Proc. International Workshop on Parallel Symbolic
Languages and Systems (PSLS'95, October 1995),
Lecture Notes in Computer Science 1068, pp. 134–153, 1996.

Experiences with Strong Moding in Concurrent Logic/Constraint Programming

Kazunori Ueda*

Department of Information and Computer Science
Waseda University
4-1, Okubo 3-chome, Shinjuku-ku, Tokyo 169, Japan
`ueda@ueda.info.waseda.ac.jp`

Abstract. Strong moding is turning out to play fundamental roles in concurrent logic programming (or in general, concurrent constraint programming) as strong typing does but in different respects. "Principal modes" can most naturally be represented as feature graphs and can be formed by unification. We built a mode analyzer, implementing mode graphs and their operations by means of concurrent processes and streams (rather than records and pointers). This is a non-trivial programming experience with complicated process structures and has provided us with several insights into the relationship between programming with dynamic data structures and programming with dynamic process structures. The mode analyzer was then applied to the analyzer itself to study the characteristics of the mode constraints it imposed and of the form of large mode graphs. Finally, we show how our framework based on principal moding can be extended to deal with (1) random-access data structures, (2) mode polymorphism, (3) higher-order constructs, and (4) various non-Herbrand constraint systems.

1 Strong Moding in Concurrent Logic/Constraint Programming

Historically, two different notions of modes have been studied in logic programming. Modes of the first kind are concerned with reasoning about temporal properties (i.e., time-of-call/exit instantiation states) of variables, and are used for answering questions such as "Is `X` unbound when `p(X)` is called?" They are usually analyzed using abstract interpretation and necessarily depends on "computation rules." Modes of the second kind, which are less well-known and we are going to deal with here, are for reasoning about non-temporal properties of variables, and are intended to answer questions such as "Which occurrence of `X` in the configuration `p(X),q(X),r(X)` may instantiate `X` eventually?" They are independent of how goals are executed and thus can be regarded as a language construct.

* Supported in part by the Ministry of Education and Japan Information Processing Development Center (JIPDEC).

Variables in logic programming languages can be viewed as communication channels. A variable may in general have many writers and many readers (blackboard communication), but in most cases variables are used for cooperative communication, namely point-to-point communication (one writer, one reader) or multicasting/broadcasting (one writer, many readers). In both logic programming and concurrent logic programming, it seems important to be able to distinguish between competitive and cooperative communication and, for the latter case, to infer the communication protocols used.

From our experience, we are strongly confident that variables in concurrent logic languages can be restricted to cooperative communication without loss of expressive power. Rather, by doing so, we will benefit very much from strong moding, as we do from strong typing in many other languages:

– It helps programmers understand their programs better.
– It detects a certain kind of program errors at compile-time.
– It establishes some fundamental properties statically:
  • Well-moded programs do not collapse due to unification failure (failure of unification body goals).
  • All variables are guaranteed to become *ground* terms upon termination.
  • It distinguishes between data with a single reference and those with multiple references. This provides us with basic information for compile-time garbage collection.
– It provides basic information for program optimization:
  • elimination of various runtime checks,
  • (much) simpler distributed unification,
  • message-oriented implementation [7][9].
– It encourages modular programming by making programmers better aware of module interface.

Since the mode system for Flat GHC was proposed [5][7], some attempts have been made to implement the system [8], but no attempts have been made to implement a mode analyzer based on the unification of mode graphs as it appeared in [5]. In this paper, we describe our first experiences with the graph-based mode analyzer. Also, we show how our framework based on principal moding can be extended to deal with (1) random-access data structures, (2) mode polymorphism, (3) higher-order constructs, and (4) various non-Herbrand constraint systems.

We assume familiarity with the basic idea of the mode system, though it will be described briefly in Section 2. Full detail of the mode system, with proofs of the fundamental properties, can be found in [7]. Implications of the mode system are discussed in [10], which contains informal introduction to concurrent logic programming and the mode system as well.

2 The Mode System

As discussed in [10], a logical variable with exactly two occurrences can be compared to a signal cable which has a certain structure (e.g., array of wires) and conveys information under some established protocol. A piece of information flowing into the n-th pin of the plug at one end of a cable will come out from the n-th pin at the other end of the cable, which means that two ends/occurrences of a cable/variable should have exactly inverse (i.e., complementary) polarity structures.

A variable with three or more occurrences in a run-time configuration will be used as a *hub* for one-to-many communication. The polarity structures of the terminals of a hub should be given so that for each set of corresponding positions in those structures, exactly one of them is the inlet of information and the others are outlets.

We call variables with exactly two occurrences *linear variables* and other variables *non-linear variables*, where we do not count the second and subsequent occurrences of a variable in a clause head or any of the occurrences in guard goals. We call clauses not containing non-linear variables *linear clauses* and clauses containing non-linear variables *non-linear clauses*.

An argument of a goal can be compared to a socket of a device. To be compatible, a plug and a socket should have opposite polarity structures when viewed from outside.

The purpose of our mode system is exactly to assign polarity structures to the arguments of predicates defining the behavior of goals, so that each part of data structures will be determined cooperatively, namely by *exactly one* goal. If the part has more than one writer goal, the communication is competitive and hence not cooperative. If the part has no writer at all, the communication is not cooperative, though not competitive, because the readers will never get a value.

A mode is a function from the set of paths specifying positions in data structures occurring in goals, denoted P_{Atom}, to the set $\{in, out\}$. Paths here are not strings of argument positions; instead they are strings of $\langle$*symbol*, *argument-position*$\rangle$ pairs in order to be able to specify positions in data structures that are yet to be formed.

Formally, the sets of paths for specifying positions in terms and atomic formulas are defined, respectively, using disjoint union as:

$$P_{Term} = (\sum_{f \in Fun} N_f)^* \quad , \quad P_{Atom} = (\sum_{p \in Pred} N_p) \times P_{Term} \quad ,$$

where *Fun* and *Pred* are the sets of function and predicate symbols, respectively, and N_f and N_p are the sets of positive integers up to and including the arities of f and p, respectively.

Mode analysis tries to find a mode $m : P_{Atom} \to \{in, out\}$ under which every piece of communication will be performed cooperatively. Such a mode is called a *well-moding*. A well-moding is computed by constraint solving. Function symbols in a program/goal clause will impose constraints on the possible polarities of the paths at which they occur. Variable symbols may constrain the polarities not only of the paths at which they occur but of any positions below those paths. The set of all these constraints syntactically imposed by the symbols or the symbol occurrences in a program does not necessarily define a unique mode because the constraints are usually not strong enough to define one. Instead it defines a 'principal' mode that can best be expressed as a mode graph, as we will see in Section 3.

Constraints imposed by a clause h :- G | B, where G and B are multisets of atomic formulae, are summarized in Figure 1. Here, *Var* denotes the set of variable symbols, and $\widetilde{a}(p)$ denotes a symbol occurring at p in an atomic formula a. A *submode* of m at p, denoted m/p, is a function (from P_{Term} to $\{in, out\}$) such that $(m/p)(q) = m(pq)$. *IN* and *OUT* are submodes that always return *in* or *out*, respectively. An overline, "$^{-}$", inverts the polarity of a mode, a submode, or a mode value.

(HF) $\forall p \in P_{Atom}\big(\widetilde{h}(p) \in \mathit{Fun} \Rightarrow m(p) = \mathit{in}\big)$
(if the symbol at p in h is a function symbol, $m(p) = \mathit{in}$),

(HV) $\forall p \in P_{Atom}\big(\widetilde{h}(p) \in \mathit{Var} \land \exists p' {\neq} p\big(\widetilde{h}(p) = \widetilde{h}(p')\big) \Rightarrow m/p = \mathit{IN}\big)$
(if the symbol at p in h is a variable occurring elsewhere in h, then $m/p = \mathit{IN}$),

(GV) $\forall p, p' \in P_{Atom}\, \forall a \in G\big(\widetilde{h}(p) \in \mathit{Var} \land \widetilde{h}(p) = \widetilde{a}(p')$
$\qquad \Rightarrow \forall q \in P_{Term}\big(m(p'q) = \mathit{in} \Rightarrow m(pq) = \mathit{in}\big)\big)$
(if the same variable occurs both at p in h and at p' in G, then
$\forall q \in P_{Term}\big(m(p'q) = \mathit{in} \Rightarrow m(pq) = \mathit{in}\big)$)

(BU) $\forall k > 0\, \forall t_1, t_2 \in \mathit{Term}\big((t_1 =_k t_2) \in B \Rightarrow m/\langle =_k, 1\rangle = \overline{m/\langle =_k, 2\rangle}\big)$
(the two arguments of a unification body goal have complementary submodes)

(BF) $\forall p \in P_{Atom} \forall a \in B\big(\widetilde{a}(p) \in \mathit{Fun} \Rightarrow m(p) = \mathit{in}\big)$
(if the symbol at p in a body goal is a function symbol, $m(p) = \mathit{in}$),

(BV) Let $v \in \mathit{Var}$ occur $n\,(\geq 1)$ times in h and B at $p_1, \ldots, p_n$, of which the occurrences in h are at $p_1, \ldots, p_k$ $(k \geq 0)$. Then

$$\begin{cases} \mathcal{R}\big(\{m/p_1, \ldots, m/p_n\}\big), & k = 0; \\ \mathcal{R}\big(\{\overline{m/p_1}, m/p_{k+1}, \ldots, m/p_n\}\big), & k > 0; \end{cases}$$

where $\mathcal{R}$ is a 'cooperativeness' relation:

$$\mathcal{R}(S) \overset{\text{def}}{=} \forall q \in P_{Term}\, \exists s \in S\big(s(q) = \mathit{out} \,\land\, \forall s' \in S\backslash\{s\}\, \big(s'(q) = \mathit{in}\big)\big)$$

Fig. 1. Mode constraints imposed by a clause h :- G | B.

Unification body goals, dealt with by Constraint (BU), are *polymorphic* in the sense that different goals are allowed to have different modes. To deal with polymorphism, we give each unification body goal a unique number. General treatment of polymorphism will be discussed in Section 5.2.

As an example, consider a `merge` program:

```
merge([],Y,Z) :- true | Z=1Y.
merge(X,[],Z) :- true | Z=2X.
merge([A|X],Y,Z0) :- true | Z0=3[A|Z],merge(X,Y,Z).
merge(X,[A|Y],Z0) :- true | Z0=4[A|Z],merge(X,Y,Z).
```

From the third clause, for instance, we obtain the following eight constraints, where "." stands for the constructor of non-empty lists:

$$\begin{aligned}
m(\langle \texttt{merge}, 1\rangle) &= \mathit{in} && \text{by (HF) applied to "."}\\
m/\langle \texttt{=}_3, 1\rangle &= \overline{m/\langle \texttt{=}_3, 2\rangle} && \text{by (BU) applied to } \texttt{=}_3\\
m(\langle \texttt{=}_3, 2\rangle) &= \mathit{in} && \text{by (BF) applied to "."}\\
m/\langle \texttt{merge}, 1\rangle\langle \texttt{.}, 1\rangle &= m/\langle \texttt{=}_3, 2\rangle\langle \texttt{.}, 1\rangle && \text{by (BV) applied to } \texttt{A}\\
m/\langle \texttt{merge}, 1\rangle\langle \texttt{.}, 2\rangle &= m/\langle \texttt{merge}, 1\rangle && \text{by (BV) applied to } \texttt{X}\\
m/\langle \texttt{merge}, 2\rangle &= m/\langle \texttt{merge}, 2\rangle && \text{by (BV) applied to } \texttt{Y}\\
m/\langle \texttt{merge}, 3\rangle &= m/\langle \texttt{=}_3, 1\rangle && \text{by (BV) applied to } \texttt{Z0}\\
m/\langle \texttt{=}_3, 2\rangle\langle \texttt{.}, 2\rangle &= \overline{m/\langle \texttt{merge}, 3\rangle} && \text{by (BV) applied to } \texttt{Z}
\end{aligned}$$

From the entire set of clauses, we obtain 24 constraints, of which 6 are of the form $m(p) = \mathit{in}$, 12 are of the form $m/p_1 = m/p_2$, and 6 are of the form $m/p_1 = \overline{m/p_2}$. Elimination of the constraints on $\texttt{=}_k$, however, leaves only four constraints:

$$\begin{aligned}
m(\langle \texttt{merge}, 1\rangle) &= \mathit{in}\\
m/\langle \texttt{merge}, 1\rangle\langle \texttt{.}, 2\rangle &= m/\langle \texttt{merge}, 1\rangle\\
m/\langle \texttt{merge}, 2\rangle &= m/\langle \texttt{merge}, 1\rangle\\
m/\langle \texttt{merge}, 3\rangle &= \overline{m/\langle \texttt{merge}, 1\rangle}
\end{aligned}$$

We could handle these constraints as logical formulae, but mode graphs described below allow us to represent and manipulate constraints efficiently.

3 Mode Graphs and Principal Modes

It turns out that most of the mode constraints are either of the six forms: (i) $m(p) = \mathit{in}$, (ii) $m(p) = \mathit{out}$, (iii) $m/p = \mathit{IN}$, (iv) $m/p = \mathit{OUT}$, (v) $m/p_1 = m/p_2$, or (vi) $m/p_1 = \overline{m/p_2}$. We call (i)–(iv) *unary* constraints and (v)–(vi) *binary* constraints.

A set of binary and unary mode constraints can be represented as a feature graph (feature structures with cycles), called a *mode graph*, in which

1. paths represent paths in P_{Atom},
2. nodes may have mode values determined by unary constraints,
3. arcs may have "negative signs" that invert the interpretation of the mode values beyond those arcs, and

4. binary constraints are represented by the sharing of nodes.

Figure 2 is the mode graph of the four constraints from the `merge` program.

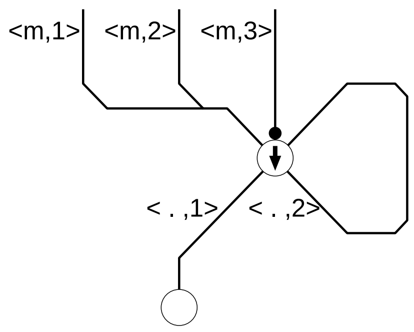

Fig. 2. Mode graph of the `merge` program.

An arc of a mode graph represents the pair of a predicate/function symbol (abbreviated to its initial in the figures) and an argument position. The pair exactly corresponds to a feature of a feature graph. A sequence of features forms a path both in the sense of our mode system and in the graph-theoretic sense.

A node is possibly labeled with a mode value (*in* shown as "$\downarrow$", or *out* shown as "$\uparrow$") to which any paths p_1, p_2, ... terminating with that node are constrained, or with a constant submode (*IN* shown as "$\downarrow$" with a grounding sign (as in Figure 4), or *OUT*) to which the submodes m/p_1, m/p_2, ... are constrained.

An arc is either a negative arc (bulleted in the figures) or a positive arc. When a path passes an odd number of negative arcs, that path is said to be *inverted*, and the mode value of the path should be understood to be inverted. Thus the number of bulleted arcs on a path determines the *polarity* of the path.

A binary constraint of the form $m/p_1 = m/p_2$ or $m/p_1 = \overline{m/p_2}$ is represented by a shared node with two (or more) incoming paths with possibly different polarities. When the polarities of the two incoming paths are different, the shared node stands for complementary submodes; otherwise the node stands for identical submodes.

Figure 2 has a node, under the arc labeled $\langle\,.\,,1\rangle$, that expresses no information at all. It was created to express binary constraints, but all its parent nodes were later merged into a single node by other constraints.

As another example, consider a program that simply unifies the two arguments:

```
p(X,Y) :- true | X=Y.
```

The program forms a simple mode graph shown in Figure 3. This graph can be viewed as the *principal mode* of the predicate p, which represents many possible particular modes satisfying the constraint $m/\langle \mathtt{p}, 1\rangle = \overline{m/\langle \mathtt{p}, 2\rangle}$. In general, the principal mode of a well-moded program, represented as a mode graph, is uniquely determined, as long as all the mode constraints imposed by the program are unary or binary.

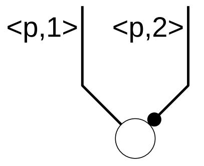

Fig. 3. Mode graph of the unify program.

Constraints imposed by the rule (BV) may be non-binary. Non-binary constraints are imposed by non-linear variables, and cannot be represented as mode graphs by themselves. However, by *delaying* them, most of them will be reduced to unary/binary ones by other constraints, as we will see later. In this case they can be represented in mode graphs, and the programs that imposed them have unique principal modes (as long as they are well-moded).

Theoretically, some non-binary constraints may remain unreduced, whose satisfiability must be checked eventually. However, a much more practical solution is to let programmers declare the modes of the paths where non-linear variables occur.

The union (i.e., conjunction) of two sets of constraints can be computed efficiently as unification over feature graphs. For instance, adding a new constraint $m/p_1 = m/p_2$ causes the subgraph rooted at p_1 and the subgraph rooted at p_2 to be unified. A good news is that an efficient unification algorithm for feature graphs has been established [1].

Figure 4 shows the mode graph of a quicksort program using difference lists. The head and the tail of a difference list, namely the second and the third arguments of `qsort`, are constrained to have complementary submodes.

Figure 5 shows the driver of a demand-driven sequence generator that receives messages `done` or `more` from an I/O stream and keeps sending requests to the sequence generator until `done` is received. The figure shows how mutual recursion can be dealt with: `driver` calls `checkinput` after sending a message and `checkinput` calls `driver` after sending two messages. These two predicates form a cycle with three nodes in the mode graph.

```
qsort([],     Ys0,Ys ):- true | Ys=Ys0.
qsort([X|Xs],Ys0,Ys3):- true |
    part(X,Xs,S,L), qsort(S,Ys0,[X|Ys2]), qsort(L,Ys2,Ys3).
part(_,[],    S, L ):- true | S=[], L=[].
part(A,[X|Xs],S0,L ):- A>=X | S0=[X|S], part(A,Xs,S,L).
part(A,[X|Xs],S, L0):- A< X | L0=[X|L], part(A,Xs,S,L).
```

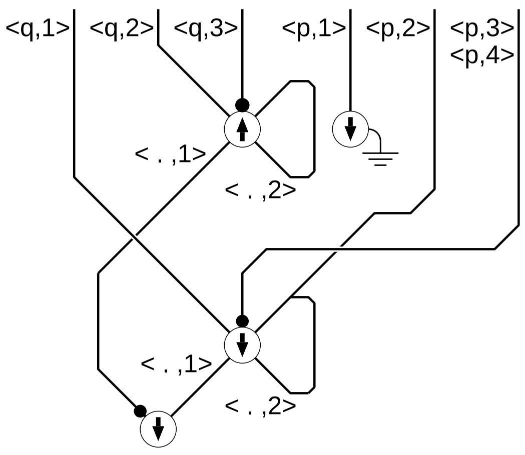

Fig. 4. A quicksort program and its mode graph.

```
driver(Fs,IOs0):- true | IOs0=[gett(X)|IOs1], checkinput(Fs,IOs1,X).
checkinput(Fs, IOs, done):- true | Fs=[], IOs=[].
checkinput(Fs0,IOs0,more):- true |
    Fs0=[N|Fs1], IOs0=[putt(N),nl|IOs1], driver(Fs1,IOs1).
```

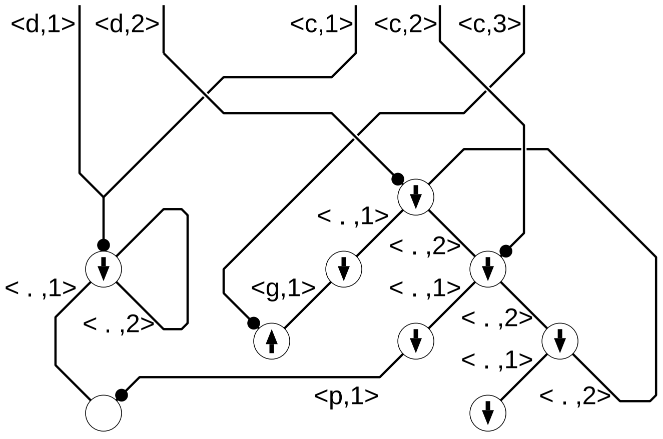

Fig. 5. A mutually recursive program and its mode graph.

4 Implementing Mode Analysis

We have implemented a mode analyzer for Flat GHC, and have extended it to deal with most features of KL1 [6]. The analyzer is itself a well-moded program entirely written in KL1. Mode analysis proceeds as follows:

1. Constraint generation
   (a) KL1 clauses are translated into their normal forms [7].
   (b) Calls to polymorphic built-in predicates are identified and numbered.
   (c) A symbol table is generated, which records all the occurrences of predicate, function, constant, and variable symbols.
   (d) Constraints imposed by symbols, occurrences of symbols, and built-in predicates are generated according to the rules in Figure 1.
2. Constraint solving
   (a) The constraints are put into an empty mode graph to form a graph representing their conjunction.
   (b) If successful, the final graph state is retrieved; otherwise, failure and its reason are reported.

4.1 Constraint Generation

Since the first part, constraint generation, is simply the syntactic manipulation of a given program, we only note that KL1 differs from Flat GHC in the following aspects, which are all handled appropriately.

1. KL1 supports modularization; a predicate is identified by the pair of the module it belongs to and the predicate name.
2. A program commences execution by calling the nullary predicate `main` in the module `main`.
3. KL1 supports vectors. A vector is denoted $\{element_1, \ldots, element_n\}$, which is regarded as a structure with a special function symbol indicating that the term is a vector of length n.
4. KL1's guard built-in predicates may have output arguments (e.g., `functor`), which is not allowed in Flat GHC. They are treated as a source of information like input arguments in the head. Output arguments in a clause guard may be used both for providing values to be used in the body (cooperative communication) or for checking the applicability of the clause (competitive communication).
5. KL1 features character strings and string operations, but a string can be treated as a constant because it cannot contain uninstantiated variables.

4.2 Constraint Satisfaction

Our constraint solver represents mode graphs using processes and streams. Although not shown in the figures, each mode graph has a root node corresponding to the empty path. [2] New constraints are added by sending messages to the root node. Retrieval of mode information can be done by sending messages as well.

Process representation is interesting in its own right as an experiment of programming complicated process structures. We must deal with (1) graphs with node sharing and cycles and (2) the merging of graph nodes. It is not clear whether they can be programmed in a strongly moded framework and how much parallelism can be exploited.

One alternative to the process representation would be to use rational trees for representing mode graphs. Unfortunately, even if the underlying language features rational trees, we cannot rely too much on the built-in unification but must define unification for feature graphs ourselves. One reason is that the unification procedure should be able to report failure explicitly. Another obvious alternative would be to represent mode graphs using arrays of nodes and arcs, modifying them as new constraints are added. The sequential nature of this approach can make global operations such as termination detection easier to implement. However, we chose the process approach to explore the viability of process representation of dynamic data structures and to be able to exploit parallelism in future (Section 4.4).

The `node` process representing an ordinary node contains the following arguments:

1. an input stream for receiving messages,
2. a node identifier used for the equality checking of nodes,
3. a mode value (*unconstrained*, *input*, or *output*),
4. a list of features corresponding to output streams,
5. a list of output streams,
6. a termination variable shared by all node processes, and
7. a flag for the retrieval of the graph state.

When there is more than one incoming arc (stream) to a node, they are merged using frontend `merge` processes. A negative arc is represented using an inverter filter process that inverts the interpretation of mode values contained in messages.

A node is created by sending a message to the node server, which provides each node with its identifier. Unlike in procedural languages, the identity of nodes cannot be checked by pointer comparison but this capability should be provided explicitly by the process. Assignment (pointer copying) and equality checking (pointer comparison) are provided as basic operations in many programming languages, but there are cases where they should not be allowed.

A node whose submode is constrained to IN or OUT is represented by a `gnode` process, which contains four of the above seven arguments: 1, 2, 3, and 7.

[2] P_{Atom} does not contain an empty path ϵ, but we could extend the domain of modes to $P_{Atom} \cup \{\epsilon\}$ and let $m(\epsilon) = in$.

Operating on Mode Graphs All operations on mode graphs are provided as messages to the root node, which are delegated to appropriate processes. The delegation of messages corresponds to the dereferencing of pointers in procedural languages.

Operations (i.e., messages) accepted by a node process include the following:

1. examine the state of the node,
2. instantiate the mode value of the node,
3. add a new outgoing arc to the node,
4. create and return a new input stream to the node,
5. unify the node with another node,
6. examine the state of the (sub)graph rooted at the node, and
7. forward the above requests to an offspring node.

Operation 4 corresponds to pointer assignment in procedural programming.

Unification of two nodes is a 'symmetric' operation that the object-oriented programming style is not very good at. The operation is divided into two phases: It first accesses one of the nodes to obtain an input stream to it, and then sends a `unify_with` message to the other node. Care must be taken so that the second phase does not start until the first phase reaches the target node. Otherwise the second phase may reach its target node earlier, blocking the first phase to be delegated to its target node. This was actually the most awkward error we first made in implementing the constraint solver.

Thanks to the monotonicity of the constraint framework, however, the first and the second phases of the unification operation can be intervened by another operation.

The `unify_with` message takes the input stream to, and the identifier of, the partner node to its own target node. If the two nodes turn out to be the same, unification simply succeeds or fails depending on whether the polarities of the two paths are the same or not. Otherwise it examines the state of the partner node. If the two nodes have compatible mode values, they are unified by (1) merging the input streams of the nodes and directing the result to one of the nodes, (2) terminating the other node, and (3) merging the two sets of outgoing arcs, unifying corresponding arcs from each set recursively.

Retrieval of Graph States Retrieval of the state of the whole graph is not straightforward due to the circularity. We have prepared two messages for the purpose: `examine` and `examined`. The `examine` message is propagated over the graph, splitting itself at nodes with two or more outgoing arcs and turning the flag of each node on, until it reaches nodes with the flags on, and collects the states of the nodes using difference lists. The `examined` message is simply for turning off all the flags. We could dispense with the `examined` message by employing toggling flags rather than set-reset flags.

For a snapshot of the dynamically changing cyclic graph to be meaningful, no messages that may alter the graph should be issued before the processing of `examine` terminates. This will not cause a performance problem because snapshots will not be taken so often.

Termination Termination of a circular process structure turned out to be not so straightforward. Stream closing corresponds to the removal of a pointer in procedural programming. However, because of the circularity, propagating the stream closing operation from the root node to child processes is not enough to close all the streams and thus terminate all the node processes. So we decided to have all the node processes share a termination variable, which will be instantiated to `abort` when the graph is to be terminated.

However, each node process cannot simply terminate itself when it finds that the termination variable has been instantiated. Since the input stream of a node has a sophisticated protocol that may require backward communication, the node cannot discard it freely but must wait until it is closed: This is an example of the '*data-as-resource*' moral enforced by strong moding. Upon instantiation of the termination variable, each process closes its own output streams, waits until its input stream is closed, and only after that it terminates gracefully.

4.3 An Experiment—Analyzing the Mode Analyzer

As an experiment, we analyzed the constraint solver of our mode system, which had 190 clauses.

Those 190 clauses imposed 2464 constraints in total, which were classified as Table 1 according to the forms of the constraints and the rules that imposed the constraints. "Built-in" stands for the constraints imposed by calls to built-in predicates.

The most remarkable thing about these statistics is that, of 1392 constraints imposed by Constraint (BV), more than 90% were of the form $m/p_1 = m/p_2$ or $m/p_1 = \overline{m/p_2}$. Thus we can say that the clauses analyzed are highly linear. Only 5% of the variables were singletons, and 3% had more than two occurrences and imposed non-binary constraints. 2% of the variables had their values examined in guards, for which Constraint (BV) were weakened to a form $m(p) = \mathit{in}$ [7].

All of the 42 non-binary constraints were reduced to unary or binary constraints using other unary or binary constraints. Actually they were reduced to 6 constraints of the form $m/p_1 = \overline{m/p_2}$ and 72 constraints of the form $m/p = \mathit{IN}$. This means that non-linear variables were all used under simple, unidirectional communication protocols.

The final mode graph contained 162 nodes and 938 arcs. Table 2 shows the depth of the nodes from the top node. Considering that the program analyzed used quite complicated protocols, this result suggests that mode graphs are, in general, very shallow and wide. The program used complicated protocols (e.g., streams of messages containing other streams), but it defined various local procedures that had access to and handled various parts of the protocols. Mode graphs obtained by larger programs will be wider due to many top-level features corresponding to predicate arguments, but they will not be too deeper.

Type	Rule	Number of constraints
$m(p) = \mathit{in}$	(BF)	453
	(HF)	288
	(GV)	54
	(BV)	24
	Built-in	22
$m(p) = \mathit{out}$	Built-in	18
$m/p = \mathit{IN}$	(BV)	69
	(GV)	2
	(HV)	4
$m/p_1 = m/p_2$	(BV)	1074
$m/p_1 = \overline{m/p_2}$	(BV)	183
	(BU)	231
Non-binary	(BV)	42

Table 1. Constraints imposed by the mode constraint solver.

Level	Number of nodes
0	1
1	124
2	22
3	15
> 3	0

Table 2. Depth of the nodes of the mode graph of the mode constraint solver.

Of the 938 arcs, 814 went from the top-level node and were labelled with predicate arguments, while 124 went from non-top-level nodes and were labelled with function arguments. Of the 814 arcs sharing 124 level-1 nodes, 462 were by polymorphic unification body goals and 36 were by arithmetic goals which were also polymorphic. The remaining 316 arcs were those corresponding to the arguments of user-defined predicates.

Of the 161 non-top-level nodes, 122 had no outgoing arcs and the remaining 39 nodes had the total of 124 outgoing arcs. The node representing the path of the messages to the input stream of the predicate `node` had 54 outgoing arcs, while the other 38 nodes had less than two arcs on average. This means that hash tables should be used for maintaining the set of outgoing arcs of the top-level node, while simpler data structures can be used for other nodes.

4.4 Parallelism

Although mode analysis is not a highly computation-intensive task, it is worthwhile to explore the possibility of parallel speedup. One prominent feature of our mode system is that inter-procedure global analysis is done simply as incremental constraint solving which has much potential for parallel execution.

The first phase of mode analysis, constraint generation, is a highly parallel task because each clause yields its own mode constraints independently.

The second phase, constraint solving, is worth closer look. An important advantage of the constraint framework is that constraints can be merged in any order. Moreover, in our case, the order will not affect the performance too much. Thus the simplest and the most practical way of parallel execution is to exploit coarse-grain parallelism by creating a mode graph for each procedure or each module independently and merging them later.

Fine-grain parallelism that could be obtained by the pipelined processing of messages is subtler. Firstly, as we saw in Section 4.3, mode graphs are not deep anyway. Secondly, as we saw in Section 4.2, unification of two nodes imposes certain sequentiality. However, it is still important for mode graphs to be able to process messages concurrently, because imposing too much sequentiality between messages leaves less freedom (on the part of implementation) in the scheduling of message handling and can lead to lower sequential performance. Optimizing compilers may well exploit independence of primitive operations to gain performance.

Fortunately, in most cases, a message can be sent to mode graphs before the previous message finishes processing. Because a mode graph becomes "constrained" monotonically, concurrent instantiation of nodes and concurrent unification of nodes will not lead to an incorrect state as long as the atomicity of primitive operations such as the instantiation of node values and the merging of nodes is guaranteed. A message may enter a mode graph even if the previous one causes a mode error. Its effect is simply that when some message causes and reports a mode error, subsequent messages may already have given additional constraints to the mode graph.

5 Extension of the Mode System

5.1 Arrays

There have been a number of proposals of mutable array constructs that hide side effects at the language level but exploit them in implementation; some early work include [2], [3], and [4]. Not a few symbolic languages lack array constructs, but they are essential in many real-life applications.

KL1 supports several built-in operations for accessing and updating the elements of vectors and compound terms. In general, the semantics of built-in predicates can be explained by means of a possibly infinite number of virtual clauses, and the principal modes of built-in predicates can be obtained by considering the mode constraints imposed by those virtual clauses.

For array constructs, mode analysis tells us that the most basic element access operation, namely the operation that has the most general principal mode, is

```
set_arg(I, T0, X0, X, T).
```

This operation receives an index value `I` and a (compound) term `T0`, and returns through `X0` the `I`th value of `T0`. In addition, it returns through `T` a compound term which is identical to `T0` except that the `I`th element is replaced by `X`. A similar operation is defined for vectors as well [10]. Figure 6 shows the operation and the mode graph of `set_arg`, where $\langle ?, ? \rangle$ stands for a wildcard that matches any feature. Note that different elements of an array are constrained to have identical modes, but the mode of the elements itself is not constrained at all.

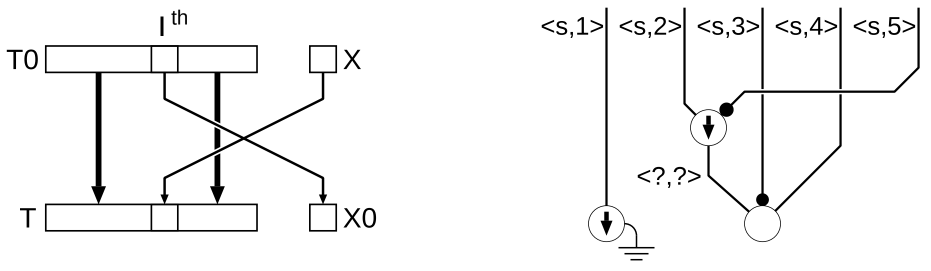

Fig. 6. The predicate `set_arg` and its mode graph.

Strong moding is deeply concerned with the number of access paths to each variable. As a result, data structures have an aspect of *resources* in general, whose access paths should not be copied or discarded freely. As can be seen in Figure 6, an array element should, by default, be removed from the array once accessed, and the resulting blank should be filled with another value.

Array creation is another fundamental operation. In Prolog, `functor` initializes the arguments of the created structure with distinct fresh variables, which are *instantiated* afterwards if necessary. However, strong moding tells us that the arguments should be initialized with constants and be *updated* by `set_arg`.

Strong moding is particularly important in array processing because it may enable update-in-place. Let the nodes of a mode graph have a *shared/non-shared* flag as well as mode values. *Shared/non-shared* means that the paths ending at the node may/won't be used for one-to-many communication, respectively. To see at which paths shared data may occur, we set the flags of all the nodes at the paths where non-linear variables occur, and of all the nodes below the nodes with the flags on. Then we can see what paths will be used only for one-to-one communication, and arrays occurring at those *non-shared* paths can be updated in place.

Aliasing is recognized as an awkward phenomenon in procedural programming, in which `a[`e_1`]` and `a[`e_2`]` may or may not denote the same variable depending on whether e_1 and e_2 evaluate to the same value. However, accessing two elements of a non-shared array using `set_arg` will not create new, implicit sharing. To access the `I`th element and the `J`th element of an array `A`, one will call `set_arg` twice:

```
set_arg(I, A, AI, AInew, A1), set_arg(J, A1, AJ, AJnew, A2).
```

The array `A1` does not contain the original `I`th element any more, so `AI` and `AJ` cannot be the same unless `AI` and `AInew` happens to be the same. However, `AI` and `AInew` cannot be the same as long as the array elements occur only at *non-shared* positions. For them to be the same, `AI` must occur in a goal for equating it with `AInew` in addition to the occurrence in `set_arg`, but then, we cannot 'use' `AI` through its third occurrence because it does not exist by the '*non-shared*' assumption. `AI` and `AJ` could be the same if one replaced `A1` in the second call by `A`, but then the array itself would become a shared array.

5.2 Polymorphic Modes

A unification body goal is polymorphic in the sense that its different occurrences in program text may have different modes as long as they obey Constraint (BU). Constraint (BU) here is considered to represent the 'principal mode scheme' for unification, and different occurrences may have different instances of it.

Array operations, stack processes, and stream merging are examples of generic programs in the sense that they do not constrain the modes of elements. Different arrays, stacks, or stream mergers should be able to accept elements with different protocols, where the necessity of polymorphic modes arises.

Although not yet implemented, polymorphism could be incorporated easily. For polymorphic predicates, their principal mode schemes (i.e., mode graphs) are computed first. To allow different instantiations of a principal mode scheme, a *copy* of the mode graph representing the principal mode scheme will be created for each call to a polymorphic predicate, which will be merged into the mode graph of the whole program. (In the monomorphic case, the original graph of the predicate is simply merged into the mode graph of the rest of the program.)

It seems that polymorphic predicates should be declared so in some way. Such a distinction was done also when introducing type polymorphism into functional languages. In ML, for instance, $(\lambda x.A)E$ and `let` $x = E$ `in` A have different meanings if types are taken into account.

The above treatment of polymorphism requires that the mode schemes of polymorphic predicates be obtained before analyzing the rest of the program that uses the polymorphic predicates. So polymorphic predicates should be *stratified* so that mode analysis can start from the 'most polymorphic' predicates that depend on no other predicates and end with the analysis of the whole program.

5.3 Higher-Order

There are two possible ways to allow a goal to dynamically determine the predicate to be called: One is `call` (analogous to `eval` in Lisp) and the other is `apply`.

Let `call(`G`)` be a goal that interprets G as a goal (by interpreting the principal function symbol as the predicate to be called) and executes it. The moding of `call` is straightforward; it simply imposes the constraint $m/\langle \texttt{call}, 1\rangle = m$.

In contrast, `apply` needs extension to the mode system. Suppose `apply(`P`,` X`,`Y`)` is a goal that executes a binary predicate P with the arguments X and Y. P may be either a function symbol representing a predicate to be called, a list of clauses (in which bound variables are represented by constants), or a compiled code with mode information. In either case, P is a ground term, but should have a mode as a predicate as well. The mode of `apply` could be represented as the left graph of Figure 7. Here, dotted lines represent the constraint that, when the first argument of `apply` is interpreted as a program, the first/second argument of that program must have the identical mode as the second/third argument of `apply`, respectively.

Then consider a predicate that applies P to X twice:

```
twice(P,X,Z) :- apply(P,X,Y), apply(P,Y,Z).
```

If `apply` is monomorphic, applying Constraint (BV) to the variables `X`, `Y`, and `Z` will result in the right graph of Figure 7 (where the constraints on `apply` is omitted).

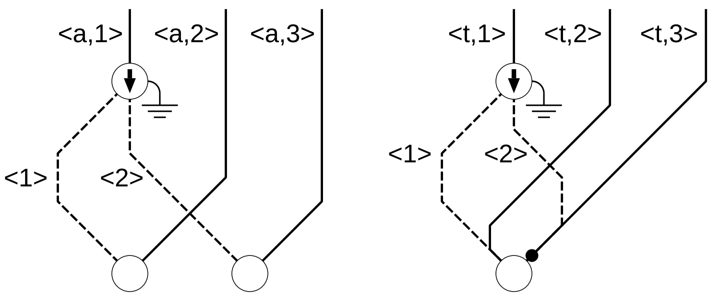

Fig. 7. Moding higher-order predicates.

5.4 Non-Herbrand Constraint Systems

Concurrent constraint programming generalizes concurrent logic programming by allowing data types that are not based on syntactic equality over the Herbrand universe (set of finite ground terms). Here we consider three extensions.

Rational terms: Our path-based mode system can quite naturally deal with non-finite rational terms. Readers may have noticed the similarity between the way our mode is defined and the way infinite trees are represented as functions.

Numerical constraints: Numerical constraints can be dealt with in a moded framework if dataflow can be determined statically. For instance, if the constraint goal `X = Y+1` is used always for determining `Y` from `X` or always for determining `X` from `Y`, it can be moded. However, dataflow caused by solving simultaneous equations will not be that simple in general.

Equational theories: Syntactic equality can be replaced by various equational theories, and a lot of work has been done on unification under equational theories. Here we focus on simple built-in theories; associativity, commutativity and idempotency.

Associativity and commutativity have the property that rewriting based on those preserve the number of occurrences of symbols. Actually they can be included naturally into the mode system. For instance, bags (multisets) enjoy the properties $t_1 \cup t_2 = t_2 \cup t_1$ and $t_1 \cup (t_2 \cup t_3) = (t_1 \cup t_2) \cup t_3$. So the paths where bags may occur should obey the constraint shown in Figure 8. That is, any subterm of a bag whose parent symbols are all $\cup$ (the bag constructor) must have an identical mode whose top-level is *in*.

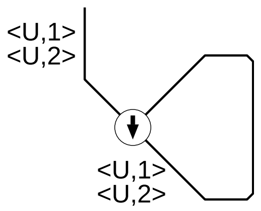

Fig. 8. Constraints imposed by an associative and commutative operator $\cup$.

On the other hand, idempotency ($t = t\ op\ t$) says that terms can be freely copied and two identical terms can be freely contracted. This is not very compatible with the data-as-resource view, and any path which is an inlet/outlet of an idempotent operator will be constrained to *IN*/*OUT*, respectively.

6 Related Work

Tick and Koshimura implemented and compared several algorithms for mode analysis [8]. One of them uses process structures to represent mode graphs, but their mode graphs have many differences from the graphs in [5] and [7]. They form 'initial mode graphs' and minimize them to form the final graphs, while we add many simple constraints (in the form of messages) to an empty mode graph to form the final graphs. Their graphs deal with non-binary constraints, while we delay non-binary constraints to avoid complication. Their graphs require additional information called 'partition node sets' to maintain the node identifiers, which we need not have. They implement the unification of cyclic structures using marking, while we dispense with marking by implementing it using incremental redirection of streams.

7 Conclusions

We implemented a mode analyzer for Flat GHC and KL1, itself described as a strongly moded KL1 program. The analyzer was applied to the analyzer program itself, which used fairly sophisticated communication protocols, to see if automatic mode analysis worked well for non-trivial KL1 programs. We also discussed how the current moding framework could be extended to deal with random-access data structures, polymorphic modes, higher order, and general constraint systems.

Our implementation of the mode system employs quite sophisticated process structures, namely feature graphs with cycles and node sharing. Concurrent operations on such data structures involve nondeterminism and make programs harder to debug. Also, operations on graphs include a rather unusual operation: the merging of two nodes. We succeeded in describing all these in a strongly moded framework and made sure that strongly moded concurrent logic programming was expressive enough for quite complicated programs. Rather, we benefited much from the mode system in debugging.

In spite of complicated process structures formed, debugging was not so difficult. Bugs had to be found manually at first, but most of them were those which could be detected by mode analysis. Many of the bugs we removed later were detected by the mode analyzer itself.

Not all bugs were identified easily, however. The most awkward one was perpetual suspension resulting from the misunderstanding of causality between messages handled concurrently in a nondeterministic program. However, exploiting concurrency is important both for parallel execution and efficient sequential execution.

Self-application of the mode analyzer has confirmed our conjectures that (1) most variables are used for one-to-one communication (i.e., are linear) and (2) non-binary constraints will be reduced to unary/binary constraints. However, these points require further study because programs written by other people may be less linear.

For the mode system to be more practical, it should generate a user-friendly error messages to non-well-moded programs. The current system simply reports the constraint that finally caused inconsistency and the mode graph immediately before the inconsistency was detected. However, a more user-friendly system should find as concise and intuitive an *explanation* of inconsistency as possible. A forthcoming paper will report an algorithmic approach to the diagnosis of non-well-moded programs.

References

1. Aït-Kaci, H. and Nasr, R., LOGIN: A Logic Programming Language with Built-In Inheritance. *J. Logic Programming*, Vol. 3, No. 3 (1986), pp. 185–215.

2. Cohen, S., Multi-Version Structures in Prolog. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, ICOT, Tokyo, pp. 265–274.
3. Eriksson, L.-H. and Rayner, M., Incorporating Mutable Arrays into Logic Programming, In *Proc. Second Int. Logic Programming Conf.*, Uppsala Univ., Sweden, 1984, pp. 101–114.
4. Ueda, K. and Chikayama, T., Efficient Stream/Array Processing in Logic Programming Languages. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, ICOT, Tokyo, pp. 317–326.
5. Ueda, K. and Morita, M., A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, The MIT Press, Cambridge, MA, 1990, pp. 3–17.
6. Ueda, K. and Chikayama, T., Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494–500.
7. Ueda, K. and Morita, M., Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, Vol. 13, No. 1 (1994), pp. 3–43.
8. Tick, E. and Koshimura, M., Static Mode Analyses of Concurrent Logic Languages, ICOT Tech. Report TR-875, ICOT, Tokyo, 1994. Also to appear in *J. Programming Languages Design and Implementation.*
9. Ueda, K. and Morita, M., Message-Oriented Parallel Implementation of Moded Flat GHC. *New Generation Computing*, Vol. 11, Nos. 3–4 (1993), pp. 323–341.
10. Ueda, K., I/O Mode Analysis in Concurrent Logic Programming. In *Proc. Int. Workshop on Theory and Practice of Parallel Programming (TPPP'94)*, Ito, T. and Yonezawa, A. (eds.), LNCS 907, Springer, 1995, pp. 356–368.

This article was processed using the LaTeX macro package with LLNCS style