Modes of Comprehension: Mode Analysis of Arrays and Array Comprehensions

B. Massey and E. Tick

CIS-TR-94-10a
November 1994

Abstract

A scheme is presented to enable the mode analysis of concurrent logic programs manipulating arrays containing both ground and non-ground elements. To do this we leverage constraint-propagation mode analysis techniques. The key ideas are to restrict multiple assignments only to variables at the *leaves of paths*, and to extend the language family with *memo comprehensions*. The result is a language not significantly different than generic committed-choice languages, which can be *safely* mode analyzed, producing useful (not overly conservative) information, even for programs that assign to unbound array elements. An implementation of the scheme is presented.

This paper submitted to *International Conference on Logic Programming*, Tokyo, June 1995.

Department of Computer and Information Science
University of Oregon

Contents

"Given for one instant an intelligence which could comprehend all the forces by which nature is animated and the respective positions of the beings which compose it, if moreover this intelligence were vast enough to submit these data to analysis...to it nothing would be uncertain..."
Pierre Simon de Laplace
Oeuvres, vol VII, Theorie Analytique des Probabilites

1 Introduction

Modes in logic programs are a restricted form of types that specify whether a logic variable (or some subterm thereof) is produced or consumed by a procedure. For example, in a pure logic program, a well-known feature of the list concatenation procedure are its two permissible modes: either two input lists are consumed, producing their concatenation, or a concatenated list is consumed, producing pairs of output lists (that if concatenated equal the input list). For such *nondeterminate* logic programs, there are numerous methods for statically determining modes [1, 2, 3, 9, 13] and several optimizations afforded by the mode information, notably converting backtracking programs into functional programs, and specializing unification operators into matches and assignments, and thereby making them execute faster.

Mode information has also been shown to be quite useful in the efficient compilation of *indeterminate* logic programming programs, i.e., committed-choice programs [5, 6, 14, 18, 21]. Mode information allows demand-driven execution [7, 20] and static partitioning of a concurrent logic program into threads of higher granularity for more efficient multiprocessor execution [4, 6]. Mode information is useful not only for compiler optimization but also for static bug detection.

All the previous schemes and implementations (ours included) deal with the problem of aliasing through array indices, which is a well-known stumbling block in many types of static analysis, by restricting array elements to ground terms. Allowing arbitrary expressions for both array elements and indices makes determining aliases undecidable; the restriction of array elements to ground terms makes safe mode analysis possible. In other words: use writable array elements at your own risk! The problem is more pervasive than it might sound offhand — a program with even a single use of non-ground arrays cannot be safely analyzed with currently available technologies.

This papers describes a novel concurrent logic programming language extension and corresponding mode analysis extension that allow safe static mode analysis that is not overly conservative for programs that use arrays with unbound elements. The objective of the paper is to describe the problem and our solution, giving examples of the technique. The method has been implemented as an extension to a previous mode analyzer that we have built [14, 17].

This paper is organized as follows. Section 2 defines the array moding problem and gives a series of increasingly flexible solutions. Section 3 discusses memo comprehensions, a technique borrowed from functional languages, which allow constraint-based mode analysis to be exploited. Section 4 gives some examples of the techniques described. Sections 5 and 6 formalize the extensions to the constraint-based mode analysis technique and describe an implementation. Conclusions and future work are summarized in Section 7. Appendix A

illustrates the automated operation of our mode analyzer upon the examples in this paper. Appendix B gives a formal version of the mode analysis rules, and shows how the memo rules fit into this framework.

2 Memos and Constraint-Based Mode Analysis

We will refer to our arrays for flat committed-choice languages as *memos*. We first describe the standard functional-style operations upon memos:[1]

- A memo is created by
  `memo_ini(N,M)`, where `N` is the number of elements, and `M` is the resulting memo. `M` is initialized to contain unbound variables (in contrast to KL1).
- A memo element may be referenced by
  `memo_sel(M,I,E)`, where `M` is the memo, `I` is the element index, and `E` is the $\texttt{I}^{th}$ element of `M`.
- A memo may be updated by
  `memo_upd(M,I,E,M')`, where `M` is the memo, `I` the index, `E` the new element, and `M'` is a copy of `M` except that the $\texttt{I}^{th}$ element of `M` is replaced by `E` in `M'`.

Before we discuss the semantics of these operations, let us review the mode analysis technique we plan to exploit. Ueda and Morita [21] proposed a mode analysis scheme based on the representation of procedure paths and their relationships as rooted graphs ("rational trees"). Unification over rational trees combines the mode information obtainable from the various procedures. For example, in a procedure that manipulates a list data stream, we might know that the mode of the *car* of the list (that is the current message) is the same mode as the *cadr* (second message), *caddr* (third message), etc. This potentially infinite set of "paths" is represented as a regular graph. Furthermore, a caller of this procedure may constrain the *car* to be input mode. By unifying the caller and callee path graphs, modes can be propagated. The analysis is restricted to "moded" flat committed-choice logic programs. These are programs in which the mode of each path in a program is constant, rather than a function of the occurrences of the path. As a consequence of this restriction, such programs can have only a single producer of any variable (although multiple consumers are allowed). These are not regarded as major restrictions, since most non-moded flat committed-choice logic programs may be transformed (by the programmer) to moded form in a straightforward fashion [21].

Ueda and Morita define the subterm s "derived" via a *path* p within a term t (written $p(t) \vdash s$) as follows: p derives s in t 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.

[1] We adopt the names from Sastry *et al.* [11], although similar builtins can be found in languages such as KL1, Strand, Parlog, etc. Many languages, such as KL1, instead provide a `memo_sel_upd/5` primitive which acts like a combination of `memo_sel/3` and `memo_upd/4`.

f is referred to as the "principal functor" of p. A program is "moded" if the modes of all possible paths in the program are consistent, where each path may have one of two modes: **in** or **out**. For example the *cadr* of the first argument of procedure q has input mode. We represent this using the notation $m(\{<q,1>, <,2>, <,1>\}) = \textbf{in}$. The notation m/p is used to represent the modes of all subtrees emanating from ("below") p.

The scheme described by Ueda and Morita has been implemented and tested in various incarnations [5, 14, 17, 18] so we want to leverage this investment to solve the array aliasing/moding problem. We will go into further details of the mode analysis in Section 5, but for now, let us discuss language restrictions.

To illustrate the sort of aliasing problems that can arise in the mode analysis of programs containing arrays, consider the procedure:

```
bad :-
  memo_ini( 1, M ),
  memo_sel( M, 0, E1 ),
  memo_sel( M, 0, E2 ),
  E1 = 3,
  E2 = 4.
```

The second assignment will cause mode analysis to fail, i.e., the procedure will be determined to be "nonmoded." The key point is that static detection of aliases is undecidable if we allow *arbitrary expressions* as indices, unless we restrict the language somehow.

One language restriction which solves the aliasing problem is to keep all array elements fully ground. Mode-analysis savants are aware of this solution, though it has never been formalized or implemented. For example, KL1 initializes array elements with integers, and if the programmer ensures they stay ground, then mode analysis could legitimately reject programs which make assignments to any element. This restriction actually covers quite a number of useful programs; however, our goal is to relieve the programmer of any burden and to cover the more general case of arrays with non-ground terms.

Our solution is to *forbid multiple assignments to an element of a memo along any given path "below" that memo.* In the previously example, the memo `M` is a flat array of a single element, so it is trivially disallowed. Consider the procedure:

```
good :-
  memo_ini( 1, M ),
  memo_sel( M, 0, E1 ),
  memo_sel( M, 0, E2 ),
  E1 = f( A, B ),
  bind( E2 ).

bind( f( A, B ) ) :- A = 1, B = 2.
```

The three assignments bind paths in `memo_sel/3`:

```
{<memo_sel,1>, <*>}
{<memo_sel,1>, <*>, <f,1>}
{<memo_sel,1>, <*>, <f,2>}
```

Here <* > is a placeholder representing the dereference of the memo itself.[2] The key point is that each of the three assignments lands at a unique leaf in the tree of paths, and thus mode analysis should deduce that the program is fully moded, which it is. To make this happen, we extend the mode analysis rules to include a simple reduction rule for `memo_sel/3`: let $p = \{$<`memo_sel`,1>, <*>$\}$, and $q = \{$<`memo_sel`,3>$\}$, then $m/p \neq m/q$.[3] In other words, ignoring the index, we alias all array elements of a given memo together. The details of this and other extensions are discussed in Section 5.

However, this scheme will disallow the following moded procedure:

```
bad :-
  memo_ini( 2, M ),
  memo_sel( M, 0, E1 ),
  memo_sel( M, 1, E2 ),
  E1 = 3,
  E2 = 4.
```

This is because we collapse indices and treat `E1` and `E2` as aliases. Thus the mode analysis is overly-conservative, and reports a mode conflict. This example may seem tolerable if we can still assign to unique paths, but actually this simple example is representative of a very serious impracticality: there is no practical way to assign a top-level value to more than one element of a multi-element array! The top-level indices are collapsed (as are all similar internal functors). So what we have is a safe, but far-too-conservative scheme.

We need some coherent way to *override* the restrictions we placed on assignment, allowing us to initialize whole arrays in some fashion digestible by the mode analyzer. Essentially we need to ensure that the indices used to bind a given path are mutually exclusive. This is one purpose of "array comprehensions" in functional languages, e.g., [10]. An array comprehension typically is something like (fictional syntax):

```
int A[20] {0,1: A[i] = 1;
  2..19: A[i] = A[i-2] + A[i-1]}
```

This example would create an array of 20 Fibonacci numbers. In the next section we expand upon the concept of memo comprehensions and show how it meshes with the mode analysis framework previously given.

3 Memo Comprehensions

We extend our language to support "memo comprehensions" (MC): *procedures designed to concurrently bind given paths in a given memo at more than one index.* Since the indices are guaranteed to be disjoint, if any binding is guaranteed to be safe, all are. Our MC syntax will be identical to that of normal procedures, except for a special guard `memo_csel/4`. Presence of such a guard in a procedure indicates that the procedure is a MC.

[2]There is nothing corresponding to < * > in Ueda and Morita's path definitions: its lack of a name reminds us that it represents an array or vector and its lack of an index reminds us that all indices are collapsed into one.

[3]The inequality here may look odd. It derives from the fact that tell unification argument must have opposite modes: What comes in from one argument must go out on the other argument. The same argument holds for the memo `M` and its element `E`.

The semantics are those of concurrent invocation: when a MC is invoked on a memo M, multiple invocations of the procedure are spawned, each with a unique index I of M.[4] A formal semantics is given in Section 6. Informally:

- `memo_csel(MO,M,I,E)` is the aliasing MC guard where MO is the memo, M is a read-only copy of MO, I is the element index, and E is the I^{th} element of M. The memo is input and the latter two arguments are produced by the guard for consumption in the body of the MC. The memo MO may be further instantiated only by binding E: performing a `memo_sel/3` on M in the MC body produces a read-only alias into M.

- A procedure containing clauses with `memo_csel/4` guards is a *memo comprehension.* A memo comprehension is automatically invoked multiple times per call, one invocation being produced for each unique element of the memo argument given to `memo_csel/4`. If an invocation matches more than one clause in the MC, a clause will be chosen nondeterministically in the usual fashion.

  Four semantic restrictions on the use of MCs are designed to make compilation and analysis easier and reduce the likelihood of programmer errors. First, all arguments of a MC, other than the memo, must be input. Second, only one `memo_csel/4` guard is allowed per clause. Third, every clause of a MC must have a `memo_csel/4` guard. Fourth, the `memo_csel/4` guard for each clause of a given MC must have the "same" first argument, i.e., if some clause of a procedure g has a MC guard for a memo at some argument path p, the MC guards of all clauses of g must be for the memo argument at p.

The first restriction can be checked during mode analysis and all others can be checked syntactically. The third restriction is meant to reduce programmer error: Without a `memo_csel/4` guard, clause arguments are read-only, and thus the clause cannot affect the outcome of the program.

Thus memo comprehensions allow a variable at a given path within (or "below") a memo to be bound *at more than one index, as long as the indices are part of the same MC.* This is the fundamental contribution of this paper. Extension of the mode analysis system to handle memos is reasonably straightforward, as discussed in Section 5.

4 Some Examples

As our first example, consider a hashed symbol table, sketched in Figure 1 (auxiliary procedures `hash/3` and `sym_match/3` are not supplied). Symbol table entries are normally fully ground and thus can be manipulated using memo functions only. Therefore, the initialization procedure `init_symtab/2` uses the only memo comprehension of the program. Given a memo consisting of unbound variables, `init_symtab/2` calls `null_table/1` to bind them all to nil. Procedure `null_table/1` can be reused for arbitrary-sized tables, and does not need

[4]Note that this is just semantics: there are a number of possible ways in which the concurrency might be limited in an implementation, for efficiency reasons — this is a topic of future research. We are unaware of any solution to date of the analogous efficiency problem in non-strict functional language implementations of array comprehensions.

```
init_symtab( NBuckets, SymTab ) :-
  memo_ini( NBuckets, Table ),
  null_table( Table ),
  SymTab = symtab( NBuckets, Table ).

null_table( Table ) :-
  memo_csel( Table, _, _, E ) |
  E = [].

insert_sym( symtab( NBuckets, Table ), Sym, SymTab_p ) :-
  hash( Sym, NBuckets, Hash ),
  memo_sel( Table, Hash, Chain ),
  Chain_p = [ Sym | Chain ],
  memo_upd( Table, Hash, Chain_p, Table_p ),
  SymTab_p = symtab( NBuckets, Table_p ).

lookup_sym( symtab( NBuckets, Table ), Name, Sym ) :-
  hash( Sym, NBuckets, Hash ),
  memo_sel( Table, Hash, Chain ),
  follow_chain( Chain, Name, Sym ).

follow_chain( [], _, Sym ) :-
  Sym = nomatch.
follow_chain( [ Sym | Syms ], Name, Sym_p ) :-
  sym_match( Sym, Name, B ),
  follow_chain_1( B, [ Sym | Syms ], Name, Sym_p ).

follow_chain_1( true, [ Sym | _ ], _, Sym_p ) :-
  Sym_p = Sym.
follow_chain_1( false, [ _ | Chain ], Name, Sym_p ) :-
  follow_chain( Chain, Name, Sym_p ).
```

Figure 1: Hashed Symbol Table Example

```
fib( M ) :-
  memo_ini( 20, M ),
  fib_memo( M ).

fib_memo( M0 ) :-
  memo_csel( M0, M, I, E ),
  I >= 2 |
  I1 := I - 1,
  I2 := I - 2,
  memo_sel( M, I1, V1 ),
  memo_sel( M, I2, V2 ),
  E := V1 + V2.

fib_memo( M0 ) :-
  memo_csel( M0, M, I, E ),
  I < 2 |
  E = 1.
```

Figure 2: Fibonacci Example

to reference the element index in its memo comprehension. The modes of `init_symtab/2` are

$m(\{< \texttt{init\_symtab}, 1 >\}) = \textbf{in}$
$m(\{< \texttt{init\_symtab}, 2 >, < \texttt{symtab}, 1 >\}) = \textbf{out}$
$m(\{< \texttt{init\_symtab}, 2 >, < \texttt{symtab}, 2 >\}) = \textbf{out}$
$m(\{< \texttt{init\_symtab}, 2 >, < \texttt{symtab}, 2 >, < * >\}) = \textbf{out}$

Note that the modes for `null_table/1` are

$m(\{< \texttt{null\_table}, 1 >\}) = \textbf{in}$
$m(\{< \texttt{null\_table}, 1 >, < * >\}) = \textbf{out}$

Since the memo is initialized by `null_table/1`, the caller can safely read its elements.

Because the calls to `memo_sel/3` in `insert_sym/3` and `lookup_sym/3` can be mode analyzed, we are guaranteed to get a mode error if `Chain` or any of its aliases are ever assigned to. This is precisely Ueda and Morita's "well-moded programs don't go wrong" condition [21]; we will get a static error rather than the runtime error which would otherwise result.

As a second example (Figure 2), we present a more complex memo comprehension to compute the Fibonacci sequence. Already, we see important advantages of MCs over chains of `memo_upd/4` operations. First, the MC will update all elements in parallel rather than sequentially if possible (although in this example only the first two elements can be updated in parallel). Second, even if multiple references to the memo are held, the memo may be updated in place rather than by copying.

```
conv( NInts, Ints, Base, V ) :-
  memo_ini( NInts, V ),
  init_all( Ints, V ),
  trans_all( Base, V ).

init_all( Ints, V0 ) :-
  memo_csel( V0, V, I, E ) |
  memo_sel( Ints, I, N ),
  memo_ini( 32, D ),
  E = item( N, D ).

trans_all( Base, V0 ) :-
  memo_csel( V0, V, _, E ),
  E = item( N, D ) |
  trans_int( N, Base, D ).

trans_int( N, Base, D0 ) :-
  memo_csel( D0, D, I, E ),
  I =:= 0 |
  Digit := N mod Base,
  Residue := ( N - Digit ) / Base,
  E = pair( Digit, Residue ).

trans_int( N, Base, D0 ) :-
  memo_csel( D0, D, I, E ),
  I > 0 |
  I1 := I - 1,
  memo_sel( D, I1, RL ),
  get_residue( RL, Residue_p ),
  Digit := Residue_p mod Base,
  Residue := ( Residue_p - Digit ) / Base,
  E = pair( Digit, Residue ).

get_residue( pair( _, Res0 ), Res ) :-
  Res = Res0.
```

Figure 3: Radix Conversion Example

Our final example (Figure 3) illustrates binding of nested arrays, showing the generality of the technique. The program `conv(Ints,Base,V)` converts an array of positive integers `Ints` into an array `V` of radix `Base` numbers, by sequentially computing residues. A single integer `Int` and its conversion is represented by the term:

```
item( Int, { D0-R0, D1-R1, ..., D31-R31 } )
```

where `Dn` is the n^{th} radix `Base` digit and `Rn` is the residue at the n^{th} position. An array of `item/2` terms is initialized by `conv/3` and `init_all/2`. Number conversion occurs in `trans_all/2` and `trans_int/3` with nested comprehensions. The innermost comprehension is sequentialized by the dependency incurred by the `memo_sel/3` goal. The process computing the n^{th} digit must wait for the residue of the $(n-1)^{st}$ digit to be computed.

5 Mode Analysis Of MCs

In this section, we informally describe our extensions to mode analysis to handle memo functions. A more formal description of the mode analysis framework appears to which our extensions apply is given in [8]. A critical first step in mode analysis of memo functions and comprehensions is to note that different memos may have different modes. Thus, for instance, one cannot blithely unify *all* `memo_sel/3` goals together! Though we cannot decide precisely which memo is denoted by a variable occurrence, we can safely approximate this notion. Thus, in the spirit of constraint-propagation mode analysis, we will initially assume that all memo functions refer to *unique* memos, and find relationships based on shared variable occurrences. This is similar to what is done in our current mode analyzers [14] to deal with the unification operator $=/2$.

To accomplish this, we first separate all the occurrences of memo functions, by assigning a unique integer subscript to each static occurrence of `memo_ini/2`, `memo_sel/3`, and `memo_upd/4` in the program. We then apply mode analysis rules (Figure 4) giving the relationship between the arguments of these functions.

Figure 6 graphically illustrates the rules using notation introduced in [18]. In brief, constraint graphs are layered networks consisting of structure nodes (squares) and variable nodes (circles). For example, `memo_sel/3` has three children corresponding to its three arguments. The first child is a variable M leading to a memo placeholder, leading to a variable E. Variable E also has a parent extending from the third argument of `memo_sel/3` because the abstract meaning of the function (in the mode domain) is to collapse all top-level elements in the memo into one element. Each variable node in a graph is labeled with its potential modes (one per input arc, since different input arcs can "see" a variable through different moded "glasses"). For instance, variable E in `memo_sel/3` has possible modes [**in**,**out**] or [**out**,**in**], enforcing the constraint that inputs in the first argument are seen as outputs in the third argument and vice versa. In fact, from these two parent paths, all graph nodes below E (represented as shaded subtree) are viewed as having opposite modes, effectively implementing the tell unification "M[$*$] = E." Further details of tell unification and the graph reduction semantics can be found in [18]. Which representation of the moding rules, axiomatic or graphical, is most understandable is a matter of taste. We wish to point

$m(\{<\texttt{memo\_ini}_i/2,1>\}) = \textbf{in}$
$m(\{<\texttt{memo\_ini}_i/2,2>\}) = \textbf{out}$
$m/\{<\texttt{memo\_ini}_i/2,2>,\ <*>\} = \textbf{in}$

$m(\{<\texttt{memo\_sel}_i/3,1>\}) = \textbf{in}$
$m/\{<\texttt{memo\_sel}_i/3,1>,\ <*>\} \neq m/\{<\texttt{memo\_sel}_i/3,3>\}$
$m(\{<\texttt{memo\_sel}_i/3,2>\}) = \textbf{in}$

$m(\{<\texttt{memo\_upd}_i/4,1>\}) = \textbf{in}$
$m/\{<\texttt{memo\_upd}_i/4,1>\} \neq m/\{<\texttt{memo\_upd}_i/4,4>\}$
$m/\{<\texttt{memo\_upd}_i/4,3>\} \neq m/\{<\texttt{memo\_upd}_i/4,4>,\ <*>\}$
$m(\{<\texttt{memo\_upd}_i/4,2>\}) = \textbf{in}$

Figure 4: Mode Analysis Rules For Memo Functions

$m(\{<\texttt{memo\_csel}_i/4,1>\}) = \textbf{in}$
$m/\{<\texttt{memo\_csel}_i/4,1>,\ <*>\} = m/\{<\texttt{memo\_csel}_i/4,4>\}$
$m/\{<\texttt{memo\_csel}_i/4,2>\} = \textbf{in}$
$m(\{<\texttt{memo\_csel}_i/4,3>\}) = \textbf{in}$

Figure 5: Mode Analysis Rules For Memo Comprehensions

out that the graph segments defining the memo functions and comprehensions are built from previously defined primitives.

The rules for moding a MC are similar in spirit to the rules for the simpler cases: the key is the correct moding of `memo_csel/4`. The memo must be read-only in the body of the MC, in order to ensure that the memo is updated only through the fourth argument of `memo_csel/4`. For instance if there is a body goal in `null_table` in the previous example which would alter $\texttt{Table}_1$, a mode conflict would be forced. On the other hand, the caller of the MC must have called the MC with the idea that the MC would bind some paths in the memo. Thus we propagate the modes of the memo elements through (`Table`), but allow the MC body to access the memo in a read-only fashion through the second argument to `memo_csel/4`. An example of this is the use of `D` in the body of `trans_int` in Figure 3.

The mode analysis rules for the MC function `memo_csel/4` are given in Figure 5. Note that *all* arguments to a MC are treated as input by the callee. This is because multiple

invocations of the MC will be created, and thus if more than one invocation tries to bind some path beginning at an argument to the MC, some assignment will fail.

A prototype implementation of these rules was added to one of our current mode analyzers. This mode analyzer [17] operates upon a finite set of paths, attempting to construct this set in such a way that the modes of all variable occurrences of the target program are uniquely and correctly determined. The original mode analyzer consists of 4487 lines of KL1 (including comments); 3362 lines of application-specific source code plus 1125 lines of KL1 devoted to general-purpose data structures such as sets and lists. The modified analyzer consists of 4021 lines of application-specific KL1. Of the 659 lines of new code, 427 lines comprise a module giving the new mode analysis rules for memo functions and comprehensions, and the remaining 232 lines represent changes to the original code necessary to integrate this module. There is no appreciable difference in the runtime performance of the analyzer (see [17] for the performance of the original system). Appendix A shows the previous three main example programs and their annotations as produced by the analyzer.

6 Semantics Of Memo Operations

To give a formal semantics to the memo operations, we give a reference implementation of memo functions and memo comprehensions in KL1, representing memos as lists. The reference language used is KL1 with the exception that names beginning with $ are not visible by the programmer. This prevents the programmer from observing the internals of the particular representation used in the reference implementation.

We first give a reference implementation of the "ordinary" memo functions, in Figure 7. These involve straightforward translations of indexing to recursion.

Next, we consider the semantics of memo comprehensions. Let p/k be a procedure of arity k whose clause heads have been flattened and consistently renamed, and each of whose clauses contain a memo comprehension guard `memo_csel(MO,M, I,E)`. Our strategy is to rewrite p to remove the memo comprehension as follows:

1. Construct a new procedure $\$p/(k+2)$ by prepending the arguments `I` and `E` of the `memo_csel/4` guard to the head of each clause of p, and then replacing the `memo_csel/4` guard in each clause with the guard "`MO = M`."

2. Let `$length/1` be a special guard which returns the length of a memo. (It is easy to see how to rewrite the program in terms of the reference implementation to remove this guard if necessary.) Construct a new procedure $\$p'/(k+1)$ as follows. Let the second and succeeding arguments of $\$p'$ be the same as those of p, and be denoted by ellipsis. Then $\$p'$ is given by

```
$p'( $I, ... ) :-
  $I < $length( MO ) |
  memo_sel( MO, $I, $E ),
  $p'( $I, $E, ... ),
  $I1 := $I + 1,
  $p( $I1, ... ).

$p( $I, ... ) :-
```

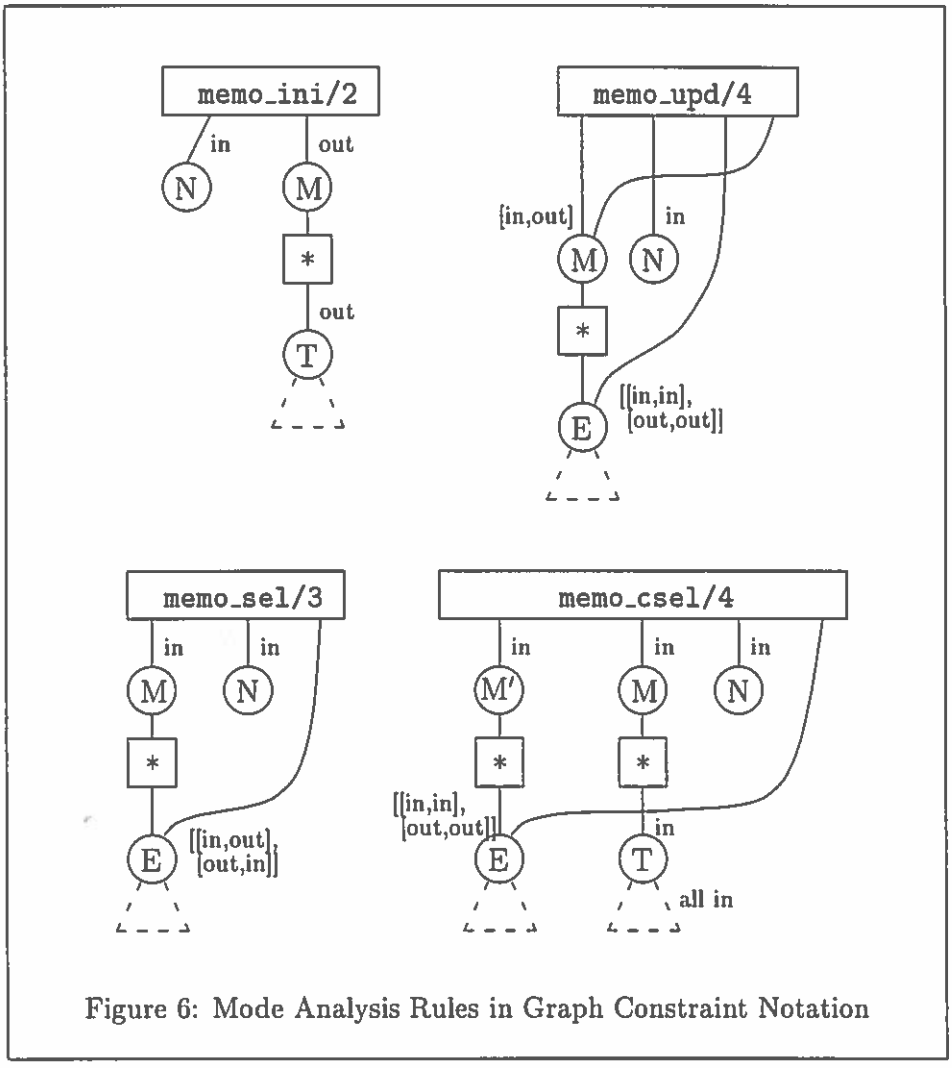

Figure 6: Mode Analysis Rules in Graph Constraint Notation

```
memo_ini( K, M ) :-
  K > 0 |
  K1 := K - 1,
  memo_ini( K1, $memo(L) ),
  M = $memo( [ _ | L ] ).
memo_ini( 0, M ) :-
  true |
  M = $memo( [] ).

memo_sel( $memo( [ _ | Es ] ), I, X ) :-
  I > 0 |
  I1 := I - 1,
  memo_sel( $memo( Es ), I1, X ).
memo_sel( $memo( [ E | _ ] ), 0, X ) :-
  true |
  X = E.

memo_upd( $memo( [ E | Es ] ), I, X, M ) :-
  I > 0 |
  I1 := I - 1,
  memo_upd( $memo( Es ), I1, X, $memo( L ) ),
  M = $memo( [ E | L ] ).
memo_upd( $memo( [ _ | Es ] ), 0, X, M ) :-
  true |
  M = $memo( [ X | Es ] ).
```

Figure 7: Semantics of "Ordinary" Memo Functions

```
$I >= $length( M0 ) |
true.
```

3. Replace all calls `p(...)` with calls `$p(0,...)`.

7 Conclusions

We have described a new technique for permitting safe mode analysis of concurrent logic programs that use arrays with both ground and non-ground elements. The implications of this work are that a broader class of programs can thus be optimized by mode analysis. We believe this is the first practical, *implemented* solution offered for this problem, particularly because it leverages proven mode analysis technology that is efficient.

We extend the concurrent logic language family to include builtin memo predicates, both guards and body goals, that allow memo comprehensions and standard memo updates and assignments. Thus algorithms that use fully-ground arrays can be programmed in the usual manner and are permissible under our mode analysis. In addition, programs that use non-fully-ground arrays can be programmed using our extensions, and if safe, will be accepted by our mode analysis.

We have implemented the memo rules within one of our mode analyzers: future work entails implementions within our other constraint-based analyzers [14, 17], and implementing comprehensions, via naive spawning, within the Monaco system [15]. We may also examine non-naive methods for implementing comprehensions, although we believe that the naive scheme uses no more process resources than existing vector implementations in concurrent logic languages such as KL1. Finally, we may need to further extend our basic scheme as we gain experience in its use, if we identify inflexibilities which make programming with memos more difficult than we currently expect.

Acknowledgements

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

References

[1] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *International Conference and Symposium on Logic Programming*, pages 669–683. University of Washington, MIT Press, August 1988. Extended version in Journal of Logic Programming, 1994.

[2] S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, July 1989.

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

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

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

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

[7] B. C. Massey and E. Tick. Demand-Driven Execution of Concurrent Logic Programs. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 215–224, Montreal, August 1994. North-Holland.

[8] B. C. Massey and E. Tick. Modes of Comprehension: Mode Analysis of Arrays and Array Comprehensions. Technical Report CIS-TR-94-10a, University of Oregon, Department of Computer Science, November 1994.

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

[10] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo 284-a, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1990.

[11] A. V. S. Sastry, W. Clinger, and Z. Ariola. Order-of-Evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates. In *Conference on Functional Programming Languages and Computer Architecture*, pages 266–275. Copenhagen, ACM Press, June 1993.

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

[13] R. Sundararajan. *Data Flow and Control Flow Analysis of Logic Programs.* PhD thesis, Department of Computer Science, University of Oregon, 1994. Also available as Technical Report CIS-TR-94-08.

[14] E. Tick. Practical Static Mode Analyses of Concurrent Logic Languages. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–214, Montreal, August 1994. North-Holland.

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

[16] E. Tick and M. Koshimura. Static Mode Analyses of Concurrent Logic Languages. Technical Report CIS-TR-94-06, Dept. of Computer Science, University of Oregon, Eugene, OR 97403, March 1994.

[17] E. Tick and M. Koshimura. Static Mode Analyses of Concurrent Logic Languages. *Journal of Programming Language Design and Implementation*, 1995. Accepted. Also available as University of Oregon Technical Report CIS-TR-94-06.

[18] 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*, pages 239–244. Kluwer Academic Publishers, 1994.

[19] 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.

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

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

A Sample Programs: Analyzer Output

Figures 8, 9 and 10 show the output of the extended mode analyzer. The annotation '^' is output and '?' is input. Constants and arithmetic goals are not annotated because their modes are fixed. Some guards are not annotated in our current prototype, e.g., `trans_all/2`: although the modes are known internally and the analysis is correct, early folding of such guards makes later annotation difficult.

Interestingly, our original version of the radix conversion program was ill-moded. The original definition of `trans_int/3` included:

```
...
memo_sel( D, I1, RL ),
RL = pair( _, Residue_p ),
...
```

which is a common type of moding error. The analyzer detected this error, leading us to make the modification listed in Figure 10.

```
fib(M^) :-
  memo_ini(20,M^),
  fib_memo(M?).

fib_memo(M0?) :-
  memo_csel(M0?,M?,I?,E^),
  I? >= 2 |
  I1 := I - 1,
  I2 := I - 2,
  memo_sel(M?,I1?,V1^),
  memo_sel(M?,I2?,V2^),
  E := V1 + V2.

fib_memo(M0?) :-
  memo_csel(M0?,M?,I?,E^),
  I? < 2 |
  E^ = 1.
```

Figure 8: Fibonacci Example as Annotated by Analyzer

```
init_symtab(NBuckets?,SymTab^) :-
  memo_ini(NBuckets?,Table^),
  null_table(Table?),
  SymTab^ = symtab(NBuckets?,Table?).

null_table(Table?) :-
  memo_csel(Table?,_,_,E^) |
  E^ = [].

insert_sym(symtab(NBuckets?,Table?),Sym?,SymTab_p^) :-
  hash(Sym?,NBuckets?,Hash^),
  memo_sel(Table?,Hash?,Chain^),
  Chain_p^ = [Sym?|Chain?],
  memo_upd(Table?,Hash?,Chain_p?,Table_p^),
  SymTab_p^ = symtab(NBuckets?,Table_p?).

lookup_sym(symtab(NBuckets?,Table?),Name?,Sym^) :-
  hash(Sym?,NBuckets?,Hash^),
  memo_sel(Table?,Hash?,Chain^),
  follow_chain(Chain?,Name?,Sym^).

follow_chain([],_,Sym^) :-
  Sym^ = nomatch.

follow_chain([Sym?|Syms?],Name?,Sym_p^) :-
  sym_match(Sym?,Name?,B^),
  follow_chain_1(B?,[Sym?|Syms?],Name?,Sym_p^).

follow_chain_1(true,[Sym?|_],_,Sym_p^) :-
  Sym_p^ = Sym?.
follow_chain_1(false,[_|Chain?],Name?,Sym_p^) :-
  follow_chain(Chain?,Name?,Sym_p^).

sym_match(Sym?,Sym?,B^) :-
  B^ = true.
sym_match(Sym1?,Sym2?,B^) :-
  Sym1? \= Sym2? |
  B^ = false.

driver(T_p^,Sym^) :-
  init_symtab(5,T^),
  insert_sym(T?,x,T_p^),
  lookup_sym(T_p?,x,Sym^).
```

Figure 9: Hashed Symbol Table Example as Annotated by Analyzer

```
conv(NInts?,Ints?,Base?,V^) :-
  memo_ini(NInts?,V^),
  init_all(Ints?,V?),
  trans_all(Base?,V?).

init_all(Ints?,V0?) :-
  memo_csel(V0?,V?,I?,E^) |
  memo_sel(Ints?,I?,N^),
  memo_ini(32,D^),
  E^ = item(N?,D?).

trans_all(Base?,V0?) :-
  memo_csel(V0?,V?,_,E?),
  E = item(N,D) |
  trans_int(N?,Base?,D?).

trans_int(N?,Base?,D0?) :-
  memo_csel(D0?,D?,I?,E^),
  I? =:= 0 |
  Digit := N mod Base,
  Residue := (N-Digit)/Base,
  E^ = pair(Digit?,Residue?).
trans_int(N?,Base?,D0?) :-
  memo_csel(D0?,D?,I?,E^),
  I? > 0 |
  I1 := I - 1,
  memo_sel(D?,I1?,RL^),
  get_residue(RL?,Residue_p^),
  Digit := Residue_p mod Base,
  Residue := (Residue_p-Digit)/Base,
  E^ = pair(Digit?,Residue?).

get_residue(pair(_,Res0?),Res^) :-
  Res^ = Res0?.

driver(X^) :-
  memo_ini(12,V^),
  countup(V?),
  conv(12,V?,16,X^).

countup(X?) :-
  memo_csel(X?,_,I?,E^) |
  E^ = I?.
```

Figure 10: Radix Conversion Example as Annotated by Analyzer

B Mode Analysis Rules

In explaining the mode analysis rules, it is convenient to first develop some notation. Let H be the set of all possible procedure/argument-index tuples of a program. Let S be the (countably infinite) set of all possible sequences of functor/argument-index tuples of the program (sequences are notated using surrounding curly braces {}). We define a path as a sequence of the form $\{h\} \cdot s$, where $h \in H$, $s \in S$, and $\cdot$ is sequence concatenation, and denote the set of all possible paths of the program by P.

Assume that the program's variables have been renamed using unique names for the variables of each clause, and that each occurrence of each term of the program has been given a unique subscript. We say that a path $p = \{<f, k>\} \cdot s$ derives a set $T(p)$ of occurrences of terms of the program, as follows: A term occurrence t is in $T(p)$ iff for some clause or call of f the k^{th} argument of f is a term occurrence t' such that descending into t' via s leads to the term occurrence t. The predicate $var(t)$ is true iff t is a variable occurrence, and the function $V(t)$ maps variable occurrences to variables. The predicates $head(t)$, $guard(t)$, and $body(t)$ refer to the location of t, and $hog(t) = head(t) \vee guard(t)$.

Associated with each path $p \in P$ is a mode m, a member of the set $M = \{in, out\}$. The function $m(p) : P \rightarrow M$ gives the mode associated with p. For brevity, we also define notation (due to Ueda) describing the mode of (possibly infinite) sets of paths. The notation

$$m/p = m'$$

where $m' \in M$ (*i.e.*, $m' = in$ or $m' = out$), is shorthand for

$$\forall s \in S : m(p \cdot s) = m'$$

We define equality and inequality of modes in the obvious way. This allows us to introduce the useful notation

$$m/p = m/p'$$

which is shorthand for

$$\forall s \in S : m(p \cdot s) = m(p' \cdot s)$$

and similarly for inequality.

We establish some standard variables which will be useful for brevity. Let p and q range over P, with $p \neq q$. Further, let t range over $T(p)$ such that $\neg var(t)$, and let v and w range over $T(p)$ and $T(q)$ respectively, such that $var(v)$ and $var(w)$ and $v \neq w$, but $V(v) = V(w)$. Finally, let s range over S.

We are now ready to give a set of rules for determining the modes of paths of a program. These rules are originally due to Ueda [19], although this formalization of them differs somewhat in character from the original.

The first few rules are quite simple. The first rule says that paths leading to non-variable terms must have input mode:

$$\neg var(t) \Rightarrow m(p) = in$$

The second rule says that paths leading to occurrences of any variable in the head or guard of a clause must have the same mode:

$$hog(v) \wedge hog(w) \Rightarrow m/p = m/q$$

This can be strengthened substantially in case guards all require ground arguments (as, for example, in *FCP*(|)[12]) to require:

$$hog(v) \Rightarrow m/p = in$$

Even when the weaker form of the preceding rule is used, we can still say something about repeated occurrences in the head: all paths here have input mode:

$$head(v) \wedge head(w) \Rightarrow m/p \leftrightharpoons m/q = in$$

The rules involving paths leading to body variables are somewhat more complicated. There are several alternative possibilities for formulating these rules, all based on the following intuition. Imagine that a clause C has m head occurrences $X_{h_1}...X_{h_m}$ of a variable X, and n body occurrences $X_1...X_n$. Since, by the previous rules, all of $X_{h_1}...X_{h_m}$ have the same mode, we will simply select X_{h_1} as a surrogate for the others, and call it X_h. The basic restriction on body variables is that there cannot be more than one *out* occurrence of X in the body, since this would imply that X might be unified with two different values, leading to program failure. By symmetry, we may thus choose to renumber the body occurrences so that the output occurrence, if any, is X_1. Finally we note that, by the definition of output mode, if X_1 is *out*, the binding for X is produced in the body of C, and thus X_h should be *out*. When X_1 is *in*, since we know that $X_2...X_n$ are *in*, X must not be bound in the body of C, and thus by the definition of input mode X_h is *in*. All of the above reasoning leads to the conclusion that there are really only two possible assignments of modes to the occurrences of X:

	X_h	X_1	X_2	...	X_n
1	*in*	*in*	*in*	...	*in*
2	*out*	*out*	*in*	...	*in*

Note that the above reasoning doesn't apply merely to occurrences of the variable itself: a variable constrains the modes of all paths below it in exactly the same fashion. Thus, if the modes of paths p and q are somehow constrained, the modes of $p \cdot s$ and $q \cdot s$ will be likewise constrained. With this understanding, we will speak informally of rules constraining the modes of "variable occurrences." Note also that there may not be any head occurrences of X: in this case, row 2 of the table is the only possibility (X can only be bound by a caller of a clause if it is visible to the caller, and thus some body occurrence must have output mode in this case).

We will thus formulate the mode rules for body variables in a fashion that allows us to draw conclusions about the modes of some occurrence of a variable from other occurrences whenever possible. This form is thus amenable to automatic implementation. (The practical difficulty in this approach is in finding a finite subset of the infinite set of paths of a program

for which the analysis is complete and safe. Koshimura discusses this problem in the context of mode analysis using MGTP [16].) Intuitively, whenever we have enough information to distinguish which of the two rows of the above table corresponds to the modes of occurrences of X, and to identify which output body occurrence corresponds to X_1 (if necessary), we may fill in the missing information. Thus, the underlined columns of a row of the following table are information which may be filled in when the rest of the columns in that row are known.

	X_h	X_1	X_2	...	X_n
a	*in*	*in*	*in*	...	*in*
b	*in*	*in*	*in*	...	*in*
c	*out*	*out*	*in*	...	*in*
d	*out*	*out*	*in*	...	*in*

Rows of this table thus correspond to inference rules in a straightforward fashion:
(a) If a head occurrence of a variable is input mode, all body occurrences are input mode:

$$head(v) \wedge m(p \cdot s) = in \wedge body(w) \Rightarrow m(q \cdot s) = in$$

(b) If all body occurrences of a variable are input mode, then the head is input mode:

$$(\forall v : body(v) \Rightarrow m(p \cdot s) = in) \wedge head(w) \Rightarrow m(q \cdot s) = in$$

(c) If all body occurrences of a variable but one are input mode, and the head is output mode, then the remaining body occurrence is output mode. Here let r range over P, with $r \neq p$ and $r \neq q$. Further, let x range over $T(r)$, such that $var(x)$ and $x \neq v$ and $x \neq w$ and $V(x) = V(w)$.

$$\begin{aligned}(\forall v : body(v) \Rightarrow m(p \cdot s) = in) \\ \wedge head(w) \wedge m(q \cdot s) = out \\ \wedge body(x) \Rightarrow m(r \cdot s) = out\end{aligned}$$

(d) If a body occurrence is output mode, then all other body occurrences are input mode, and the head is output mode:

$$\begin{aligned}body(v) \wedge m(p \cdot s) = out \wedge body(w) &\Rightarrow m(q \cdot s) = in \\ body(v) \wedge m(p \cdot s) = out \wedge head(w) &\Rightarrow m(q \cdot s) = out\end{aligned}$$

These are the ground rules of the analysis. Builtin procedures are handled by special rules: in particular, the restriction that all objects denoted by a path have the same mode does not apply to paths originating in builtin procedures (equivalently, one can affix a unique subscript to each occurrence of a builtin, as is done in the text). Instead, special case rules are used to indicate mode constraints for calls to these procedures. Also confusing is that some guard and body constraints have identical names, and are distinguished only by position. We will distinguish these using subscripts g (guard) and b (body).

For an occurrence of the body equality (*i.e.*, tell unification) predicate $=_b/2$, the mode constraint is that the modes on opposite sides are opposite:

$$m/\{<=_b/2,1>\} \neq m/\{<=_b/2,2>\}$$

In dialects in which guard equality testing (*i.e.*,ask unification) predicates $=_g/2$ do not require ground arguments in order to succeed, it is still necessary that the modes on opposite sides are equal:

$$m/\{<=_g/2,1>\} = m/\{<=_g/2,2>\}$$

Most guards, such as guard arithmetic assignment, have all input arguments:

$$m(\{<:=_g/2,1>\}) = m/\{<:=_g/2,2>\} = in$$

(confusingly, the LHS of the assignment is also *in*, since it is produced in the head of the clause rather than its body) whereas body arithmetic assignment expects its RHS to be input and produces an output LHS:

$$\begin{aligned} m(\{<:=_b/2,1>\}) &= out \\ m/\{<:=_b/2,2>\} &= in \end{aligned}$$

Our memo comprehension rules fit into this mode analysis framework in exactly the same way as the rules for other builtin functions described above. When defining builtin functions, it is important that these functions are useful, while still allowing mode constraints which fit into this formal framework; otherwise, it may be impossible to mode-analyze programs which use these builtins. The mode constraints for memo functions given in the text satisfy this requirement. Thus, our work in providing mode rules for memo builtins is an important but fully compatible extension of earlier work.