.4, and 2.5. Our implementation of these depends on the fact that a *finite* (and indeed a small) set of paths are generated for the target program, in contrast to the work described on pages 15–16 of this thesis.

2.3.3 Binary Analysis

This step derives the modes of paths which have binary relations. These relationships correspond to §1 and §3 of Figure 2.2. These rules are implemented according to the

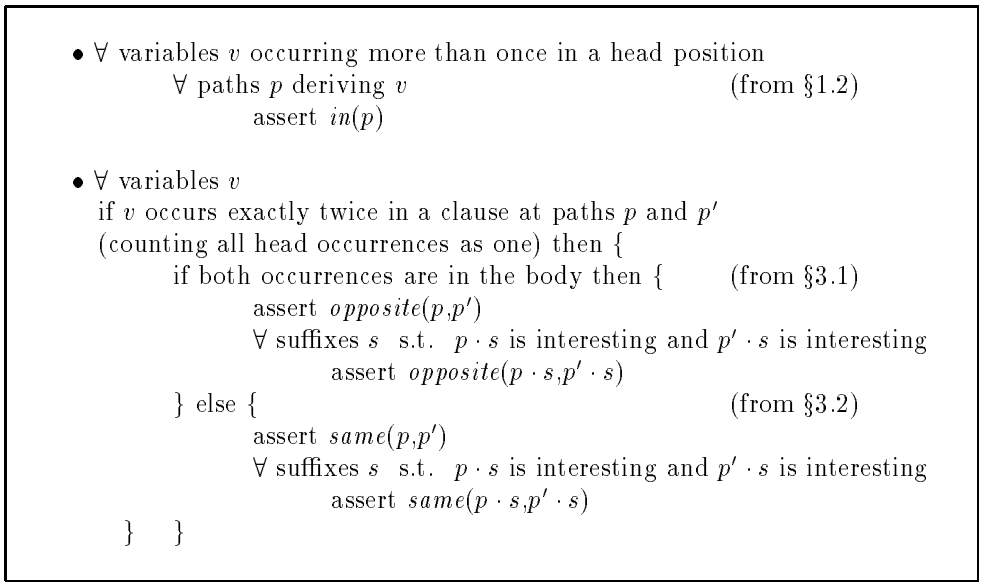


Figure 2.4: Binary Analysis Algorithm

algorithm shown in Figure 2.4. Note that the ordering of operations of this algorithm is somewhat arbitrary. This particular ordering was chosen both for efficiency and ease of implementation, but it is not unlikely that some other order could be faster or simpler. In particular, each rule is currently applied in turn to all of the applicable objects in the program. Reversing the nesting order so that all possible rules are applied to each syntactic object in turn would not affect the correctness of the algorithm and might provide some speed increase.

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

2.3.4 Multiway Analysis

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

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

Figure 2.5: Multiway Analysis Algorithm

Table 2.2: Possible Modes For X_1

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

Therefore the multiway rule is implemented according to the recursive generate-and-test algorithm of Figure 2.5. Starting with the partition table output by binary analysis, the algorithm examines all possible values for each set of mutually constrained paths generated by occurrences of a variable meeting the conditions of §4 of Figure 2.2. Each of these possibilities is tested by applying the multiway algorithm recursively to the remaining constraints, and the resulting collections of partition tables are merged and returned. Thus, the overall structure of the call graph of the test algorithm is a tree whose leaves are each either a partition table or a failure indication — the output of the test algorithm is simply the collection of partition table leaves of the tree. Note that the shape of the tree is determined only by the output of the iterative generate algorithm: thus, the recursive test function could easily be made iterative via standard transformations. Note also that since our implementation of this algorithm is in *FGHC*, subtrees will naturally be evaluated concurrently in a parallel implementation of the language.

In Quicksort only clause #1 meets the multiway criteria, where X_1 occurs once in the head and twice in the body. The three possibilities to be checked are summarized in Table 2.2. By this point in the analysis, however, we have already derived that $\{< q, 1 >, < ., 1 >\}$ is *in* and that $\{< s, 2 >\}$ is *in*, so we find that $\{< q, 3 >, < ., 1 >\}$ is *in*. This example is nice because it is completely determinate — the multiway rule derives only one set of possible modes for the program. In general, this may not be the case and several possible final modings for the program will be emitted.

2.3.5 Complexity

The complexity of the algorithm can best be understood by examining its component pieces. Everything up to the beginning of binary analysis is fundamentally linear on the length of the program — a small fixed number of passes are made over the program to derive facts about it. The binary analysis is also close to linear on the number of variables in the program meeting the constraints of Figure 2.4. A significant quadratic component derives from the fact that inner loops of the analysis iterate over a set of suffixes of paths — the size of this set is approximately linearly proportional to some measure of the "complexity" of the program.

The multiway analysis is difficult to analyze. If it were performed first, it would be exponential in the number of variables meeting the constraints of Figure 2.5, but by the time it is actually performed, most alternatives contradict the known modes, and thus are not explored further. In practice, the time spent in this analysis seems to be reasonably short, as seen from the timings on page 21.

2.3.6 Consistency, Completeness, Safety, Restrictions

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

The current implementation will report any contradiction, ignore the contradictory assertion, and proceed with the derivation. This allows the user to examine the final modes produced by the analysis and determine which might be incorrect. In our experience, this is usually sufficient to correct the problem. In the absence of user intervention, this also in practice allows the modes of most of the remaining paths to be determined. For example, when using the mode information for sequentialization [19], we may sequentialize all calls not involving a conflicting path, and then "safely" compile calls involving the conflicting path.

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

> *Definition*: A set of paths generated for a program is *complete* iff the existence of a consistent moding for the set of paths implies that the program is fully-moded. (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. As an example of the incompleteness of our path generation algorithm, consider the program

a :– $true \mid b(2)$.
$b(Z)$:– $true \mid c(Y),\ d(Y,Z)$.
$c(X)$:– $true \mid X = [1]$.
$d([V \mid \_], W)$:– $true \mid W = V$.

With the path generation as described, the path $\{< c,1 >, < .,1 >\}$ will never be considered. But without noticing that this path is *out*, it is impossible to discover that the path $\{< d,2 >\}$ is *out* and thus derive the non-moded call in $a/0$.

It is difficult to extend path generation in such a way as to obtain a finite complete set of paths, much less a minimal one. We have expended much thought and effort on this problem, but are currently unaware of an adequate solution. In fact, it is probable that there exist programs for which no finite complete set of paths exist.

Note that the path generation algorithm previously described on pages 6–7 is *unsafe*. It is also a consequence of the incomplete set of generated paths that even if the program

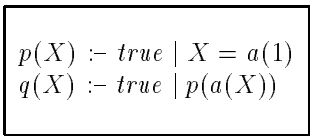


Figure 2.6: Mode Conflict

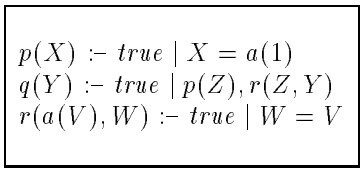


Figure 2.7: Mode Conflict Resolved

contains information about the mode of a path, that information may not be derived by the mode analysis algorithm. Nonetheless, most generated paths in typical programs are moded by this analysis, and if the program being analyzed is known to be moded, all modes derived are correct.

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

It is important to note that moded *FGHC* is a strict subset of *FGHC*, in the sense that there exist correct and meaningful *FGHC* programs which are nonetheless not moded. However Ueda's assertion in [31] that "Fortunately, most GHC programs written so far are written, or can be easily rewritten, following these conventions" seems to be correct.

Perhaps the most common moding error in common use is the use of passive unification in body goals to select elements. Thus, the program of Figure 2.6 is non-moded, since the mode of $\{< p, 1 >\}$ is clearly *out* according $p/1$, and yet that path is being bound in the body of $q/1$. Fortunately, this problem can be solved quite generally and even automatically, by transformations of the form shown in Figure 2.7

2.4 Comparison With Other Work

The original mode analysis scheme is due to Ueda and Morita [30], and was later clarified by them [31]. It is based on the representation of procedure paths and their relationships as rooted graphs ("rational trees"), and the utilization of unification over rational trees to combine the mode information obtainable from the various procedures.

The mode analysis rules were simplified to their present form by Korsloot and Tick [19], who also gave a set of simple inference rules for deriving paths and their modes. However, no selection algorithm is given for application of the inference rules. That work still deals

with an infinite set of possible paths, but considers only those paths with a finite known prefix. As testament to the inherent difficulty of dealing with inference rules, they failed to consider certain "interesting" paths and thus incompletely moded the Quicksort example!

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

Ueda and Morita's scheme is more general than ours, however. In particular, it can determine the modes of paths which we do not generate during interesting path generation. Their trees are constructed by placing nodes in all paths descending from the procedure and its associated body goals. We follow Ueda's notational conventions, in which procedure and body goal names are enclosed in square boxes, terms are enclosed in circles with numbered subterms, and a dot on an edge means mode inversion. Thus, for our quicksort example, the graphs for $q/3$ and $s/4$ might look something like those of Figure 2.8.

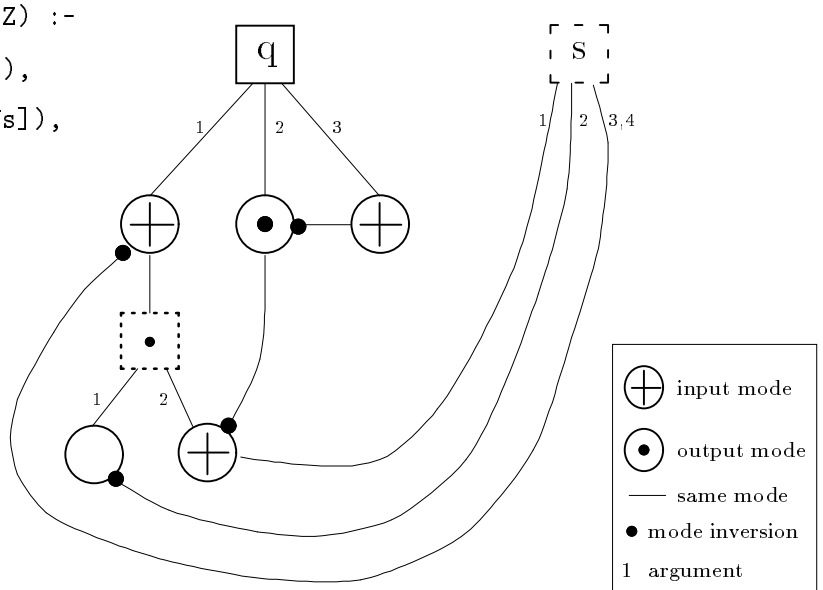
The rational unification of two graphs proceeds from the assertion that some identically labeled node in each of the two graphs is in fact the same node. One then traverses the two graphs, making an identity relationship between corresponding nodes in the graphs; this may mean that many nodes in one graph correspond to a single node in the other, and thus the substructure of the graphs is "compressed" in some sense. The unification proceeds until either both graphs have been completely traversed or until a conflict is found, in which latter case the unification fails. If we circularly unify the graphs of Figure 2.8, we obtain a graph something like that of Figure 2.9. Note that this unification fixes the mode of $\{< q, 1 >, < ., 1 >\}$ as *in*, whereas it was previously undetermined.

To see the potential advantage of Ueda and Morita's method, consider the example of page 14 with which we demonstrated the failure of our scheme. The graphs associated with this are those of Figure 2.10 (where the mode of $\{< d, 1 >, < ., 2 >\}$ is fixed in $d/2$ by the fact that an anonymous variable in the head must always be input, since it may never be bound by the body). Note that the graph of Figure 2.11 comprising the unification of the individual procedure graphs (other than that of $a/0$) fixes all the modes of all paths of these procedures. Then, the incorrect mode of $\{< b, 1 >\}$ in $a/0$ is easily identified.

One might expect that we could simply adjust the heuristic for interesting path generation to produce paths like the one we are lacking in this example, and thus mimic Ueda and Morita's scheme. However, we do not yet understand how to extend the interesting path generation while avoiding generating infinite sets of paths. Note that, since the graphs of Ueda and Morita's method may be circular, they may denote the mode of infinite sets of paths, and thus may contain more information than we possibly could store.

Ueda and Morita believe that their scheme will in general completely and consistently mode any *FGHC* program. Unfortunately, we know of no proof that this is indeed the case, though it seems likely that such a proof could be constructed. If so, this scheme could serve as a workable definition for the moded *FGHC* language. We believe our algorithm is no worse in execution time than the scheme of Ueda, and is easier to implement. In fact, our initial attempts to utilize graphs led us to our current algorithm. Nonetheless, because of the problems with completeness and safety, we should perhaps revert to Ueda and Morita's scheme.

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




```
s([],_,L,G) :-
 L=[],
 G=[].
s([X|Xs],Y,L,G) :-
 X<Y
 |
 G=[X|Ws],
 s(Xs,Y,L,Ws).
s([X|Xs],Y,L,G) :-
 X>=Y
 |
 L=[X|Ws],
 S(Xs,,Y,Ws,G).
```

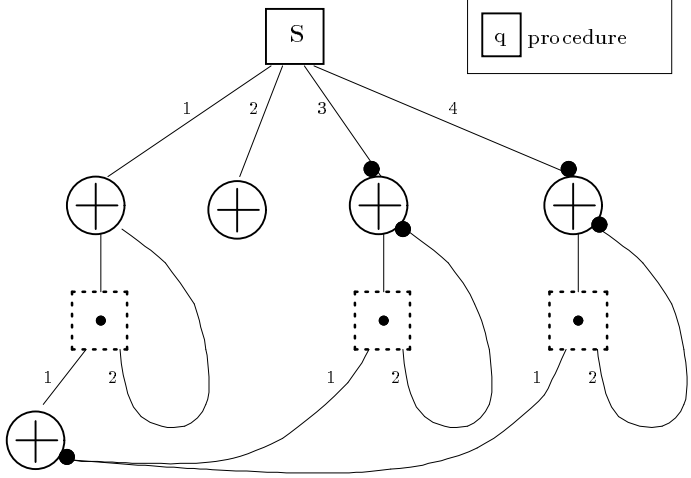


Figure 2.8: Graphs For q and s of qsort Example

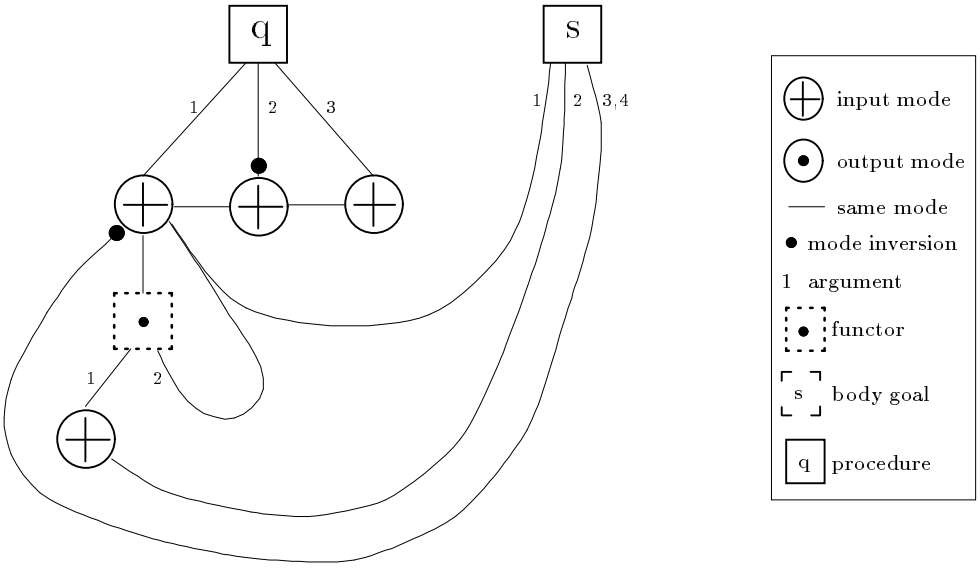


Figure 2.9: Unified Graph For qsort Example

Table 2.3: *FGHC* Benchmarks: Path Analysis with No Explicit Information

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

2.5 Empirical Results

Our experimental implementation of the path generation and mode analysis algorithms consists of approximately 4,000 lines of code comprising 23 modules totaling about 500 clauses, written in *FGHC* running under the PDSS system [5]. We examined the moding and execution characteristics of the analysis of a group of seven *FGHC* programs listed in Table 2.3, including one module (msort) from the implementation itself.

Table 2.3 shows the number and percentage of derived (input and output) paths, as well as the number of paths that could not be derived. The "missing modes" column shows the number of explicit modes needed to give modes to all paths. The benchmarks averaged 3.3% of non-derivable paths, with some variance. As mentioned on pages 14–15, there are two approaches which appear viable for reducing the percentage of non-derivable paths. One method is to generate a richer set of paths to drive the mode analysis. Currently, we

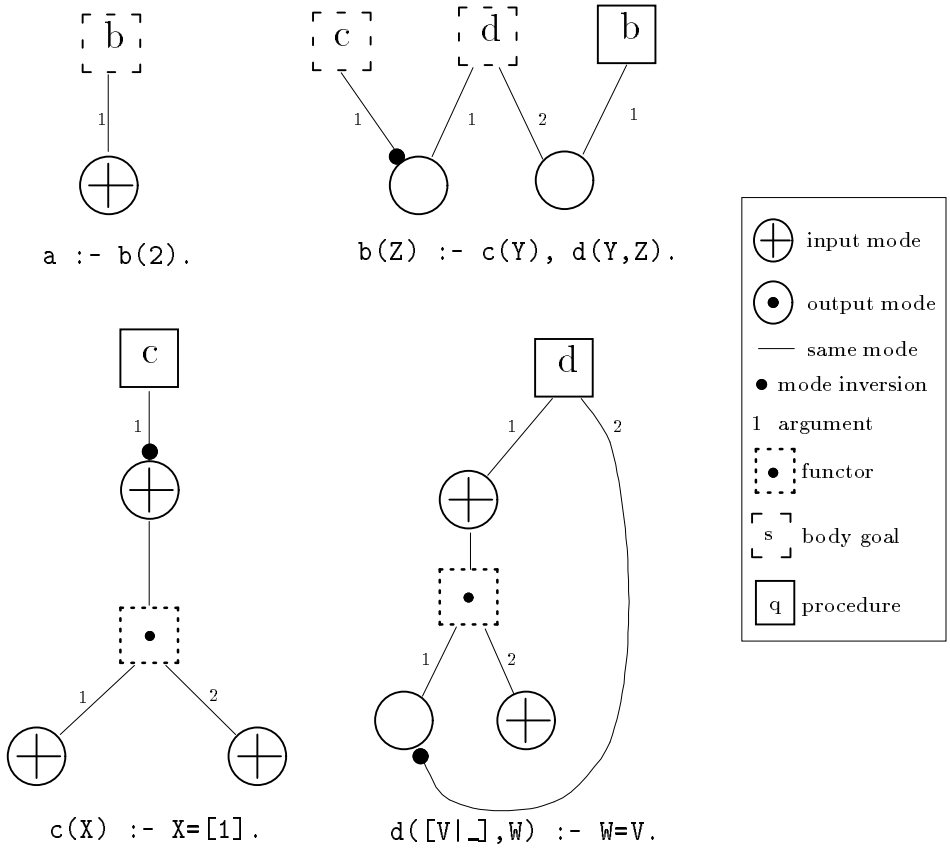


Figure 2.10: Clause Graphs For Error Example

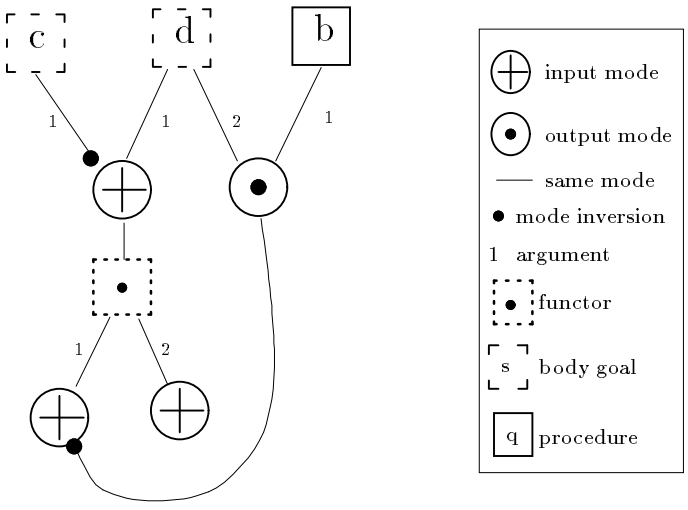


Figure 2.11: Unified b, c, and d From Error Example

Table 2.4: Breakdown of Paths by Type: Raw Counts and (Percentages)

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

generate only "simple" paths, which are local to each clause. This can be extended to more elaborate generation, effectively abstract interpretation on a domain of "important" paths.

A second method, illustrated in these measurements, is to have the programmer give explicit mode declarations to help the analysis. The final column of Table 2.3 gives the number of explicit path modes needed to permit the derivation of all paths. Note that only a fraction of the unknown paths are needed to fully constrain those remaining. Waltz has the largest requirement because its data structure manipulation is far more complex than that of the other programs. Local path generation does not suffice because deeply nested subterms are decomposed through chains of procedure invocations. Five declarations for Waltz state the seven explicit paths needed to uncover all 28 unknown paths:

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

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

Table 2.4 categorizes the paths by type. User paths are paths defining variables in user-defined procedures (c.f. =/2 paths, which are less interesting, except for assignment and for propagating modes within the analysis). User input paths are further split into 1-paths (top-level formal parameters of procedures) and k-paths for $k > 1$. Assignments are =/2 output 1-paths.

By type, user paths constitute the largest portion with 52%. 15.9% are user output paths, and the user input paths split almost evenly between paths of length one and greater. The length of the k-paths is highly program dependent, although usually the average length is close to two. Assignments constitute 10.4% of all paths. Surprisingly, a large percentage (38%) of the paths are builtin paths needed only for mode propagation during analysis.

Table 2.5 gives the execution times of the analysis on a Sparcstation II under PDSS. Unfortunately, this preliminary implementation suffers because the path-table access functions are linear in the number of generated paths. This in turn worsens the performance

Table 2.5: Execution Time for Analysis: Raw Seconds and (Percentages)

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

of most of the algorithms from linear to quadratic. We expect the implementation to be approximately linear on the number of paths after the re-implementation of our path table e.g., as a hash table. Furthermore, PDSS is an emulation-based system, and the timings include the full impact of frequent garbage collections.

However, these measurements do indicate the approximate *relative* weight of each phase of the analysis. We see that the multiway rule, although potentially exponential, is in practice quite cheap. Almost all of the computation (85%) arises from path generation and binary mode analysis. Complex programs show significant (11–20%) multiway analysis. The last column of the table estimates program complexity by the metric of paths analyzed per second. As explained above, the current performance is quadratic (i.e., paths^2/second is linear in Table 2.5), which in fact confirms that we can linearize the implementation.

2.6 Summary

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

Unfortunately, we have also seen that our analysis algorithm has serious problems of consistency, completeness, and safety, which limit its uses. We have seen that these problems are both difficult and fundamental, and that the circular-unification method of analysis suggested by Ueda and Morita may provide a solution to these problems.

Chapter 3

Sequentialization

3.1 Introduction

Consider the case where one has a program written in fully-moded *FGHC*, and thus knows the modes of all variable references from mode analysis. As noted in Chapter 1, one may now perform a number of traditional compiler optimizations on the code which sets and dereferences these variables.

However, these optimizations are typically not very effective at improving the performance of an *FGHC* program. The major source of overhead in traditional execution of *FGHC* programs is the fact that the execution order of body goals is determinable only by run-time examination of the data dependencies. Thus, a scheduler creates and manages a large number of extremely fine-grained processes which execute the program. The scheduler, process creation, and process suspension/resumption overhead are generally the major source of inefficiency in traditional *FGHC* implementations [2].

Thus, it is desirable to determine the execution order of body goals at compile time to the extent that this is possible, in order to increase the granularity of execution and reduce the the above-cited overheads.

The material of this chapter is strongly inspired by the work of King and Soper [17] on formal conditions for threading general *FGHC* programs, which was in turn inspired by work such as that of Tick and Korsloot [19] on sequentializing *FGHC* programs with mode analysis.

This chapter is organized as follows: The reasons for sequentializing an *FGHC* program are presented. The nature of sequentialization and the role of mode analysis is discussed. The problem of feedback in sequentialization is examined, and various possible solutions to this problem are suggested, including local analysis, global analysis, and language restriction. Our sequentialization algorithm is presented. Our implementation of this algorithm in a *FGHC* to C translator is presented and evaluated. Finally, the results of this chapter are summarized.

3.2 Goals of Sequentialization

We begin consideration of sequentialization of *FGHC* programs with a formal statement of the goal to be achieved.

> *Definition*: An *FGHC* program P has been *fully sequentialized* when the sequence of execution of the body goals $B_{i,j}$ of each clause C_i of P may be precisely fixed at compile time, in such a way that the resulting program P' will never

fail on any input bindings for which P could not fail, and will never produce different output bindings than P would produce on any input bindings. □

One needs to sequentialize only the body goals of individual clauses, because once all the body goals of each clause have been sequentialized, the execution order of the clauses is determinable merely by insisting that the body goals within clauses be executed in depth-first fashion [17].

While it would be nice if every *FGHC* program, or even every fully-moded *FGHC* program, was fully sequentializable as defined above, this is not the case. However, there are still useful sequentializations possible even in programs which are not fully sequentializable. This observation leads us to define two different limited forms of sequentialization which may allow us to exploit sequentialization in a more general way.

Definition: An *FGHC* program P has been *threaded* when each body goal $B_{i,j}$ of each clause C_i of P has been assigned to some "thread" t_k of C_i, and each thread has been fully sequentialized. □

Thus, each clause of the resulting program P' consists of some number N_t of threads executed in parallel, each of whose body goals is executed sequentially. This definition is due to King and Soper [17], who also give an algorithm for finding the minimum number of threads in a feedback-free (see below) *FGHC* program.

The other possibility is that we do not know that any two body goals in a clause are precisely sequentializable, but we can divide the body goals up into groups which execute in a sequential order.

Definition: An *FGHC* program P has been *grouped* when each body goal $B_{i,j}$ of each clause C_i of P has been assigned to some "group" g_k of C_i, and the order of execution of the groups has been been fully sequentialized. □

This turns out not to be as useful as threading for efficient execution, since it does not reduce the number of processes actually required to execute the program, although it may still be useful in improving scheduler performance.

Various combinations of grouping and threading can occur among the body goals as well. The combination of grouping and threading leads naturally to a situation in which the body goals are given a *partial order* at compile time. The various schemes for sequentialization of body goals may be characterized by the implicit partial orders they introduce.

3.3 Sequentialization and Mode Analysis

If one is trying to establish a partial order of body goals in an *FGHC* program, one has a distinct advantage if the program is fully-moded. In this case the relation being ordered over may be taken as the producer-consumer relation between body goals which has been completely fixed by mode analysis. This chapter will consider only this fully-moded case.

King and Soper [17] deal with the more complicated case in which the producer-consumer relation may not be determinable until runtime, and thus sequentialization may only be approximated at compile time if safety is to be preserved. We note that the results of

$$
\begin{array}{ll}
q(\ [\ !X \mid !Xs\ ]\ , Y, !Z) :\!\!- \mathit{true} \mid & \\
\qquad s(Xs, X, !L, !G), & \text{[B1]} \\
\qquad q(L, !Y, [\ X \mid Vs\ ]\ ), & \text{[B2]} \\
\qquad q(G, !Vs, Z). & \text{[B3]}
\end{array}
$$

Figure 3.1: Fully-Moded Clause Of Quicksort

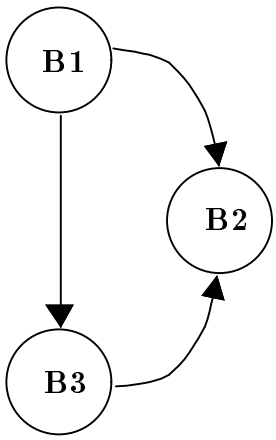


Figure 3.2: Dependency Graph Of Quicksort Clause

Chapter 2 indicate that the fully-moded restriction upon the language is not particularly onerous. We will discover in this chapter that it is *much* easier to sequentialize a program in the presence of complete and fixed mode information, an additional important reason to prefer fully-moded programs.

Offhand, it is difficult to construct a fully-moded *FGHC* program which is *not* fully sequentializable via the simple algorithm of constructing a DAG representing the partial order imposed by the producer-consumer relation and then topologically sorting it. Consider the example of Figure 3.1 (clause 2 of the Quicksort example discussed in Chapter 2), where exclamation marks have been placed before each variable reference which is a producer of its value in the clause. This convention is similar to that of Janus [13], a language in which the producer and consumer of data must be explicitly identified. (Note that this convention is defined relative to the body of the clause, as is necessary for the analysis. This makes the clause head look somewhat "backward" from the programmer's point of view.) This naturally produces a dependency DAG (partial order) between clauses which is illustrated in Figure 3.2. From the DAG, we immediately discover a full sequentialization of the clause: body goal **B1** must execute first, followed by **B3** and then **B2**. Similar analysis may be applied to all clauses of the program, obtaining a fully sequential version of Quicksort.

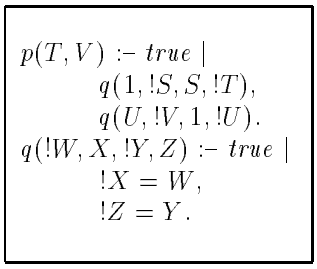


Figure 3.3: Fully-Moded *FGHC* Program Exhibiting Feedback

3.4 The Feedback Problem

It would seem, then, that any fully-moded *FGHC* program may be fully sequentialized. Indeed, this was the conclusion of Korsloot and Tick [19]. However, the recent work of King and Soper [17] points out the flaw in this analysis; there exist a class of fully-moded programs whose body-goal dependencies are cyclic, and thus not orderable by a topological sort, but which nonetheless describe executable *FGHC* computations.

Adopting the terminology of King and Soper, we refer to this problem as *feedback*, as it manifests itself in the program by the feeding of information obtained from the output of a body call back into that body call itself. This operation makes no sense in traditional languages. In languages with "logical variables," the binding of an output variable of a body call may also implicitly bind an input variable to the call, allowing the call to proceed by "examining its own output."

At this point, a concrete example will serve to illustrate the nature of the problem. Consider the moded *FGHC* program of Figure 3.3 with query $p(\_, \_)$. Several interesting points are raised by this example. The clause body of $q/4$ consists of a DAG with two disconnected components. Thus, it should be possible to execute the body goals in either order. However, the first body goal of $p/2$ requires the output of the first body goal of $q/4$ *as the input of* the second body goal of $q/4$. For the second body goal of $p/2$, the reverse is true! Thus, no static ordering of the clause body of $q/4$ allows the program to avoid deadlock on the given query, although the program would execute correctly in a concurrent implementation.

The basic problem in this example is that information from $q/4$ is fed back into $q/4$ at runtime, thus forcing concurrent execution. We formally define feedback, for our purposes, as follows:

> *Definition*: A fully-moded *FGHC* program contains *feedback* if, for some clause body of the program, the producer-consumer dependency graph for variables of that clause contains a cycle. □

Note that this definition allows for the possibility of what King and Soper call "indirect feedback" as well. Consider carefully Figure 3.4. This program clearly still enforces a cyclic dependency graph on $p/2$, although no variable occurs twice in the same body goal (i.e., there is no "direct feedback"). Clearly, the same sequentialization problem exists.

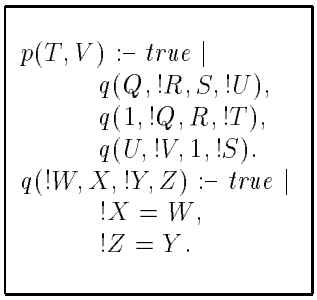


Figure 3.4: Indirect Feedback

3.5 Possible Approaches To Feedback

Since there exist legal fully-moded *FGHC* programs containing feedback, we must cope with this situation in some fashion. Three possibilities are detailed below, which represent various engineering tradeoffs between implementability, sequentializability, and ease of use.

3.5.1 Local Analysis

One possibility is to simply allow for all possible caller feedback patterns in sequentializing each clause body. Clearly, the program will not fully sequentialize in this fashion, but it is possible that it may be threaded or grouped to a large enough extent that significant optimizations may be attained.

One such threading strategy is *linear threading* — if a portion of the dependency graph is linear, that portion of the graph may be threaded in the linear order. Thus, the clause

$r(!A, D)$:- *true* |
$r_1(A, !B)$,
$r_2(B, !C)$,
$r_3(C, !D)$.

may always be safely sequentialized in the order r_1, r_2, r_3, regardless of the feedback behavior of the caller of $r/2$.

The clause $r/2$ above indicates another threading strategy — a clause's caller may exhibit feedback which affects the clause only if the clause has more than one input. In other words, feedback is a phenomenon which requires at least two inputs. Thus, we can fully sequentialize $r/2$ regardless of its body.

One might expect that it is possible, by applying enough observations like those above, to "mostly" thread most clauses of a fully-moded *FGHC* program. Unfortunately, this does not prove to be the case. A quick scan of a number of clauses from fully-moded *FGHC* benchmarks reveals very little local threading or grouping information available from these clauses, as evinced by constructing counterexamples to various proposed threadings and groupings.

Given the difficulty of formalizing the analysis, and the inferior results possible from this analysis in any case, local analysis does not currently appear to be a productive technique

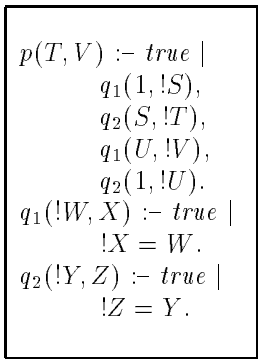


Figure 3.5: Global Feedback Elimination

for the threading of fully-moded *FGHC* programs.

3.5.2 Global Analysis

One might expect, however, that the situation is similar to that of the mode analysis itself — given complete global information about a fully-moded *FGHC* program and its queries, one could obtain the maximum possible threading of the program as a whole.

Indeed, this does seem to be the case in practice. However, formulating a consistent system for automatically deriving this information has proven to be quite difficult. It is conceivable that dataflow analysis [1] could be used, as follows:

Consider a schema in which the basic blocks for dataflow analysis are clauses, and the block state variables are sets of clause inputs which are "cotemporal" in the sense that they may all be "supplied at the same time." To make this notion of cotemporality precise, consider a clause containing feedback, and consider the inputs to body goals of that clause.

> *Definition*: A set of inputs is *cotemporal* iff whenever any input in the set is available, all inputs in the set are available. □

Thus, in our feedback example of Figure 3.3, we initially assume that W and Y are cotemporal (available simultaneously in the head of $q/4$). However, analyzing $p/2$ reveals that Y must be available in $q/4$ strictly later than X, so the inputs to $q/4$ are split into two cotemporal sets. This splits the body goals of $q/4$ into two groups — those dependent on W, and those dependent on Y. This information might be used to create new clauses q_1 and q_2, so that the program could be rewritten as in Figure 3.5 and then sequentialized. While this example gives some insight into global feedback elimination, any actual algorithm would have to handle a number of special cases more complicated than those just described. While this is probably the general way in which a human would deal with *FGHC* programs exhibiting feedback, this approach appears quite complex to attempt automatically.

In addition to the complexity just described, note that the reason for the strong threadedness of practical programs is that feedback does not seem to occur much in real examples — it is unusual to write a fully-moded *FGHC* program exhibiting feedback. We have encountered only two situation in our work in which feedback occurred, and only one of them

is interesting (the other is that some primitives built into an *FGHC* system we are using encourage the use of feedback).

Consider the situation in which one is generating "difference lists" of data, and appending them together. It is natural to append the difference lists in the reverse of the order in which they are generated, so that the resulting difference list will have its elements in the right order. If one also has to thread state information through the generators in the forward direction, one naturally obtains feedback. It is straightforward to code around this problem by not using difference lists, but at some efficiency penalty.

Given that dealing with feedback through global analysis is difficult, and that feedback does not often occur in real programs, it seems profitable to consider a third strategy.

3.5.3 Language Restriction

With feedback defined as above, it becomes easy to *test* for the presence of feedback in a fully-moded *FGHC* program. Further, the concept of feedback in fully-moded *FGHC* programs is easily explained to *FGHC* programmers. Thus perhaps the most promising approach is to simply restrict consideration to the class of feedback-free fully-moded *FGHC* programs. This language restriction, as noted above, has the pleasant property that one may fully sequentialize any feedback-free fully-moded *FGHC* program! Further, the language restriction is easily explained to programmers, easily checked by the compiler, and does not seem to affect many real programs — a language designer's dream condition!

3.6 Sequentialization Algorithm

Having chosen to restrict attention to feedback-free fully-moded *FGHC* programs, the next step is to specify the algorithm for:

1. Detecting feedback in the program.
2. Fully sequentializing the feedback-free program.

Fortunately, both of these goals may be accomplished by a single algorithm: a topological sort with cycle detection [23] as in Figure 3.6.

This algorithm performs the appropriate topological sort of each clause inline. We say that a body goal is a "sink" if all the inputs of that body goal are produced only by

- The head of the clause in which the body goal appears.
- Already-sequentialized body goals of the clause.

Thus, by repeatedly finding sinks in the clause, we eventually topologically sort the entire clause.

This algorithm has the drawback of $O(n^2)$ worst-case running time. An algorithm exists which is $O(n)$ [24], but it is quite complex to program. Since the complexity is reall only a function of the number of body goals in a clause, $O(n^2)$ complexity is considered acceptable. The algorithm is easy to understand, and has the advantage that the typical-case

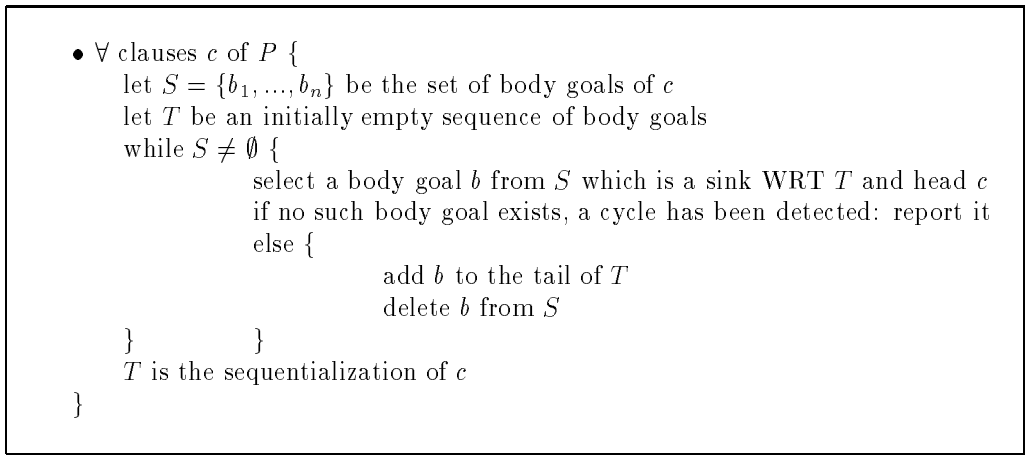


Figure 3.6: Sequentialization Algorithm

performance may be nearly linear: if the body goals are considered in the reverse order of appearance, the typical *FGHC* programming style will result in a linear-time sequentialization of the clause body.

A remaining question is that of the use of the resulting full sequentialization. There are many possible sources of improved performance for a fully-sequentialized version of an *FGHC* program. As discussed in the introduction to this chapter, on a uniprocessor, or even on a machine with a small number of processors, the largest gains are likely to come when the traditional work-queue based method of processing body goals [28] is replaced by the stack-based depth-first procedure call method of traditional imperative and functional languages [1]. Other uses of sequentialization in compilation include those discussed in [30, 19, 17], such as elimination of redundant tests, and creation of local ("extra-logical") variables.

3.7 Implementation and Evaluation

The algorithm of Figure 3.6 has been implemented in *FGHC* running under the PDSS system [5], as part of a system for automatically transforming *FGHC* programs to sequential C code. This system, comprising about 1500 lines containing 183 clauses of *FGHC* code, accepts *FGHC* programs in an intermediate form produced by the front-end transformer of the Monaco shared-memory multiprocessor *FGHC* system, and produces executable C code. The Monaco front-end takes care of producing decision graphs and combining clauses into a single procedure, but leaves clause bodies essentially unaffected. Our sequentializer accepts Monaco front-end output which has been annotated with the mode information automatically generated by our mode analyzer, and uses this information to generate C code of reasonably high quality. Our observation has been that for normal-sized clauses, the sequentialization time is so small as to be completely swamped by the overhead of code generation.

Our sequentializer is perhaps best understood in the context of a simple example. When

the procedure $append/3$

$append([], X, Y) :- true \mid X = Y.$

$append([A|B], X, Y) :- true \mid Y = [A|Z], append(B, X, Z).$

is transformed by our system into sequential C code and then compiled, we obtain the code of Appendix A. All in all, our produced code is very good; our inner loop of $append/3$ compiles to about 39 machine instructions, a respectable number.

Currently, the support code and macros are very simple — for example, there is no garbage collector, and a machine word per object is wasted on tag information. Nonetheless, since the system produces reasonably good C code using a fairly modular generator, it will be easy to remove limitations like these in the future. Self-tail-call and other tail-call optimization (not possible in this example) is not done in the source program, although these sorts of optimization may be performed by the very best C compilers. This will be very difficult to fix, because of the restrictions of the C language.

FGHC procedures processed by our sequentializer may have several output parameters. It thus would be difficult and expensive in the general case to return all the output parameters in the C function result — the parameters would have to be encapsulated in a single structure to be returned. We chose to implement multiple output parameters by passing output parameters "by reference," using C pointer expressions. Unfortunately, the potential aliasing this introduces inhibits the C compiler from performing some optimizations. By way of comparison, an older version of the code generator produced C code which passed input parameters by reference as well, but the combination of more potential aliasing and extra dereferences made it significantly slower on our benchmarks.

The execution of the qsort *FGHC* program of Chapter 2, was measured under various conditions on several systems. The results are displayed in Table 3.1. All measurements were made on a 20-processor Sequent Symmetry running DYNIX V3.2.0. PDSS is the high-quality interpreted *FGHC* system under which the code described in this thesis runs; the PDSS system emulates parallel scheduling and execution order on a single processor. Monaco [11] is a research compiler which produces native code executables capable of utilizing an arbitrary number of the processors of the Symmetry in a shared-memory multitasking fashion. The sequential ANSI C code generated by our sequentializer was compiled using GCC 2.2.2 [27], at optimization level 2 but with no other special optimizations. Strand [12] and JAM Parlog [6] are other emulator-based compilers for *FGHC* languages — the benchmark was slightly different under these implementations to accommodate minor language differences. The handcrafted C code was written to be as close as possible to the qsort benchmark algorithm while retaining a natural C style — in particular, parameters were passed with appropriate types whenever possible. However, a more traditional array-based Quicksort [18] would be faster.

The benchmark consisted of generating a list of numbers in forward order, and then using the Quicksort algorithm to sort these numbers into reverse order. This is a worst-case input to Quicksort, and leads to an $O(n^2)$ expected running time. An inspection of the data indicates that this time complexity was in fact achieved for all systems tested.

As expected, the interpreted PDSS system was much slower than the other systems tested, although the performance was quite impressive for an interpreted implementation.

Table 3.1: Performance of Sequentialized and Parallel qsort

implementation	processors	problem size	time
PDSS	1	125	1.4s
		250	5.2s
		500	20.5s
Monaco	1	500	10.5s
	4	125	0.20s
		250	0.75s
		500	2.9s
	8	500	1.5s
sequential C	1	125	0.14s
		250	0.58s
		500	2.2s
Strand	1	512	10.8s
JAM Parlog	1	512	13.1s
handcrafted C	1	125	0.10s
		250	0.40s
		500	1.5s

The sequential C code seems to be about four times faster than the parallel code on the benchmark, which accords quite well with what is known of the internals of the Monaco system. It is conceivable that the Monaco implementation could be made about twice as fast by better compiler optimization and more careful runtime tuning. As implied by the performance of the handcrafted C code, it is likely that the C code could be made twice as fast by source optimization, especially considering that neither C program contains the obvious self-tail-call optimization (which would require a slight modification to the sequentialization algorithm so that it would try to "save a call to self for last"). That the the automatically generated C code is only about 50 percent slower than the handcrafted C code shows that a lot can be done with even a simple automatic code generator!

3.8 Comparison With Other Work

Our work is perhaps most comparable with that of Chikiyama, who has constructed a translator from *FGHC* to C producing code which is concurrently scheduled on a uniprocessor. In [4], Chikiyama describes and measures his inner loop for an *append*/3 benchmark similar to ours.

As mentioned on page 31, our inner loop of *append*/3 compiles to about 39 machine instructions. Four of these machine instructions are involved with setting and restoring the frame pointer, and may be removed by GCC. Of the remaining 35 instructions, eight of them are devoted to procedure call and return. This yields an inner loop of approximately 27 instructions, which compares well with the 24 instruction loop of Chikiyama. This is an especially pleasant result since the eight instructions of the procedure prolog and epilog constitute the full overhead of scheduling and invocation in the program! It would be very difficult to achieve this sort of low-overhead scheduling and invocation in a concurrent

implementation.

This system is superficially similar to the *jc* compiler [13, 14] which translates Janus to C code. However, *jc* uses a traditional concurrent logic programming execution model, which leads to a number of problems. The C code produced for the append benchmark is rather opaque, and apparently no faster than the code produced by our system. Furthermore, more complicated benchmarks may suffer significantly from the extra suspensions and reductions introduced by the *jc* compilation scheme.

As presented, our system is only able to sequentialize entire programs. This is a larger granularity than is desired for most real applications. It should be possible to fully-mode and sequentialize modules, and then call these modules from concurrent *FGHC*. To accomplish this, our translator would insert code in the sequentialized module which would check each runtime variable dereference to be sure the variable was bound, and suspend otherwise. (Unfortunately, suspension would be difficult, since one would like to avoid preserving the entire C stack across suspensions. It would probably be best to use the "thread" support available in most modern operating systems for this.) A module stub would invoke the sequentialized module, wait for it to complete, and then awaken any suspended invocations which were hooked to output variables of the module. This mechanism would typically add about two extra instructions to each input variable reference of the sequentialized module, but would allow a mixture of sequentialized modules with concurrently scheduled ones in a real system.

3.9 Summary

The sequentialization work reported here illustrates the practicality of sequentialization as an optimization technique for *FGHC*, given some innocuous language restrictions. We have discussed what it means to sequentialize a program, taken a look at some fundamental problems in sequentialization of *FGHC* programs, and noted alternative analyses. We have suggested a language restriction which makes complete sequentialization straightforward, and have given an algorithm which will fully sequentialize a fully-moded feedback-free *FGHC* program. Finally, we have discussed our implementation of this algorithm as part of a translator from *FGHC* to sequential C, and examined the efficiency gain resulting from this translation.

Overall, we have seen that sequentialization of *FGHC* programs is both possible and beneficial given some language restrictions and appropriate program analysis, and that this technique is a promising one for *FGHC* execution.

Chapter 4

Conclusions And Future Work

4.1 Conclusions

It is common practice in programming language implementation to apply static analysis to computer programs at compile time, and to use the information derived from this analysis in producing optimized compiled output. In our case, the analysis consists of mode analysis, which describes the dataflow of an *FGHC* program, and sequentialization, which uses the mode information to produce compiled code optimized for execution on a uniprocessor.

Our mode analysis algorithm is both easily implementable and fairly efficient, and is suitable for automatically obtaining the modes of most simple *FGHC* programs. However, its principal defect is that it is unsafe, which is a result of the fact that it is incomplete. A complete mode analysis algorithm, such as that proposed by Ueda and Morita, would provide an axiomatic definition of the language "fully-moded *FGHC*," a language which is a strict subset of *FGHC*, but with about the same expressive power.

Our sequentialization algorithm is a simple topological sort with cycle detection. Nonetheless, this algorithm is adequate to sequentialize most fully-moded *FGHC* programs, and provides an axiomatic definition of the language "feedback-free fully-moded *FGHC*," a language which is a strict subset of fully-moded *FGHC*, but with about the same expressive power.

This thesis has shown that it is practical to write programs in feedback-free fully-moded *FGHC*, and that these programs may be efficiently compiled into high-quality native code for uniprocessors. This is particularly important in the current environment, in which the cost/performance ratio of uniprocessor computers is dropping so rapidly that it is often impractical to try to exceed the performance of a uniprocessor system with a similarly priced multiprocessor system; by the time the multiprocessor system is designed and implemented, the uniprocessor systems in the same price range have far exceeded the maximum theoretical performance of the multiprocessor. This is especially true for medium-grain MIMD machines, as these tend to make heavy use of off-the-shelf components, and are thus locked to a lagging point on the price/performance curve of the uniprocessor machines.

One possible way out of this dilemma is to write programs in languages which allow programming without reference to the number of processors in the underlying architecture, but languages which nonetheless allow efficient parallel execution on a multiprocessor system. In this way, programs can be run on the current cheap, fast uniprocessors, and then be run without change on parallel machines when the price differential changes or when high performance is desired regardless of cost. The *FGHC* family of committed-choice logic programming languages is of precisely this sort.

However, most past implementations of *FGHC* have been hampered by very high execution overheads, which has made *FGHC* unattractive in comparison with more traditional

languages for use in uniprocessor environments. Our work has shown that these overheads are not an inherent feature of all *FGHC*-based languages, and that committed-choice logic-programming languages are thus a reasonable vehicle for programming in uniprocessor environments.

We also expect that we will be able to produce sequentialized versions of fully-moded feedback-free *FGHC* modules which are callable by more traditional runtime-scheduled *FGHC* code. This would allow the granularity of *FGHC* programs to be increased, while still retaining the concurrent execution capabilities of the language. Such code might be a very attractive alternative to fine-grained *FGHC* and to traditional non-concurrent languages in coarse-grained parallel processing environments such as that of the Sequent Symmetry.

4.2 Future Work

Much work remains to be done. The whole problem of fully-moding *FGHC* programs in a complete and safe way has yet to be solved. The moding proposal of Ueda and Morita should be directly implemented, in order to understand the practical aspects of their technique. This implementation is made difficult by the incomplete explanations of their technique in the literature, but is important enough that it should be attempted nonetheless.

The more general case of sequentialization of incompletely-moded *FGHC* programs in the presence of feedback has been thoroughly studied by King and Soper [17]. While their work shows great promise in solving this more general problem, the inability of the techniques proposed therein to fully sequentialize many *FGHC* programs makes it somewhat difficult to exploit the sequentialization that is obtained. Nonetheless, their analysis is clearly an important step in improving the performance of existing *FGHC* code.

The strong connection between fully-moded feedback-free *FGHC* and strict functional languages needs to be further explored. It is entirely likely that this restricted form of *FGHC* is isomorphic to some some strict functional language, in which case one could explore both the relationship in compilation techniques between the two languages and the possibility of source-to-source translation of fully-moded feedback-free *FGHC* into one of these languages.

The analysis and compilation of concurrent logic programming languages has come a long way since the introduction of these languages just a few years ago. This thesis has attempted to advance the state of this art just a little bit further. In this, the author believes and hopes he has succeeded.

Appendix A

C Code Generated For append Benchmark

```
#include "fghc.h"

extern void proc_append_3( object_t *p_param_0,
        object_t *p_param_1, object_t **p_param_2 );


void proc_append_3( object_t *p_param_0, object_t *p_param_1,
        object_t **p_param_2 ) {
/*
        pushl %ebp              ; allocate stack frame --
        movl %esp,%ebp          ; gcc can omit
        subl $4,%esp
        pushl %edi              ; callee saves
        pushl %esi
        pushl %ebx
*/
        object_t *p_var_4;
        object_t *p_var_3;
        object_t *p_au_tmp_1;
        object_t *p_au_tmp_0;
        object_t *p_local_4;

/*
        movl 8(%ebp),%edx       ; param 0 in edx
        movl 12(%ebp),%ecx      ; param 1 in ecx
        movl 16(%ebp),%esi      ; param 2 in esi
        movl (%edx),%eax        ; get param 0 type
        cmpl $34,%eax           ; nil case (should be 2nd)
        je L34
        cmpl $48,%eax           ; pair case
        je L35
        jmp L41
*/
        switch( p_param_0->type ) {
        case NIL:
                /* clause 1 */
/*
```

```
L34:
                movl %ecx,(%esi)        ; save result
 */
                (*p_param_2) = p_param_1;
/*
                jmp L32                 ; and return
 */
                return;
                abort();
        case PAIR:
/*
L35:
                movl 4(%edx),%ebx       ; car of param 0 in ebx
 */
                p_var_4 = p_param_0->value.pair.left; /* XXX */
                p_var_3 = p_param_0->value.pair.right; /* XXX */
                /* clause 2 */
/*
                leal -4(%ebp),%eax
                pushl %eax              ; address of local 4
                pushl %ecx              ; param 1
                pushl 8(%edx)           ; cdr of param 0
                call _proc_append_3
                addl $12,%esp           ; pop args
 */
                proc_append_3( p_var_3, p_param_1, (&p_local_4) );
/*
                movl -4(%ebp),%edx      ; local 4 in edx
 */
                p_au_tmp_1 = p_local_4;
                p_au_tmp_0 = p_var_4;
/*
                addl $12,_curheap       ; allocate a pair in eax
                movl _endheap,%edi      ; destroying edi
                cmpl %edi,_curheap
                jb L38
                call _abort             ; should gc here
        L38:
                movl _curheap,%eax
                movl $48,(%eax)         ; type is pair
                movl %ebx,4(%eax)       ; car is car param 0
                movl %edx,8(%eax)       ; cdr is local 4
                movl %eax,(%esi)        ; save in param 2
*/
```

```
                (*p_param_2) = make_pair( p_au_tmp_0,p_au_tmp_1 );
/*
                jmp L32                 ; return
 */
                return;
                abort();
        default:
p_label_2:
                goto p_label_1;
                abort();
        }
p_label_1:
/*
L41:
        call _abort
 */
        abort(); /* suspend */
/*
L32:
        leal -16(%ebp),%esp     ; undo frame
        popl %ebx               ; restore regs
        popl %esi
        popl %edi
        leave                   ; get out
        ret
 */
}
```

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading MA, 2nd edition, 1985.

[2] L. Alkalaj, T. Lang, and M. Ercegovac. Architectural Support for Goal Management in Flat Concurrent Prolog. *IEEE Transactions on Computers*, 25(8):34–47, August 1992.

[3] 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.

[4] T. Chikayama. A Portable and Reasonably Efficient Implementation of KL1. ICOT, Tokyo. unpublished draft, June 1992.

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

[6] K. L. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[7] J. Cohen. A View Of The Origins And Development Of Prolog. *Communications of the ACM*, 31(1):26–36, January 1988.

[8] J. S. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Norwell MA, 1987.

[9] 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.

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

[11] S. Duvvuru. Monaco: A High Performance Implementation Of FGHC on Shared-Memory Multiprocessors. Technical Report CIS–TR–92-16, Department of Computer and Information Sciences, University Of Oregon, July 1992.

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

[13] D. Gudeman, K. De Bosschere, and S. K. Debray. `jc`: An Efficient and Portable Sequential Implementation of Janus. In *Joint International Conference and Symposium on Logic Programming.* Washington D.C., MIT Press, November 1992.

[14] David Gudeman and Sandra Miller. *User Manual for jc — A Janus Compiler (version 1.56).* University of Arizona, July 1992.

[15] A. King and P. Soper. Granularity Control for Concurrent Logic Programs. In *International Computer Conference*, Turkey, 1990.

[16] A. King and P. Soper. Partitioning and Scheduling Concurrent Logic Programs. In *Workshop on Future Directions of Parallel Programming and Architecture, International Conference on Fifth Generation Computer Systems*, Tokyo, June 1992. ICOT.

[17] A. King and P. Soper. Schedule Analysis: A Full Theory, A Pilot Implementation, And A Preliminary Assessment. Technical Report CSTR 92–06, Department of Electronics and Computer Science, University Of Southampton, February 1992.

[18] D. E. Knuth. *The Art of Computer Programming: Searching and Sorting.* Addison-Wesley, Reading MA, 2nd edition, 1973.

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

[20] R. Kowalski. The Early Years Of Logic Programming. *Communications of the ACM*, 31(1):38–43, January 1988.

[21] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

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

[23] K. A. Ross and C. R. B. Wright. *Discrete Mathematics.* Prentice-Hall, Englewood Cliffs NJ, 2nd edition, 1988.

[24] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading MA, 1st edition, 1983.

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

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

[27] R. M. Stallman. *Using and Porting GNU CC (version 2.0).* Boston, MA, November 1990.

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

[29] K. Ueda. *Guarded Horn Clauses.* PhD thesis, University of Tokyo, March 1986.

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

[31] K. Ueda and M. Morita. Moded Flat GHC And Its Message-Oriented Implementation Technique. *New Generation Computing*, 1992. To Appear.