A system of precise modes for logic programs

Zoltan Somogyi

Department of Computer Science
University of Melbourne
Parkville, Victoria 3052
Australia

Abstract

Two desirable properties of a mode system are precision and the ability to handle partially instantiated data structures. We present the first system that has both these properties. It analyzes binding information in terms of the smallest possible unit and relies on a type system to keep binding patterns regular.

1. Introduction

Many logic programming languages have features related to the pattern of instantiation of a procedure's variables. These features range from the purely advisory (DEC-10 Prolog [Bowe82]) through the meaningful but optional (MU-Prolog [Nais85b], NU-Prolog [Thom86], and Concurrent Prolog [Shap83a]) to the compulsory (Parlog [Greg85]). The two mechanisms at the extremes of this list have in common only the name *mode declaration*; Parlog tries to be precise much harder than the other languages.

Our mode system lies at the Parlog end of this scale. It has the same objective as Parlog: to impose structure on the communication patterns of stream AND-parallel logic programs. Like Parlog, we use compile-time analysis to enforce mode-correctness and base our analyses on declarations attached to procedures. Unlike Parlog, we consider the fundamental binding units to be unifications of the form $\texttt{x = f(y}_1\texttt{, ..., y}_n\texttt{)}$ where the $\texttt{y}_\texttt{i}$ are distinct variables. We build up other, more complicated bindings by combining these units. For example, given the bindings `x = f(y)`, `y = g(z)` and `z = a`, we can combine the first two to get `x = f(g(z))` or we can combine them all to get `x = f(g(a))`. This exposes the natural progression from unbound variables through partially instantiated data structures to ground terms.

The two end-points of this progression lend themselves to easy analysis. The middle, however, is quite chaotic. When a procedure produces a partially instantiated data structure a precise system should know not only this fact but also *where* in the term the unbound variables are. For this the system must know the possible structures of terms. Static type systems provide exactly this information. This paper shows how to use one such system to impose regularity on the chaos and achieve precision in the handling of partially instantiated data structures.

Our mode system is not intended to be the only control mechanism of a language, but it certainly is intended to be the framework within which other control mechanisms (such as sequential AND) should operate. Our proposal is fairly independent of the specifics of these other control primitives, but we should note that our vision of the parallel logic programming language of the future does not include a *compulsory* committed-choice mechanism.

The structure of the paper is as follows. In sections 2 and 3 we define types and modes respectively. Section 4 gives an outline of the mode analysis algorithm, and section 5 describes some parts in detail. Section 6 discusses some implications of our proposal (focusing on software engineering) while section 7 compares it to other systems.

2. Types

Our type system is a static one with Milner-style polymorphism [Miln78]. It is virtually identical to those of the functional languages HOPE [Burs80] and ML [Card84], and to the system described in [Mycr84]. A *type* T is either a *type variable* or a *rational AND/OR tree*, written as an expression of the form

$$T = f_1(T_{11}, \ \ldots \ T_{1m_1}) \text{ or } \ldots \text{ or } f_n(T_{n1}, \ \ldots, \ T_{nm_n})$$

The f_i are distinct function symbols (constants if $m_i = 0$, in which case we omit the empty pair of parentheses); the T_{ij} are themselves types that may be syntactically equal to T. Intuitively, an AND node like $f_1(T_{11}, ..., T_{1m_1})$ is a template for terms while an OR node is a type; variables of the type will take a term matching one of the templates as their value.

We require all our programs to be well-typed. [Mycr84] defines type-correctness for systems like this; in any case the notion should be intuitively obvious. The type of lists whose elements are of type T is

```
listof[T] = nil or cons(T, listof[T])
```

where T is a type variable. Figure 1 shows the tree corresponding to this definition. The figure also has a label on each OR-node; these labels are used in the next section.

We call the set of type trees of the arguments of a procedure the procedure's *type forest*. Together with the type trees of the local variables of a clause, they form an *extended type forest* for that clause.

3. Modes

Parlog has a concept of *directional* logic programs, which are functional programs in a logic setting. These programs have the elegant property that for each variable the compiler knows precisely which call will bind it. However, according to the definition in [Greg85], a Parlog program in which different parts

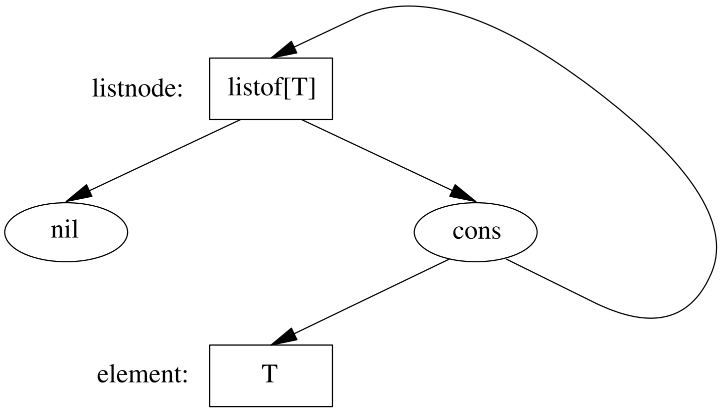

Figure 1.

of a term are constructed by different calls is not directional. In particular, no program that uses the paradigm of incomplete messages [Shap83b] is directional.

Our system is based on a notion that captures the essence of directionality but which is more widely applicable: the compiler should know precisely what call will bind *the main functor* of each variable. This definition allows two or more calls to cooperate in constructing a data structure as long as the compiler knows which call instantiates which part. As a result, our mode system needs no escape mechanisms like Parlog's *weak input arguments*, and we include none. In our system a *mode* is an assignment of a tag from the set {INBOUND, INFREE, OUTBOUND, OUTFREE} to all the OR-nodes[1] of the type forest of a procedure.

The tag of a variable `x` in a procedure `p` governs what actions `p` and the procedures it calls are allowed to do with `x`. The meanings of the tags are as follows:

INBOUND

If `x` is tagged INBOUND then neither `p` nor any procedure it calls is allowed to instantiate the main functor of `x`. If they try to do so by executing `x = t` for some non-variable term `t`, that unification will be delayed until `x` is instantiated to a non-variable by a unification outside the call-tree of `p`.

INFREE

If `x` is tagged INFREE then neither `p` nor any procedure it calls will touch `x` in any way except by unification with a void variable[2]. A node with an

---

[1] Later we will refer to OR-nodes as simply *nodes* if there is no risk of confusion.

INFREE tag must have a parent OR-node with an INBOUND tag.

OUTBOUND

If `x` is tagged OUTBOUND then it will have its main functor bound by a unification within the call-tree of `p`. The mode system imposes the restriction that no unification outside the call-tree is allowed to do likewise (this is the topic of the next section).

OUTFREE

If `x` is tagged OUTFREE then `x` will be created within the call-tree of `p` and will be exported from `p` without any bindings. The parent of a node with an OUTFREE tag may not have an INBOUND tag.

The semantics of the INBOUND and OUTBOUND tags are similar to Parlog's strong input and output modes; the difference is that they apply only to one functor. The INFREE and OUTFREE tags are new: they represent unbound variables imported and exported by a procedure. Given this semantics, the tags of nodes reachable only from FREE nodes don't have any meaning, so it seems reasonable to forbid their names from appearing in modes.

Mode declarations are written as

```
inbound ? infree => outbound ! outfree
```

where `inbound` is the list of the names of the nodes with an INBOUND tag, and similarly for the others. Node names consist of two parts: the name of a parameter (to identify its tree) and the label of a node in that tree. The `=>` is called the *mode implication*; the `?` and the `!` are *class separators*. If there are no INFREE (or OUTFREE) nodes, the corresponding class separator is dropped; if there are no OUTBOUND and no OUTFREE nodes, the mode implication is dropped as well[3].

Two conventions make mode declarations shorter. One, the second part of the name may be omitted for root nodes. Two, if a node's name doesn't appear in any list, the compiler will (recursively) find the tags of its immediate ancestors and if they all agree, assign that tag to the node. The following program is an adaptation for our system of the standard Prolog code for `length`. The syntax is *not* necessarily the one the programmer would use; it is an intermediate language whose details are not important.

---

[2] A variable is *void* if it occurs only once in a clause. Unifications with void variables always succeed without effect; they are necessary because of a technicality (see the next example).

[3] This notation and terminology is derived from [Port83].

```
procedure length (list: listof[T]; n: integer⁴)
local
      list1: listof[T]; n1: integer; head: T;
mode
      list ? list.element => n;
      n => list ! list.element;
def
      list = nil & n = 0
or
      list = cons(head, list1)
      & length(list1, n1)
      & plus(n1, 1, n)
end length
```

The mode declarations state that `length` may be used in two ways. One is to give `length` a list whose elements may be uninstantiated, in which case `length` will compute its length; the other is to give `length` a number, in which case it will return a list of that length with unbound elements. These modes correspond to the programmer's intuition about the reasonable uses of the `length` predicate.

In its first mode procedure `length` uses a partially instantiated data structure; in its second mode it constructs one. In both cases it unifies the variables of the FREE nodes with the void variable `head`. In the first mode the purpose of the call `list = cons(head, list1)` is to get the value of `list1`. The unification of `head` with the anonymous INFREE variable representing the first element of `list` is a mere syntactic formality. In the second mode the purpose of the call is to construct the value of `list`; the unification *creates* the anonymous OUTFREE variable.

A tag given to the node of a type variable has a special meaning. Since procedure `length` does not know the structure of the type U which T takes as its value, it must invoke convention two and assign the tag of the node of T to all the nodes in the tree of U. This means for example that `length` must treat `list.element` as atomic: if a mode of `length` gave `list.element` the tag OUTBOUND, this would obligate `length` to instantiate the list elements to ground terms.

4. Mode assignments

A *mode assignment* for a procedure `p` in mode `m` is an assignment of one of the modes of the called procedure to each call in the body of `p`. A mode assignment is *valid* iff in each conjunction in the disjunctive normal form of the body of `p`

---

[4] Conceptually, the tree of the type `integer` has one OR-node (the root) and an infinite number of AND-nodes, one for each integer.

(1) the INBOUND nodes of arguments of `p` appear only in INBOUND or INFREE positions of the called procedures;

(2) the INFREE nodes of arguments of `p` appear only in INFREE positions;

(3) the OUTBOUND nodes of arguments of `p` and some of the nodes of the type trees of the local variables appear exactly once in an OUTBOUND position - they may appear once in an OUTFREE position and any number of times in INBOUND and INFREE positions;

(4) the OUTFREE nodes of arguments of `p` and all other nodes of the type trees of the local variables appear exactly once in an OUTFREE position - they may appear any number of times in INFREE positions[5].

The special procedure = provides the basis for this recursive definition. `u = v` itself has three modes: `u => v` and `v => u` (the assignment modes) and `u, v` (the test mode). These apply when neither `u` nor `v` is a void variable. If `v` *is* void, the mode can be either `? u => ! v` or `=> ! u, v` and similarly for a void `u`. Each of these modes applies to a single functor; the recursive cases can have other modes, as shown in the next example. The procedure `length` above has one valid mode assignment in each of its modes. In the first mode, `list = nil` is a test, `n = 0` is an assignment to `n`, the recursive call `length(list1, n1)` has the mode `list ? list.element => n` and the call `plus(n1, 1, n)` on the procedure `plus(a, b, c)` has the mode `a, b => c`. The first level of `list = cons(head, list1)` is a test, while the second level has a void unification and an assignment to `list1` (this is explained further in section 5.2).

None of the above modes implements the full unification algorithm. They implement one-way unification, or *term matching*, which is more efficient (though efficiency is not our primary purpose). A procedure is *well moded* (or *mode correct*) iff in all its modes it has at least one valid mode assignment[6]. A program is well moded iff all its procedures are well moded.

It is the programmer's job to specify each procedure's modes. It is the compiler's job to find a valid mode assignment for each mode of each procedure, thus proving that the program is mode correct and providing information needed by the run-time system. We now present the algorithm the compiler uses to do this. It is a modified form of the one in [Port83] and bears some resemblance to one used in the old version of Parlog. Here we give only an outline; the loose ends will be cleaned up in the next section.

---

[5] Note that this definition treats local variables just as it treats output arguments; the difference is that they don't appear in mode declarations and their values are not exported to the caller.

[6] If it has more than one, the choice among them affects only the efficiency, but not the semantics, of the procedure.

This version of the algorithm works for programs with definition bodies in disjunctive normal form. It examines each conjunction of each mode of each procedure independently. It bases its analysis on a *known nodes list*. A node of the extended type forest is on this list if the algorithm has already determined its producer. At the start the list consists of the INBOUND nodes of the procedure. The algorithm then looks for producers of the other nodes in stages. At each stage it selects a feasible call[7] and adds its OUTBOUND nodes to the known node list. A call is *feasible* if the procedure being called has a mode such that at each of the mode's INBOUND nodes the call has either a manifest functor or a variable on the known node list. (For a negated call that mode must have no OUTBOUND nodes and hence no OUTFREE nodes.) The algorithm terminates when all the calls have been selected. It signals an error if

(1) the mode has an OUTBOUND node which does *not* appear on the known nodes list;

(2) the mode has an INFREE or OUTFREE node which *does* appear on the known nodes list;

(3) it can't find a feasible call while some calls are still unpicked (but see the section below on circular modes).

Our mode system does not depend on definitions being in disjunctive normal form. We restricted our description to that case because a slight complication can otherwise arise. Consider the program fragment

```
... & { p(x, y, z) or { q(x, y) and r(y, z) } } & ...
```

If procedure `p` has mode `x => y, z`, procedure `q` has mode `x => y` and procedure `r` has mode `y => z`, the algorithm has to add the OUTBOUND nodes of `p` to the known node list in two passes, once together with the outputs of `q` and once with the outputs of `r`. A forthcoming technical report will present a detailed algorithm which solves this problem and accommodates the points raised in the next section. We trace the algorithm in its analysis of the `append3` predicate [Vasa86], which is canonically defined as

[7] When there is more than one feasible call, the compiler is free to select any of them. This point is examined further in section 5.4.

```
procedure append3(a, b, c, d: listof[T])
local
      e: listof[T];
mode
      a, b, c => d;
      d => a, b, c;
def
      append(a, b, e) & append(e, c, d)
end append3
```

where `append(x, y, z)` has the modes `x, y => z` and `z => x, y`. For the first mode of append3, the known node list starts out containing `a`, `b` and `c` and all their descendants (the elements of the lists). At this stage only the first call is feasible (in the first mode of `append`), so we add its output nodes (`e` and its descendants) to the list. Now the second call becomes feasible as well (also in the first mode of `append`), and after adding its output nodes (`d` and its descendants) to the list the algorithm terminates. It has determined that in its first mode `append3` will use the first mode of `append` for both its calls. Similarly, for the second mode of `append3` both calls will use the second mode of `append`.

5. Variations on a theme

5.1. Implied modes

The algorithm outlined above does not say what happens when the call has a manifest functor or a known variable in a position where the called procedure has an OUTBOUND tag. Given that `length(list, n)` has the mode `n => list ! list.element`, it seems obvious that the system should allow calls like `length(cons(x, cons(y, nil)), 2)`. Formally we say that one can always shift nodes from the right hand side of the mode implication to the left (from OUTBOUND to INBOUND) provided that one shifts any orphaned OUTFREE descendants as well[8]. Modes created this way are *implied modes*. Modes not implied by other modes are *principal modes*. For unification, the assignment modes are principal modes while the test mode is an implied mode (implied by both the other modes).

In theory, a mode with n outputs can imply $2^n - 1$ other modes. In practice, however, there is no need to manage the exponential explosion. Most procedures have only a small number of outputs, and few of the implied modes actually get used because people tend not to write procedures that are more general than necessary.

---

[8] I.e. the mode of that call is `n, list ? list.element`; not `n, list => ! list.element`.

The exception to this is the mode required of negated calls (they must have *all* the OUT nodes shifted to the left). This requirement is sufficient to ensure soundness but it is not as restrictive as the standard safety condition which requires negated calls to be ground; for example it allows `~length(x, 5)` to go ahead when the elements of `x` are not instantiated, whether or not they are going to be.

One can also shift nodes from INFREE to INBOUND. This makes no difference operationally; if a procedure does not require the value of a variable it won't wait for it even if it is tagged INBOUND. On the other hand, it is not possible to shift a node from OUTFREE to OUTBOUND, since OUTBOUND nodes must have a producer but OUTFREE nodes are not allowed to have one.

5.2. Unrolled type trees

The algorithm above assumes that the type forests of the head and the body are compatible. This is often not so. Consider the unification `list = cons(head, list1)` in the first mode of `length`. The system knows the first `listnode` of the right side (since it is a manifest constant), but it does not know the others. From the structure of `cons(h, list1)` it derives the type tree shown in Figure 2 for the right side of the unification. By the rules of infinite trees this type tree is equivalent to the standard one in Figure 1. It is generated from the standard one by *unrolling*, which is the replacement of the recursive arrow back up the tree with a copy of the pointed-to subtree[9].

Once the compiler has found the type tree of the right side of the unification, it unrolls the type tree of the variable on the left side to match. At this point the known node list contains `u.listnode1` (or simply `u`), `u.listnode2` and `v.listnode1` (as before, `u` and `v` are the names of the two arguments of the unification procedure). The compiler now issues instructions to perform a test for `listnode1`, assignments for all the `listnode2`'s, and void unifications for `element1` and the `element2`'s, just as we said in section 3.

5.3. Circular modes

The mode assignment algorithm tries to find a *partial order* in which the calls in a conjunction may be conceptually evaluated[10]. For programs which make use of incomplete messages, however, there is no such order. Consider the following program which implements a toy "fileserver" with one atomic piece of data:

---

[9] The name comes from the fact that one can view the loop in the original tree as an infinite branch "rolled-up" on a spool; "unrolling" releases one part of the branch from the spool.

[10] In reality the calls may be run in parallel, with the partial order representing only a communication constraint, just as in Parlog.

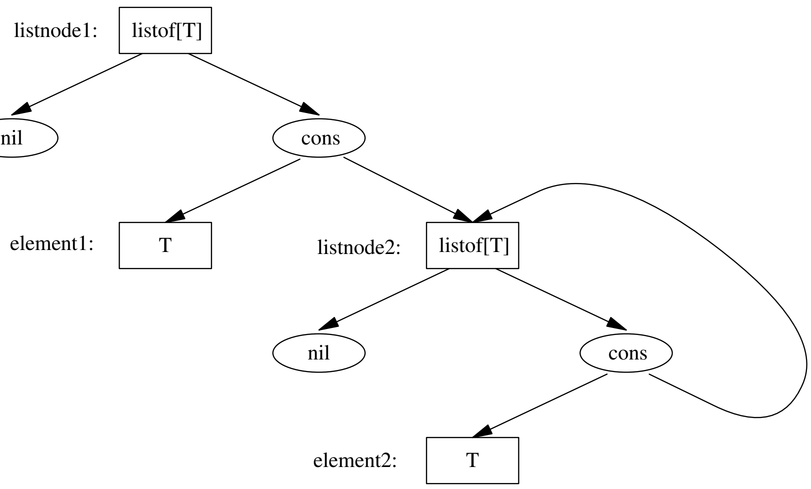

Figure 2.

```
type cmd[T] =
      read(indata: T) or write(outdata: T)
end cmd

procedure server(stream: listof[cmd[T]]; data: T)
local
      stream1: listof[cmd[T]]; data1: T;
mode
      stream, stream.element, stream.outdata
            => stream.indata;
def
      stream = cons(read(data), stream1)
      & server(stream1, data)
or
      stream = cons(write(data1), stream1)
      & server(stream1, data1)
end server
```

The client generates commands and the server carries them out. The channel between them is set up by the conjuction

```
server(stream, initial_data) & client(stream, ...)
```

The client's mode has to include

```
stream.indata => stream, stream.element, stream.outdata
```

which creates an apparent circularity: the server cannot execute until given a command and the client cannot generate commands until it is given `stream.indata` by the server. Intuitively, the second half of this is false, but how can the system know this ?

`stream.indata`, which is an INBOUND node of the client, has as its parent `stream.element`, which is an OUTBOUND node. The compiler should notice this and figure out by itself that the client does not need `stream.indata` to start executing. If the compiler *assumes* that the node `stream.indata` is known and lets the client execute, it will generate commands, and *then* the server will instantiate `stream.indata`, thus fulfilling the assumption.

Because data flows in both directions between client and server the example program is not directional in the Parlog sense. Nevertheless it still has the main property which makes directionality desirable: the server and the client will never conflict on whose job it is to bind the main functor of a given variable. That is why our definition of mode correctness allows such a possibility.

The compiler should make assumptions to deal with circularities only when told to do so by the programmer, either explicitly or via a *tag inversion* as above. Despite the necessity for circularities, there is a motive which should discourage their use. This is the fact that code called in a circular manner may need the occur check, while programs without circularities do not (Parlog has a similar but weaker property). However, many circular programs are spared the need for the occur check as well, because occur check failure cannot occur if the types of the data flowing in and out of a call are not identical. This observation can be put to good effect for most programs using incomplete messages, including the example.

5.4. Producer annotations

When a node can be generated by two or more calls, the mode analysis algorithm is free to choose any one of the candidates. Which one it selects should affect only the efficiency, not the correctness of the program. Of course, efficiency is often important, so the programmer should be able to control producer selection.

The classic example is query optimization for database programs. Suppose that `p` and `q` are database procedures[11], and that we wanted to use the code fragment `p(x, z) & q(z, y)` in the modes `x => y` and `y => x` with the help of the local variable `z`. Obviously only `p` can produce `x` and only `q` can produce `y`, but in each mode there are two candidate producers of `z`. In the `x => y` mode, `p` need only search a small part of its database to produce `z` (the part selected by `x`), while to do likewise `q` has to scan its whole database, most of which is likely to be irrelevant to the query. So in this mode `p` should produce `z`. For the other mode, exactly the same considerations lead us to prefer `q` as the producer of `z`.

---

[11] *Database procedures* can produce all their arguments without any input.

This is the reason why bare producer annotations in the style of the Relational Language [Clar81] do not work with multimode procedures. Therefore we allow the programmer to associate a name with each mode declaration; he can then quote the appropriate name(s) with his producer annotation. If the modes of our code fragment are named `forward` and `backward`, this annotation could take the form `p(x, z) [forward: z] & q(z, y) [backward: z]`. This notation has the advantage that it extends easily to arbitrary first-order formulae. For example, `{ p1(x, z) or p2(x, z) } [forward: z]` tells the compiler to proceed as if the disjunction were a procedure call and that call were designated the producer of `z`.

Incidentally, a program with annotations can simulate join AND-parallelism in our stream AND-parallel system [DeGr84]. Given two procedures `p(x, y)` and `q(x, y)`, both with mode `=> x, y`; the code `p(x, z1) [z1] & q(z2, y) [z2] & z1 = z2` has the same effect in our system as `p(x, z) & q(z, y)` does under join.

5.5. Heuristics for automatic producer selection

Often the programmer knows not only that a procedure like `append` can produce values for `z` given `x` and `y`, but also *how many* of these values it will produce. Based on this information the compiler should be able to make intelligent choices among alternative producers by itself, so the programmer should have a way of putting it into the program. In our system he can do this by writing `x, y => z: 1`. This says that `append` will always produce exactly one value for `z` based on any given values of `x` and `y`.

A count like this applies to an implied mode only in a limited sense. Given a mode `a => b: 1, c: 1`, we cannot be sure that `a, b => c: 1` is correct: if the value of `b` supplied by the caller differs from what would be computed by the procedure, no values for `c` would be computed at all. On the other hand, we *can* be sure that the procedure will produce at most one binding for `c`. This is expressed by `a, b => c: 0 to 1`.

In this example the count for the implied mode was less precise than the count for the principal mode. Sometimes it can be the other way around. `append` may have any number of solutions in mode `z => x, y`, but in the implied mode `x, z => y` it may not have more than one.

If all the OUTBOUND variables of a mode are given counts, the products of the lower and upper limits of the ranges establish limits on the behaviour of the call. In the terminology of [Smol84], a mode where none of the lower limits is zero is in effect a *totality assertion*, while a mode where all the upper limits are one denotes a *functional* mode. Such explicit declarations of totality and functionality make programs much easier to read. In addition, the properties themselves can be exploited by the implementation [Debr86a].

5.6. Types as generators

Some programs may use variables without any producers. The canonical example is `forall x: ~p(x)`. The mode analysis algorithm, however, has an ace up its sleeve: the type of `x` can itself act as a producer. The problem is that types usually have an infinite number of values, so the compiler should warn the user whenever it finds it necessary to use an infinite type as a producer. In fact, in that case it is probably a better solution to require the programmer to write something like `forall x: typegen(x) & ~p(x)`, as this ensures that the programmer is fully aware of the potential infinite loop.

6. Implications

Some critics would claim that our system is too restrictive. For example, append for difference lists cannot be well-moded because its instantiation patterns are not regular in our sense. Indeed we agree that our system takes some flexibility away from the programmer. The question is, how serious is this loss and is it adequately compensated for ? We contend that the loss is minor and compensation abundant.

We can say that the loss is minor because in our experience good Prolog programs *have* precise modes, even if they are not written down; those that do not are traps just waiting to spring on some unsuspecting programmer. For example, after a call to d-list append (whose standard definition is the unit clause `dappend(a-b, b-c, a-c)`) to concatenate `1.x-x` and `2.y-y`, one can no longer concatenate `1.x-x` and `3.z-z` because `x` has been bound to `2.y`. So the disappearance of `dappend` should not be mourned.

On the other hand, our modes do not rule out the *disciplined* use of difference lists. The following version of quicksort, written in Parlog, is a slight modification of one in [Clar86]. It is not well-moded by Parlog's standards; nevertheless it can be well-moded under our rules because there is an orderly flow of data from `remainder` to `frontsortlist` through `remplus`. Unlike Parlog, our system does not care whether difference lists are represented by two arguments or by the two substructures of one.

```
sort(list, sortlist) <-
      sortd(list, sortlist - []).
sortd(head.tail, frontsortlist - remainder) <-
      partition(head, tail, list1, list2),
      sortd(list1, frontsortlist - head.remplus).
      sortd(list2, remplus - remainder).
sortd([], list - list).
```

The only significant logic programming technique which cannot be easily adapted to precise modes is metaprogramming. This is a case of "Can it do this ? Yes, but not that way." We believe that the proper framework for metaprogramming is the context of higher-order functions/procedures. In this view, the compiler is merely a procedure-producing procedure which programs invoke just like they invoke other procedures. The fact that the type and mode of the procedure produced by the compiler has to be anticipated by the calling program is not a real restriction: von Neumann operating systems have lived with it for years[12].

While our system may have slightly less expressive power than other logic programming languages, it still has more than functional languages. One can see that it has at least as much when one considers that each mode of a predicate defines a function; and one can see that it has more when one considers that predicates may have more than one mode and that our system has no need for tagged streams to implement two-way communication [Hend82].

The compensation takes two forms. The first and more important is reliability. A typical program in our system can reasonably be expected to be more reliable than (say) its Prolog counterpart. The communication patterns between its procedures are more regular and a lot more explicit, and hence more readily understandable. The compiler has two extra sources of redundant information (types and modes) which it can use to find errors. Furthermore, our programs should be easier to debug; declarative debugging algorithms converge on bugs faster when they are based on binding information [Pere86].

Tradeoffs between flexibility and reliability are not new: the paper [Denn76] lists a long succession of such cases. The lesson of history is that reliability must prevail in the end.

The other form of compensation is the possibility of greatly increased execution speed. One source of this is intelligent dependency-directed backtracking: one could readily adapt an algorithm like the one in [Demb85] to our system. The other source is parallelism. A full discussion of the impact of precise modes on parallel architectures deserves another paper, but we point out one important fact. Because other systems have to consider the possibility that two procedures might try to bind a variable at the same time, their parallel implementation requires atomic read-write transactions. The huge number of these transactions would slow the system down considerably. In our system there is no possibility of conflict and the implementation can use ordinary read and write instructions[13].

7. Comparisons with other systems

Instantiation control systems have a number of uses. The important ones are

---

[12] The fact that this scheme gives the compiler special privileges (e.g. only the compiler can create procedures from text) also has a precedent in the Burroughs machines.

[13] This is not true in the extremely small minority of cases where a circular-mode program needs the occur check.

(1) to prevent bugs;

(2) to document the program;

(3) to help control the execution of the program;

(4) and to gain efficiency.

DEC-10 Prolog's mode declarations are not really declarative at all: the validity of DEC-10 mode depends on the entire program, not just on the procedure referred to. This lack is understandable, since the DEC-10 mode system is there exclusively for optimization. However, the overly flexible nature of Prolog interferes with this objective. Algorithms for the automatic inference of DEC-10 modes can only locate about two-thirds of the "interesting" modes [Mell81, Debr86b] even though they go to considerable lengths in an effort to obtain precise binding information.

The *wait declarations* of MU-Prolog [Nais85b] and the *when declarations* of NU-Prolog [Thom86] are control mechanisms whose main purpose is to stop infinite loops caused by insufficiently instantiated calls. Like DEC-10 Prolog modes, these declarations can be generated automatically [Nais85a], however they have a fairly high overhead and their declarative value is strictly limited. For example, the standard when declaration `append(A, B, C) when A or C` does not reflect the nature of `append`, which is symmetrical in the first two arguments.

The *freeze* predicate of Prolog II [Boiz86] is a general purpose control mechanism. Its semantics is actually quite similar to the semantics of INBOUND nodes. The principal differences are that freeze delays even those parts of the call which do not depend on the frozen variable and that freeze occurs in the body, thus making it hard to write multimode procedures.

The read-only annotation of Concurrent Prolog [Shap83a] is also a general purpose control mechanism. However, its non-declarative nature and the fact that Concurrent Prolog does not impose any structure on its use means that read-only annotations are very error-prone. Also, they intrinsically require a complicated unification algorithm.

Saraswat's input-only annotation avoids the most important problems of read-only annotations, but as explained in [Sara86], it cannot support *protected embedded channels* (i.e. one cannot use just input-only annotations to implement two-way communication between calls).

GHC [Ueda86] has no real instantiation control system at all. Its *rules of suspension* are purely operational and have no declarative meaning.

Parlog [Clar86] is the only language at present that shares our objectives and comes close to a complete and efficient mode system. In fact, for programs without partially instantiated data structures our modes and Parlog's are almost identical. Unfortunately, the mechanism of *weak input arguments* (which Parlog needs to handle partially instantiated data structures) destroys much of the effectiveness of the system.

We give two examples. Parlog's mode declarations do not state whether input arguments are strong or weak: input arguments are made weak by the code. This is bad from a software engineering point of view, and usually it also prevents a procedure from being used in more than one mode[14].

A much more serious problem is that weak input arguments make it impossible to decide in general whether a procedure will instantiate a variable. For example, the following Parlog program may or may not instantiate `x`, the choice being nondeterministic.

```
mode p(?).
p(f(x)) <- true.
p(f(x)) <- x <= a.
```

It is striking that none of these systems with the possible exception of Parlog really try to use their mechanisms to help document the program, although this has long been known to be useful [Bruy82].

8. Conclusions

Mode systems are desirable because they enforce good programming practices and because they enable an implementation to do statically what used to be done dynamically. To fulfill their promise, however, their rules must be cast in iron: the mere possibility of a violation can destroy the programmer's confidence in the system and necessitate the re-inclusion of elaborate run-time mechanisms. And as long as the scope of modes is any bigger than a single functor, programs using the paradigm of incomplete messages will find it *necessary* to violate the mode system.

We believe that giving up some flexibility for a solution of this problem is a good bargain. The flexibility we lose is needed only rarely but the reliability and efficiency we gain are useful always.

We would like to thank Ann Nicholson, Graeme Port, K. Rao and John Shepherd for their comments on drafts of this paper.

This research was supported by a Commonwealth Postgraduate Research Award and the Australian Computer Research Board.

References

[Boiz86] P. Boizumault, 'General model for implementing DIF and FREEZE', *Proceedings of the Third International*

[14] The version of Parlog described in [Clar83] allowed multi-mode procedures, but the current version does not. When they made this change the designers were probably influenced by the problem above and by the fact that multimode procedures are not very useful in a language with a compulsory committed-choice mechanism.

Conference on Logic Programming, Imperial College of Science and Technology, London, England, July 1986, pages 585-592, published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Bowe82] D.L. Bowen, Lawrence Byrd, Fernando C.N. Pereira, Luis M. Pereira and David H.D. Warren, 'DECsystem-10 Prolog user's manual', Occasional Paper 27, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, November 1982.

[Bruy82] Maurice Bruynooghe, 'Adding redundancy to obtain more reliable and more readable Prolog programs', *Proceedings of the First International Logic Programming Conference*, Faculte des Sciences de Luminy, Marseille, France, September 1982, pages 129-133.

[Burs80] R.M. Burstall, D.B. MacQueen and D.T. Sannella, 'HOPE: an experimental applicative language', *Conference Record of the 1980 LISP Conference*, Stanford, California, August 1980, pages 136-143.

[Card84] Luca Cardelli, 'Compiling a functional language', *Conference Record of the ACM Symposium on LISP and Functional Programming*, Austin, Texas, July 1984, pages 208-217.

[Clar81] Keith L. Clark and Steve Gregory, 'A relational language for parallel programming', *Proceedings of the 1981 Symposium on Functional Programming and Computer Architecture*, Portsmouth, New Hampshire, October 1981, pages 171-178.

[Clar83] K.L. Clark and S. Gregory, 'PARLOG: a Parallel Logic Programming Language', Research Report DOC 83/5, Imperial College of Science and Technology, May 1983.

[Clar86] Keith L. Clark and Steve Gregory, 'PARLOG: parallel programming in logic', *ACM Transactions on Programming Languages and Systems*, 8:1, January 1986, pages 1-49.

[DeGr84] D. DeGroot, 'Restricted And-parallelism', *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1984, pages 471-478.

[Debr86a] Saumya K. Debray and David S. Warren, 'Detection and optimisation of functional computations in Prolog', *Proceedings of the Third International Conference on Logic Programming*, Imperial College of Science and Technology, London, England, July 1986, pages 490-504, published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Debr86b] Saumya K. Debray and David S. Warren, ‘Automatic mode inference for Prolog programs’, *Proceedings of the 1986 Symposium on Logic Programming*, Salt Lake City, Utah, September 1986, pages 78-88.

[Demb85] P. Dembinski and J. Maluszynski, ‘AND-parallelism with intelligent backtracking for annotated logic programs’, *Proceedings of the 1985 Symposium on Logic Programming*, Boston, Massachusetts, July 1985, pages 29-38.

[Denn76] Peter J. Denning, ‘Sacrificing the calf of flexibility on the altar of reliability’, *Proceedings of the Second International Conference on Software Engineering*, San Francisco, California, October 1976, pages 384-386.

[Greg85] Steven Gregory, ‘Design, application and implementation of a parallel logic programming language’, Ph.D. thesis, Department of Computing, Imperial College of Science and Technology, University of London, September 1985, 263 pages.

[Hend82] P. Henderson, ‘Purely functional operating systems’, in *Functional programming and its applications*, J. Darlington, P. Henderson and D.A. Turner (eds.), Cambridge University Press, 1982, pages 177-192.

[Mell81] C.S. Mellish, ‘Automatic generation of mode declarations in Prolog programs’, *Proceedings of the Workshop on Logic Programming for Intelligent Systems*, Long Beach, California, September 1981.

[Miln78] Robin Milner, ‘A theory of type polymorphism in programming’, *Journal of Computer and System Sciences*, 17:3, December 1978, pages 348-375.

[Mycr84] Alan Mycroft and Richard A. O’Keefe, ‘A polymorphic type system for Prolog’, *Artificial Intelligence*, vol. 23, 1984, pages 295-307.

[Nais85a] Lee Naish, ‘Automating control of logic programs’, *Journal of Logic Programming*, 2:3, October 1985, pages 167-183.

[Nais85b] Lee Naish, ‘The MU-Prolog 3.2 reference manual’, Technical Report 85/11, Department of Computer Science, University of Melbourne, Melbourne, Australia, October 1985, 17 pages.

[Pere86] Luis Moniz Pereira, ‘Rational debugging in logic programming’, *Proceedings of the Third International Conference on Logic Programming*, Imperial College of Science and Technology, London, England, July 1986, pages 203-210, published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Port83] Graeme S. Port, ‘Mode directives as an AND control mechanism for logic programs’, Honours report, University of Melbourne, Melbourne, Australia, November 1983.

[Sara86] Vijay A. Saraswat, ‘Problems with Concurrent Prolog’, Technical Report CMU-CS-86-100, Department of Computer Science, Carnegie-Mellon University, January 1986, 35 pages.

[Shap83a] Ehud Y. Shapiro, ‘A subset of Concurrent Prolog and its interpreter’, ICOT Technical Report TR-003, Institute for New Generation Computer Technology, Tokyo, Japan, 1983, second version.

[Shap83b] Ehud Y. Shapiro and Akikazu Takeuchi, ‘Object oriented programming in Concurrent Prolog’, *New Generation Computing*, 1:1, 1983, pages 25-48.

[Smol84] Gert Smolka, ‘Making control and data flow in logic programs explicit’, *Conference Record of the ACM Symposium on LISP and Functional Programming*, Austin, Texas, July 1984, pages 311-322.

[Thom86] James Thom and Justin Zobel eds., ‘NU-Prolog reference manual, version 1.0’, Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.

[Ueda86] Kazunori Ueda, ‘Guarded Horn clauses’, D.Eng. Thesis, University of Tokyo, Tokyo, Japan, March 1986.

[Vasa86] T. Vasak, ‘A survey of control facilities in logic programming’, *Australian Computer Journal*, 18:3, August 1986, pages 136-145.