A parallel logic programming system based on strong and precise modes

Zoltan Somogyi

Technical Report 89/4
January 1989

Thesis submitted for the degree of
Doctor of Philosophy

Supervisor:
Dr. Kotagiri Ramamohanarao

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

Chapter 1

Introduction

> It is not from space that I must seek my dignity, but from the government of my thought. I shall have no more if I possess worlds. By space the Universe encompasses me and swallows me up like an atom; by thought I comprehend the world.
>
> Blaise Pascal

Logic programming is one of the most active areas of computer science today. The ideas of logic programming and of declarative programming in general have shown tremendous potential. Through their clean semantics, they promise to lift programmer productivity, to help in the production of provably correct programs, and to harness effectively the power of parallel computers. And yet, logic programming languages are still viewed as toys, suitable only for research into deductive databases and artificial intelligence and possibly for one-off commercial applications. They are still not viewed as suitable for serious large-scale programming projects, e.g. operating systems [Gall87], despite a few exploratory forays in this direction (e.g. [Fost87a, Kusa88, Chik88]). The research presented in this thesis is an attempt to correct this situation.

The main reason for the above view is not very hard to find: most logic programming languages were simply not *designed* to be used in production environments. They were created to explore a new style of computation, and their designers chose to keep their creations as simple as possible. This choice was a wise one in the circumstances: a minimal environment is essential if one wishes to separate the fundamental properties of the paradigm from the accidental properties of the system that implements it. However, these "accidental" properties are very important in languages intended for production use, and such languages must be judged by different criteria, criteria that reflect the needs of software engineers.

The main difference between the requirements of exploratory and production programming is the dissimilar levels of importance they attach to flexibility. In exploratory programming flexibility is all-important; in production programming it should be subordinated to the requirement of reliability. The following quote, from a paper appropriately titled "Sacrificing the calf of flexibility on the altar of reliability" [Denn76], illustrates the software engineer's point of view:

> The term "flexibility" is usually used to denote the existence of a range of choices available to a programmer or implementor - the more choice, the better the flexibility. Flexibility is sometimes referred to as "generality". Because it is frequently presented in the desirable terms, "delaying the binding of virtual objects to their realisations", increased flexibility has generally been considered a fatted calf of great succulence, sought voraciously and in the hope that the increased complexity that invariably attends it can be kept under control. In recent years, the wisdom of this quest has been called into question, for flexibility without discipline appears contrary to the plans of the god of reliability. (An automobile with independent steering mechanisms for both front wheels is more flexible but less disciplined than an ordinary one. It is mainly of metaphysical interest, offensive to the god of safe highways.)

This quote is a reaffirmation of one of the most important notions to have come out of software engineering research in the past two decades: the notion of “liberating constraint”. The idea is that if you give the system software control over the less important aspects of your task, you will have more time to devote to the crucial issues. The evolution of programming languages supports this view: each language generation so far has assigned to system software functions whose control used to be the programmer’s responsibility, and consequently removed possible sources of programmer error. With assemblers it was errors in the management of addresses; with high level languages (such as Fortran) it was errors in low-level code selection; with strongly typed high level languages (such as Pascal) it was errors involving inconsistent assumptions about variables; with very high level languages (such as SETL) it was pointer errors.

Logic programming languages are as yet too new to have undergone much evolution along these lines, but the beginnings are there. The evidence is the emergence of type systems for logic languages [Zobe87, Dart88, Nais88c, Yard87, Gogu84, Voda84, Voda88, Xu88, Redd88, Bruy82, Bruy88, Kana85, Mish84, Mycr84, Fruh88, Debr88a, Nais87, Nils83, Diet88]. However, with a few exceptions, these type systems are intended to be administered by special tools and are invisible to the compiler. This is somewhat analogous to C, where the compiler and the program checkout tool *lint* have different ideas about the strength of C’s type system. Based on the (large) number of times we have debugged other people’s programs just by running *lint* on them (something they neglected to do), we believe that for full effectiveness type correctness must be enforced by the compiler.

The focus of this thesis is the next step beyond type systems: mode systems. Just as type systems govern what values may be bound to a variable, mode systems govern what parts of the program may do the binding. Just as a type system is *strong* if the compiler knows the type of every data item [Fokk81], a mode system is strong if the compiler knows the instantiator of every data item. A strong mode system is desirable for the same reason a strong type system is: because it detects a class of programmer errors at compile time (errors involving incorrect management of logical variables). It thus saves the programmer time that would otherwise be needed to hunt down the bug with less direct methods, e.g. execution tracing.

The term “mode” was first given its present meaning by Edinburgh Prolog [Warr77]. In this original incarnation, modes were efficiency hacks. They asserted that all calls to a procedure had certain instantiation patterns in common; the language implementation exploited this information to omit the appropriate instructions from the procedure entry code. If the mode declarations were correct, they sped up program execution but did not affect it in any other way; if they were incorrect, the results were undefined.

The first language in which modes *governed* execution, and did not merely describe it, was the Relational Language [Clar81]. The mode system of the Relational Language was strong, enforced by the compiler’s requiring the programmer to indicate the producer of each variable. Further work by Clark and Gregory moved away from this. Parlog [Clar86] retains the mode declarations of the Relational Language, but allows them to be overridden in the bodies of procedures. The motivation was the desire to let processes cooperate in the construction of terms; this ability is a prerequisite for a number of programming techniques, the most important of which is the paradigm of incomplete messages [Shap83b]. These techniques are so useful that Clark and Gregory sacrificed the strength of the mode system for their sake.

The key insight of our research is that *a strong mode system can be flexible enough to allow programs adhering to these paradigms if it has precise information about the possible structures of terms, and that this information is exactly what is provided by a strong type system*. All the results we report in this thesis hinge on this observation. Chapter 3 will present detailed arguments supporting this insight; for now we note that even strongly moded logic programs are strictly more expressive than functional programs (i.e. there is a simple mapping from functional programs to equivalent strongly moded logic programs).

Strong modes have many uses beyond simple error prevention. The most important of these is that they serve as a basis for synchronisation in stream AND-parallel systems. (Historically, this was the main motivation for the development of the line of languages that most influenced our research: IC-Prolog [Clar79] with its coroutines, the Relational Language and Parlog.) In the deductive database area, modes generalise the notion of adornments, and as such can extend the applicability of optimisations such as the magic set transformation [Beer87]. For conventional (top-down) implementations of logic programs, modes help by making it easier to analyze operational behaviour (we will give an important example in section 4.4). Compilers can exploit this to perform more optimisations than would otherwise be feasible (e.g. better indexing), thus helping to narrow the gap between the efficiencies of von Neumann and logic programming implementations. The gap probably cannot be eliminated totally, but it probably need not be: the automatic exploitation of parallelism in logic languages promises to conceal any remaining inefficiencies.

Strong modes also have applications in the theory of logic programming. Well-moded programs have virtually all the good properties that allowed programs have, and some more besides. One, they accommodate evaluable predicates in a natural manner; two, they allow procedures to succeed without grounding all their arguments (see section 2.2). Further, modes can help reality match the theorems: the theorems we are concerned with require that unification include the occur check, but implementations invariably leave it out. For a very large class of programs, the information provided by modes shows that it is actually safe to do this. Note that modes themselves are neither logical nor non-logical; they serve to restrict the domain of discourse by disallowing non-well-moded programs. They could thus be called “extra-logical”. (By restricting the set of allowable computation rules they may also introduce incompleteness, but we do not expect this to be a problem in practice; we note that all practical implementations of logic programming languages have restrictions at least this strong.)

The semantics of logic programming languages are much simpler than the semantics of their traditional counterparts. This leads to programs that are easier to understand, easier to analyze, and easier to verify. However, much of the benefit is lost in the concurrent logic languages because their implementations stray too far from theory. Modes help us to stay close to the theory by making the combination of don’t-know nondeterminism with stream AND-parallelism efficient enough for practical applications; in chapter 5 we prove this by presenting a backtracking algorithm for stream AND-parallel logic programs. Without compile-time knowledge of modes, such an algorithm would not be viable: the forward execution algorithm would have to spend too much time collecting dataflow information.

In stream AND-parallel systems, side-effects manifest themselves as instantiations of variables that cause other things to happen as well. One can think of this as having a logic program compute a result and the real world reacting accordingly (e.g. by printing the result). This is a cleaner model than Prolog’s side-effect-causing procedures, which cannot be thought of in that way. Still, they cause some difficulties, the chief of which is that they cannot be backtracked over. One must therefore ensure that no need for this will ever arise. Side-effects are the life-blood of systems programs, so they cannot be prohibited; instead it is backtracking whose scope must be restricted. The three main logic languages used for systems programming, Parlog, Concurrent Prolog and GHC, prohibit it altogether by requiring the employment of a non-logical feature (the guard operator). Armed with the information provided by modes, we can afford to be more selective, disallowing backtracking (or OR-parallelism in its place) only in those parts of the program where this is necessary. We present a scheme for doing this in chapter 4.

By accepting the discipline required by a strong mode system, we gain all the benefits listed above. These benefits are worthwhile by themselves, but they are even more valuable together, because given all of them we can build a parallel computer system programmed almost entirely in a clean logic programming language (device drivers and other low-level mechanisms may still be written in a conventional language or in microcode). We give this language the name *Ptah*, after the chief god of Memphis, the ancient capital of Egypt.

1.1. Outline of the thesis

The rest of this chapter presents the background material necessary for full understanding of this thesis.

Chapter 2 presents our mode system; it is the first strong mode system that can handle partially instantiated data structures. It analyzes binding information in terms of the smallest possible unit and relies on a type system to keep binding patterns regular.

Chapter 3 examines the implications of our type and mode systems, concentrating on software engineering issues. It also compares our mode system to other mechanisms used previously for the same purposes.

Chapter 4 investigates the application of logic programming to programming environments and operating systems. Its main result is a solution of the conflict between logic programs' tendency to have multiple solutions and real world devices' dislike of being told to do more than one thing in any one situation.

Chapter 5 presents a backtracking algorithm that can handle stream AND-parallel logic programs. It relies on the mode system to supply compile-time knowledge of the dataflow graph of each clause; this allows it to determine efficiently which goals to kill or restart when a goal fails. We also show that modes can increase the precision of the backtracking algorithm and allow our algorithm to handle efficiently programs that manipulate partially instantiated data structures and an important class of programs with circular dependency graphs (including those based on the client-server model).

Chapter 6 investigates implementation issues, concentrating on the implications of our backtracking algorithm. For example, it shows how we can trade off the precision of the backtracking algorithm against its overhead on a procedure-by-procedure and call-by-call basis. On code that does not need backtracking, the efficiency of our algorithm approaches that of the committed-choice languages; on code that does need backtracking its overhead is comparable to that of the independent AND-parallel backtracking algorithms.

1.2. Background

Throughout this thesis we use a mixture of standard terminology and some new terms. This section reviews our use of standard terms and introduces some new ones; other new terms will be introduced in the main body of thesis, as we encounter the need for them. These new terms can also be found in the glossary at the end of the paper.

A *term*, which we will sometimes call a *structure*, is either a variable V or a function symbol (alias functor) f of arity n applied to n other terms $f(term\_1, \ldots, term\_n)$. A *constant* is a functor of zero arity; we omit the empty pair of parentheses. An *atom* or *procedure call* or *goal* is a predicate symbol p of arity n applied to n terms $p(term\_1, \ldots, term\_n)$. A *literal* is an atom or the negation of an atom. A *clause* is a formula of the form $A :- B0, \ldots, Bn$, for $n \geq 0$, where A is the *head* atom and the Bi are the *body* literals. A *procedure* or *predicate definition* is a set of clauses whose heads have the same predicate symbol. A *normal program* is a set of procedures defined by such clauses [Lloy87].

We follow the Prolog convention of starting the names of variables with upper case letters and starting the names of functors and predicates with lower case letters. However, we break with tradition in adopting as the basis of Ptah the *extended syntax* of [Lloy84]. In a language based on extended syntax, the body of a procedure can be any well-formed first-order formula; it does not have to be a formula in disjunctive normal form, which is what a set of clauses amounts to. This freedom allows programmers to express their intentions without resorting to the use of auxiliary procedures when the logic of the problem calls for (say) the negation of a disjunction in a procedure definition. However, the first-order formulae of mathematics are not enough for some of the applications we have in mind (e.g. operating systems), so we consider the bodies of procedures to be first-order formulae extended with operations such as sequential AND and guards; we call these formulae *expressions*.

An expression is built up by connectives from primitive expressions (procedure calls, including calls to built-in procedures). We denote the AND connective by a comma , or by an ampersand &. This operator does not imply sequencing; the implementation is free to evaluate the two sides concurrently, or it may evaluate the right side before the left. For occasions when the programmer wishes to enforce a fixed evaluation order, we provide a sequential AND operator, which we denote by & > (pronounced "and then"). We will spell out the OR connective, denoting it by *or*. We denote negation by a tilde ˜ and implication by an arrow ->. Our guard operator is **/**, and we separate alternative guarded expressions by *alt* (this operator is distinct from normal disjunction). The symbols ∃ and ∀ denote the standard forms of quantification: they are followed by a list of variables and an expression, with the obvious meaning. The order of precedence of the operators is: negation, parallel conjunction, sequential conjunction, the guard operator, the alternative operator, disjunction, and quantification. Curly braces *{}* denote grouping; as an expression in braces is an indivisible unit, programmers can use braces to override the precedence rules. Comments are introduced by percent signs % and extend to the end of the line.

Our standard set of variables is X, Y and Z, sometimes supplemented by U, V and A, B, C etc. Our standard procedure names are p, q, r, s and t, while our standard functor names are f and g. We will denote terms by t and expressions by e, although in some cases we will also denote expressions by c for "condition", g for "guard" or b for "body". Any of the above letters may be followed by a number (or sometimes two numbers). We will of course use more descriptive names when our subject matter permits us to do so. We will sometimes write the functor *cons* as an infix dot. We may represent other functors similarly; when we do, we will state this in advance.

The mathematical connectives have their traditional semantics. The semantics of the & > operator is the same as that of & except that the evaluation of the right subexpression will not start until the left subexpression has been completely evaluated. This semantics can be exploited to override our usual anti-substitutive unification [Ueda86b] and implement atomic unification; e.g. to bind X to $f(g(Y))$ atomically, one executes $Z = g(Y)$ & > $X = f(Z)$. Guards have the same semantics as in the concurrent logic languages. This semantics is twofold: all but one solution of the left subexpression will be discarded, and if several guarded expressions are joined by the *alt* connective then all but one of these guarded expressions will be discarded. The *alt* operator itself can only appear when each of its subexpressions is a guarded expression, i.e. its main connective is the guard operator, or the *alt* operator itself; this latter possibility is needed to allow guard systems with three or more alternatives. The scope of guard operators is limited to the left subexpressions of the alternatives in their guard systems; when running the code *e* & *{g1* **/** *b1 alt g2* **/** *b2}*, commitment cannot affect the execution of *e* because it is outside the guard system. Note that an expression such as $g1$ **/** $b1$ *or* $g2$ **/** $b2$ is legal: it denotes two guarded commands that do not compete against each other. That expression may have solutions from each arm of the disjunction, whereas the expression $g1$ **/** $b1$ *alt* $g2$ **/** $b2$ may have solutions from at most one arm. We do not require all expressions to have a single solution, although we do insist on this when necessary (see section 4.4).

We require the heads of procedures to consist of distinct variables. Unification must be explicitly invoked in the body, as in the *homogeneous form* of normal programs. As each of our procedures is defined by a single expression, neither structures nor repeated variables make much sense in their heads. The main advantage of our approach is that it allows and requires programmers to name every argument of every procedure. These names can be very valuable as documentation.

Although our programs are expressed in extended syntax, we sometimes restrict our attention to normal programs (e.g. our backtracking algorithm only applies to normal programs). At such times, we do not intend that our discussion should apply only to procedures whose definition bodies just happen to be in disjunctive normal form. Instead, we intend that all procedure bodies should be transformed to disjunctive normal form if they are not already in that form. This transformation could be effected through the standard Lloyd-Topor algorithm [Lloy84], but we prefer to modify it in two respects. One is that we prefer to translate *e*1 *->* *e*2 not as *~e1 or e*2 but as *~{e1 & ~e2}*, as the second alternative allows *e*1 to generate values for *e*2. (Note that the expression *~{e1 & ~e2}* is not itself in disjunctive normal form and would therefore have to be transformed further.) The other is that where the Lloyd-Topor algorithm replaces a "clause" of the original definition with two new clauses derived from the old one, we prefer to introduce a call to a new predicate that itself has two clauses. This yields faster and possibly smaller code as the other calls in the original clause do not have to be replicated, and it allows us to handle guards without any special precautions. Note that a Lloyd-Topor translation of *{e1 or e2}* **|** *b* to *e*1 **|** *b or e*2 **|** *b* would be incorrect because the latter code will duplicate every solution of *b* when both *e*1 and *e*2 are true. Replacing the *or* with *alt* yields correct code in the above case, but it does not work when the disjunction is in the body: *g* **|** *{e1 or e2}* is not equivalent to the expression *g* **|** *e*1 *alt g* **|** *e*2 (in this case *or* is the proper connective).

We call a variable *void* if it occurs only once in its scope. Unifications with void variables always succeed without effect. We say that a term is *ground* if it contains no variables. We say that a variable is *instantiated* if it is bound to a nonvariable term; this term need not be ground.

We require all procedures to declare the types of their arguments and to declare their allowed modes of use. We will give the syntax for these in section 2.1 and 2.2 respectively. Both type and mode declarations are checked by the compiler, and we consider only programs that are type- and mode-correct. Without loss of generality, we assume that queries are single atoms, calls to a distinguished procedure such as *main*, and that they are type- and mode-checked as well.

We require our readers to be familiar with the notions of stream AND-parallelism [Greg85] and don't-know/don't-care nondeterminism [Kowa80; p. 113]. These concepts are best explained in the context of the *AND/OR process model* [Cone83]. This model views (normal) logic programs as being executed by a tree of alternating levels of AND and OR processes. *AND processes* are responsible for solving the bodies of clauses; *OR processes* are responsible for solving individual calls. The root of the tree is an AND process corresponding to the query. The names of the processes are derived from their characteristics: the solution of a clause requires the solution of every one of its calls, while the solution of a call requires the solution of just one of its clauses.

An AND process solves a clause by coordinating OR processes activated for the calls of the clause. *AND-sequential* systems allow only one OR-process to be active in any given clause at any given time. *Independent AND-parallel* systems allow multiple OR-processes to be active if they share no unbound variables. *Stream AND-parallel* systems do not even have this lesser restriction; they allow all OR-processes to be active concurrently. These processes communicate via shared variables. They can synchronise by suspending themselves on a variable until that variable is instantiated by another process. The mechanism of this suspension is language-dependent, but it usually involves an attempt to bind a variable that the OR process is not allowed to instantiate. In GHC these are the variables that occur in the parent goal when the child is executing its guard; in Parlog it is the variables that have been declared input if this declaration is not explicitly overridden in the code; in Concurrent Prolog it is variables with read-only annotations, annotations that a variable can acquire at runtime.

Independent and stream AND-parallelism are both *forwarding* schemes: if two OR processes in a conjunction share a variable, then one will instantiate the variable and the other will consume its value. This does not happen in systems based on *join AND-parallelism*. In those systems, the two OR processes will each compute sets of values for the shared variable and the AND process will take the intersection of the two sets. Because of the possibility of compatible but non-identical solutions, this intersection process requires the use of unification [Poll81]. For example, if $p(X)$ has the only solution $X = f(a, Y)$ and $q(X)$ has the only solution $X = f(Z, b)$, then the only solution of $p(X) \,\&\, q(X)$ is $X = f(a, b)$. This overhead is not present in forwarding systems. Forwarding systems also have the advantage that they naturally restrict the search space of the consumers of a variable to the set of values that the producer provides for the variable. In standard joining systems, there are no producers or consumers, and no such restriction of the search space takes place; the result is the waste of computing power on useless searches. This can be prevented by moving the execution algorithm of the joining system closer to that of a forwarding system, e.g. as demonstrated by [Kasi85], but only at the cost of more overhead.

The classification of AND-parallel schemes into the forwarding and joining classes is from [DeGr84]. Because of the weaknesses of the joining approach that we have outlined above, most research on AND-parallelism these days concentrates on the forwarding approach. This research has two main thrusts: independent AND-parallelism for the transparent parallelisation of old sequential languages, i.e. Prolog (the “dusty deck” approach) and stream AND-parallelism for the execution of new, concurrent logic programming languages.

The main problem for independent AND-parallel systems is how to verify that two goals are in fact independent. The obvious approach, which computes the sets of variables that occur in each goal and checks for variables that are in both sets, is extremely inefficient when goals are large, as they often are. Because of this, most independent AND-parallel systems [Herm86b, Herm87a, Lin87, Lin88c, Lin88b, West87, Baro88] compile their programs into a form that requires only limited checks: to see whether two variables are the same or not and to see whether a term is ground. This compilation must be based on a worst-case analysis of the program according to *restricted AND-parallelism* or RAP model that was introduced in [DeGr84]. Because of the essentially approximate nature of the analysis, systems based on the RAP model may miss some potential parallelism, as they sometimes consider two goals to share variables when in fact they don’t. However, this is definitely preferable to investigating large goals in the hope that they are independent and then finding that they are not.

The other problem that plagues most independent AND-parallel systems is an artifact of their dusty-deck approach: the inherently sequential semantics of Prolog’s side-effects. Although the RAP model can be modified to handle these [DeGr87], the cost is a significant increase in complexity. As a consequence of this complexity, it is not clear whether the semi-intelligent backtracking algorithms that have been devised for the original RAP model, e.g. [Herm86c], also apply to the modified model. Two other problems are concerned with efficiency: the modification requires an increase in overhead, and the extra synchronisation required for the proper handling of side-effects reduces the amount of parallelism available. For these reasons, some systems, e.g. PEPSys [Baro88] and APEX [Lin88c], apply AND-parallelism only to side-effect free code.

Unfortunately, at the moment none of the independent AND-parallel implementations (that we know of) can outperform a good sequential Prolog implementation (e.g. Quintus Prolog), except in a few cases when using many processors: about 8 processors in [Baro88] and 16 in [Lin88c]. This situation should improve when the parallel implementations move to machines with faster individual processors, and if/when their implementors recode them in assembler. However, at the moment we have no numbers for independent AND-parallel systems that can handle Prolog side-effects; we can expect these to be substantially slower than their side-effect-free counterparts.

Stream AND-parallel systems take a somewhat different approach to the issue of side-effects: they usually consume their inputs and generate their outputs in the form of *streams* [Card83]. The idea is that the input is a list or stream of characters that is conceptually generated by a *perpetual process* that represents the keyboard, and the output is a stream of characters that is consumed by a perpetual process that represents the screen. (A perpetual process is represented by a call to an infinitely self-recursive procedure [Warr82].) In these systems, side-effects are generated by binding special variables, not by executing special goals. To write a character on the screen, programs execute $Out = 'a'.Out1$ or something similar; the screen process monitors the value of *Out*, and will cause its elements to appear on the terminal as they become instantiated. Similarly, to read a character from the keyboard, programs execute a unification $In = C.In1$, making sure in language-dependent ways that they do not instantiate *In* before the keyboard process does.

The main virtue of this arrangement is that it allows us to give a logical semantics to code that does input/output (programs using the Prolog built-ins *read* and *write* and their cousins can have no such semantics). However, in the pragmatic sphere this advantage is overshadowed by another one: stream-based I/O replaces Prolog's left-to-right sequencing of side-effects with an order based on data dependency. To print two characters, programs execute $Out = 'a'.Out1$ and $Out1 = 'b'.Out2$, but the order of executing these two goals does not matter. This approach can be generalised to handle all kinds of *soft* side-effects (side-effects that do not change the program itself; the term is from [DeGr87]). This leads us to the basis of operating system research based on stream AND-parallel logic languages: the insight that the environment of a logic operating system can be characterised by the "query"

```
?- device_1(S1), ..., device_n(Sn), operating_system(S1, ..., Sn).
```

where the *device_i* procedures are device drivers implemented outside the logic programming system itself and the *Si* are streams of commands and data [Clar81, Shap83a, Kusa86].

In Prolog, *hard* side-effects are represented by the built-in procedures *assert*/*retract* and their variants; some of the concurrent logic languages had similar primitives, e.g. *addcl* and *delcl* in early Parlog [Clar83]. Apart from wreaking havoc with the declarative semantics of programs, as pointed out in [Warr84], and being tied to a sequential execution order, they defeat virtually all attempts at optimisations [Debr88b]. They even make life difficult for simple compiled systems: they require the implementation to contain either an interpreter or an incremental compiler, and in the absence of a decompiler they require the implementation to keep two copies of the user's program, one each in source and object forms.

In stream AND-parallel systems, one can use perpetual processes instead of non-logical primitives to represent databases. A perpetual process employed to do this carries state information in some of its arguments; this information can be updated simply by giving different arguments to the recursive call. The rest of the program is a *client* of the perpetual process, which can be viewed as a *server* in standard operating system terminology. The client communicates with the server by sending it messages on one of its arguments, according to the paradigm of *incomplete messages* [Shap83b] by instantiating this argument *S* to a value such as $Msg.S1$. The message *Msg* could be e.g. *assert*(*some_clause*) to update the database or *clause*(*Clause*) to retrieve some information from it. In the first case, all the information is supplied by the client. In the second, the client leaves *Clause* unbound, allowing the server to instantiate it (hence the name of the paradigm). In both cases, both the client and the server recurse on $S1$ instead of S.

As for the use of *assert/retract* to change the code of a running program, we believe that running programs should never be modified. Even the implementation that has "defensible semantics" in Prolog [Lind87] is indefensible in a stream AND-parallel environment, because it relies for its defensibility on a sequential evaluation strategy. However, we do allow the *addition* of new procedures to running programs, to cater for applications that need to construct code dynamically. In our system, a program can construct a new procedure either in text form or as a term representing a syntax tree. To use the new procedure, the program must know the types of its arguments and the modes in which it may be used, and declare a higher-order variable of that type and mode accordingly. The program can then give the source to the compiler; if the compiler finds no errors, it will instantiate the procedure-valued variable. This variable may appear whenever a procedure name may; calls to procedures represented by variables are of course delayed until the procedure-valued variable gets its value. We will have more on this in section 4.1. For now the important point is that our system does not have *assert/retract*; indeed it has *no side-effects at all in the Prolog sense*.

One of the Prolog side-effects that we do without is the cut. This would make the implementation of OR-parallelism much easier [Haus88, Butl88], but in this thesis we are not much interested in OR-parallelism. Therefore, for the sake of simplicity of presentation we assume that the engine that evaluates our logic programs is OR-sequential, i.e. it tries one clause at a time in the solution of any call. However, this is not essential. The implementation could be OR-parallel, exploiting *backup OR-parallelism*: if a processor is idle and a goal has some unexplored clauses left, then we could start computing on the idle processor the solution to that goal using one of the unexplored clauses, taking care to rename (i.e. get fresh value-cells for) the output variables of the goal first (our mode system will tell us what (parts of) arguments these are). If and when the goal backtracks into the clause we have chosen, we can make the backup computation the main computation by simply installing pointers from the original to the backed-up value-cells of the output variables. It should be possible to implement this scheme more efficiently than its closest relative for Prolog, the *favoured bindings* approach of [Disz87], because in our scheme more information is available at compile time.

The other issue affecting OR processes is the distinction between don't-know nondeterminism and don't-care nondeterminism. The two differ in the method of selecting the clause to be used in the derivation of a solution. A *don't-know nondeterministic* system using backtracking tries each clause in turn until it finds a solution; if the solutions computed by the current clause are all rejected (by other calls that are siblings of its ancestors), it tries the next clause until it runs out of clauses. This is sometimes called *backward execution*, in contrast to the usual *forward execution*. A system that implements don't-know nondeterminism via OR-parallelism would try all the clauses at the same time, and return all solutions computed by all the clauses. A *don't-care nondeterministic* system, on the other hand, will choose just one clause, and use only this clause to find solutions. To help choose a clause that will lead to a solution, the system usually executes a part of each clause (the part designated by the programmer as the guard), and selects only from among the clauses whose guards can be solved. However, this still does not guarantee that the clause as a whole has a solution, and of course it discards any solutions that may be computed by the other clauses. Note that don't-care nondeterministic procedures may have multiple solutions if the system allows don't-know nondeterministic procedures as well; the don't-care nondeterministic procedure can call a don't-know nondeterministic procedure in its body. If, however, all clauses are required to have a guard, and all built-in predicates are deterministic (the usual case), then no don't-know nondeterministic procedures can exist and all computations will have at most one solution.

As should be obvious from our discussions above, we do not impose this restriction; our system allows don't-know nondeterminism. The main concurrent logic languages, Concurrent Prolog, GHC, and Parlog, do not, for two reasons. One is that bindings made to a side-effect variables should never be retracted, as could happen in a backtracking system, nor should such variables be given alternative values, as could happen in an OR-parallel system. We discuss our solution of this problem in chapter 4. The other reason is that the implementation of backtracking in a stream AND-parallel system is difficult in the absence of mode information; we present our algorithm to do this in chapter 5. All the recent proposals for systems with both don't-know nondeterminism and stream AND-parallelism [Nais88a, Yang86, Hari88, Bahg88b], are very careful to separate the stream AND-parallel part of the execution algorithm from the don't-know nondeterministic part (which uses OR-parallelism in three of those proposals and backtracking in the fourth).

Chapter 2

Language issues

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 [Nais85c], 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. The main difference is that our mode system pays attention to the structure of terms. We consider the fundamental binding units to be unifications of the form $X = f(Y1, \ldots, Yn)$ where the *Yi* 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.

2.1. Types

Our type system is not a new one. It is virtually identical to those of the functional languages HOPE [Burs80] and ML [Card84], and to the Prolog type system described in [Mycr84]. Its style of parametric polymorphism is based on [Miln78].

Definition A *type T* is either a *type variable* (see below) or a *rational AND/OR tree*, written as an expression of the form

```
T = f1(T11, ... T1(m1)) or ... or fn(Tn1, ..., Tn(mn))
```

The *fi* are distinct function symbols; their arity is given by *mi* (an *fi* is a constant if $mi = 0$, in which case we omit the empty pair of parentheses). The *Tij* are themselves types that may be syntactically equal to T. Intuitively, an AND node like $fi(Ti1, \ldots, Ti(mi))$ 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. In C terminology, our types are unions of structures; in Pascal terminology, they are variant records, with the proviso that the non-variant part of the record is empty. A type for which all of the *mi* are zero (all the functors are constants) is analogous to an enumerated type.

We will assume that the implementation provides built-in types *integer* and *character*. One could define the type *character* as an OR node with 2^8 constant symbols as its children, but *character* and *integer* have to be built-in types anyway since we want to be able to write character and integer constants in the traditional manner (e.g. *'a'* and 100). In any case, declaring the type *integer* by enumerating its possible values is a futile exercise in practice, if not in theory.

To make programs a bit shorter, and (more importantly) to encourage the use of abstraction, we allow the programmer to give names to types. In our system, a *type definition* looks like this:

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

This is the type of lists whose elements are of type T. *listof* is the new *type constructor*; T is a type variable that is a parameter of the definition; what follows the = sign is the body of the type definition. The definition states that for any type T, one can obtain the type "list of elements of type T" by consistently substituting the description of type T in the body of this type definition. The type variable T acts as a *parameter* of the definition; hence this form of polymorphism is called *parametric polymorphism* [Miln78]. In general, a type constructor may have any number of parameters, including zero.

Figure 2.1 shows the tree corresponding to this definition. The figure also has a label on each OR-node. The mode system requires these labels; without them, it could not look inside the structures of terms. The full version of the type definition, which supplies these labels, is

```
type listof[T] = listnode: nil or cons(element: T, listof[T]);
```

Later we will refer to OR-nodes as simply *nodes*; there should be no risk of confusion, since we will rarely concern ourselves with AND-nodes.

Our mode system requires the language to be strongly typed. Consequently, we require every procedure to declare the type of every formal parameter. We allow programmers to declare the types of local variables, but this is not strictly necessary: the compiler can infer the types of local variables from the declared types of the arguments of the called procedures. Moreover, it can do so deterministically and in close to linear time, as proven in implementations of ML [Mitc88].

As an example of our syntax, we present the header of the procedure *length*:

```
procedure length(List: listof[T]; N: integer)
```

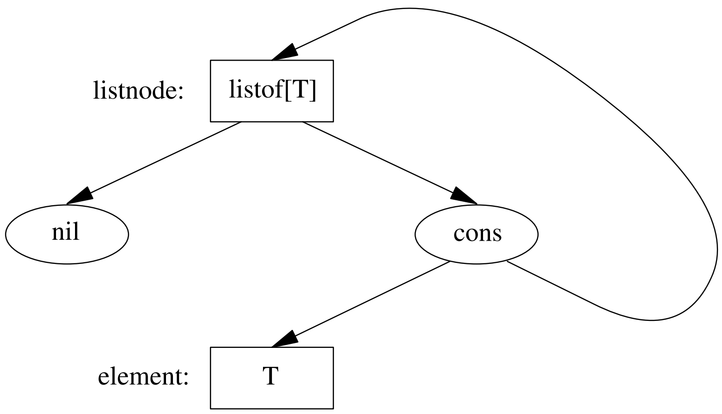

Figure 2.1: type tree of listof[T].

Definition 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 an expression they form an *extended type forest* for that expression.

Node names consist of two parts: the name of the parameter or local variable (to identify the tree) and the name of a node in that tree. The full name of a node is the list of the labels of the nodes required to reach it from the root, but the path may be omitted if the tree has only one node with a given label. For root nodes, we allow the name of the node to be omitted entirely.

Example Suppose we have defined a type *iocmd* as

```
type iocmd[T] = read(readdata: T) or write(writedata: T);
```

and suppose we have a variable S whose type is $listof[iocmd[T]]$. Figure 2.2 shows the type tree of S. The full name of the node labelled S is $S.listnode$; the full name of the node labelled $S.element$ is $S.listnode.element$; the full name of the node labelled $S.readdata$ is $S.listnode.element.readdata$; and the full name of the node labelled $S.writedata$ is $S.listnode.element.writedata$.

We require all our programs to be well-typed. By this we mean that the type of an actual parameter must match exactly the type of the corresponding formal parameter by the rules of the equivalence of rational trees, which decree that the two rational trees are equivalent if they represent the same set (usually infinite) of ordinary trees. Such a match is not allowed to instantiate type variables in the actual parameters. If the match instantiates type variables in the formal parameters, the instantiation values must be consistent both within and among parameters. A formal definition of type-correctness for systems like this appears in [Mycr84]. The literature contains many papers on how to perform the checks required by these systems, e.g. Mycroft & O'Keefe's paper itself presents one such algorithm. Alternatively, one can adapt a unification algorithm for rational terms, such as the one presented in [Jaff84].

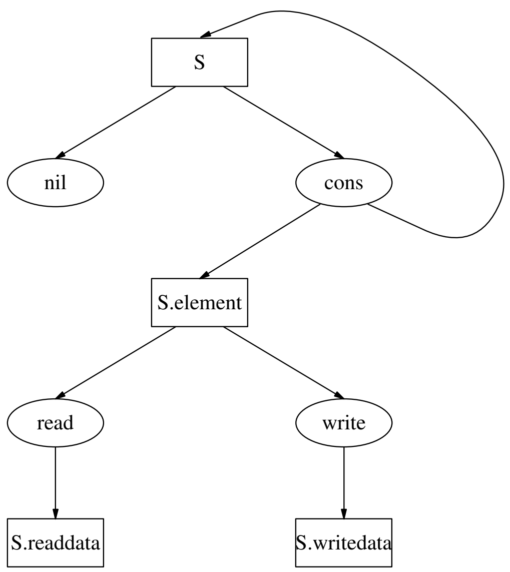

Figure 2.2: the type tree of S.

2.2. 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 produce its value. However, according to the definition in [Greg85], a Parlog program in which different parts 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 is more widely applicable. We attach producers not to variables but to the OR-nodes of the type trees of those variables. 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.

Definition In our system a *mode* is an assignment of a tag from the set {INBOUND, INFREE, OUTBOUND, OUTFREE} to each of the OR-nodes of the type forest of a procedure.

During program execution, variables become bound to terms containing other variables, e.g. *List* will probably be instantiated to $cons(Head, List1)$. Each of these runtime variables corresponds to an OR-node in the type tree of the original variable, e.g. *Head* corresponds to $List.element$ and $List1$ to the same node as *List*. In a procedure p, the tag of the node that a variable X corresponds to 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 $X = f(Y1, \ldots, Yn)$ for some functor f and variables Yi, 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 (which has no effect except conceptually "instantiating" the void variable). A node with an INFREE tag must not have a parent OR-node with an OUTBOUND or OUTFREE tag; either the parent's tag must be INBOUND or INFREE or the node must be the root node.

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 section 2.3).

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. A node with an OUTFREE tag must have a parent with an OUTBOUND or OUTFREE 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; to get the same effect as Parlog's strong modes, one has to attach the INBOUND or OUTBOUND tag to every node in the type tree of a variable. The advantage of our scheme is that it attaches mode information to the smallest possible unit; unifications of the form $X = f(Y1, \ldots, Yn)$ cannot be decomposed. Our scheme is therefore maximally flexible among strong mode systems based on our type system. For example, we can describe communication between two calls by dividing the nodes of the type trees of the shared variables into two classes, and ensuring that the nodes in one class have

INBOUND tags in the first procedure and OUTBOUND tags in the second procedure; the nodes in the other class have OUTBOUND tags in the first procedure and INBOUND tags in the second procedure (see example 2.5).

The INFREE and OUTFREE tags are new. OUTFREE represents an unbound variable created in and exported by the procedure; INFREE represents a variable whose value is not needed by the procedure, so the variable may be unbound as far as the procedure is concerned. (We will give an example of both new tags shortly.) Given these semantics, the tags of nodes whose parent has a FREE tag don't have any meaning, so we won't include them in mode declarations.

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. 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. (This notation and terminology is derived from [Port83].)

Two conventions make mode declarations shorter. One we have covered already: we allow abbreviations of node names. 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 (if they don't agree, it represents an error in the program). Together, these two conventions frequently allow us to write mode declarations involving only variable names.

Example The following program is an adaptation for our system of the standard Prolog code for *length*.

```
procedure length (List: listof[T]; N: integer)
mode
        List ? List.element => N;
        N => List ! List.element;
def
        List = nil,  N = 0
or
        List = Head.List1, length(List1, N1), plus(N1, 1, N)
end length
```

Example 2.1: procedure 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 a programmer's intuition about the reasonable uses of the *length* predicate. (A mode in which *length* checks that a given list has a given length is also reasonable, but it does not have to be declared; see section 2.4.1).

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 inside *List* corresponding to the FREE nodes with the void variable *Head*. In the first mode, the purpose of the call $List = cons(Head, List1)$ is to generate a value for $List1$. 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 against *Head creates* the anonymous OUTFREE variable inside the value of *List*.

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.

2.3. Mode assignments

Definition 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* if in each conjunction in the disjunctive normal form of the body of p

(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.

Definition A procedure is *well moded* (or *mode correct*) if in all its modes it has at least one valid mode assignment. A program is well moded if all its procedures are well moded.

The intention is to ensure that every node has exactly one producer (except for nodes that are never consumed which may have no producer). This prevents conflicts that would otherwise arise when two calls try to instantiate the same variable at the same time; this can never happen in well-moded programs.

Note that our definition treats local variables just as it treats output arguments; the differences are that the nodes of local variables don't appear in mode declarations, their values are not exported to the caller, and the compiler, not the user, figures out whether their nodes should have OUTBOUND or OUTFREE tags. Note also that if a procedure has more than one valid mode assignment, the choice among them affects only the efficiency, but not the semantics, of the procedure. In the ordinary course of events, the compiler decides which mode assignment it uses to compile the program, though the programmer may influence this decision through explicit directives (see section 2.4.3).

The bases for our recursive definition are the built-in procedures, the most important of which is unification. The special "procedure" $U = V$ has three normal modes: $U \Rightarrow V$ and $V \Rightarrow U$ (one input, one output; the assignment modes) and U, V (both input; the test mode). These apply when neither U nor V is a void variable. If say V is in fact void, the mode can be either $?\,U \Rightarrow\, !\,V$ or $\Rightarrow\, !\,U, V$, i.e. the other argument may be INFREE or OUTFREE. Unification is special in that each of these modes applies only to a single pair of nodes (one from each argument). The recursive cases can have other modes, as shown in the next example.

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 $List\ ?\ List.element \Rightarrow N$ and the call $plus(N1, 1, N)$ on the procedure $plus(A, B, C)$ has the mode $A, B \Rightarrow C$. The first level of $List = cons(Head, List1)$ is a test, the second level has a void unification and an assignment to $List1$ (this is explained further in section 2.4.4).

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).

2.3.1. The basic mode analysis algorithm

The programmer's job is to specify each procedure's modes. The compiler's job is 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. The algorithm the compiler uses to do this is descended from the one in [Port83] and bears some resemblance to one used in the old version of Parlog [Clar83]. Here we give only an outline; the loose ends will be cleaned up and a comprehensive algorithm presented in section 2.5. Both versions of the algorithm assume that the type trees of the actual and the formal parameters are identical node-for-node, not just in the set of ordinary (finite) trees they represent; we will cover this in section 2.4.4.

This version of the algorithm works for programs with definition bodies in disjunctive normal form. It examines each clause of each mode of each procedure independently. It bases its analysis on a *known nodes list*. A node of the extended type forest of the clause 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 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 (part of a) variable on the known node list; for a negated call this 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 that does *not* appear on the known nodes list;

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

(3) it can't find a feasible call while some calls are still unpicked (but see section 2.4.5).

Example We trace the algorithm in its analysis of the *append*3 predicate:

```
procedure append3(A, B, C, D: listof[T])
mode
        A, B, C => D;
        D => A, B, C;
def
        append(A, B, E) & append(E, C, D)
end append3
```

Example 2.2: procedure append.

where *append*(X, Y, Z) has the usual modes $X, Y \Rightarrow Z$ and $Z \Rightarrow X, Y$. For the first mode of *append*3, 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 *append*3 will use the first mode of *append* for both its calls. Similarly, for the second mode of *append*3 both calls will use the second mode of *append*. Note that unlike Prolog, our system does not require that the order of the two calls to *append* be reversed for *append*3 to be usable in this mode.

When the algorithm finds that a node has no producers, the mode analysis algorithm in fact has an ace up its sleeve: the type of a variable can itself act as a producer. The problem is that types usually have an infinite number of values. If the compiler uses the type of X which is (say) an integer to generate X in the expression $\forall X$: *~p(X)*, then the resulting program would most likely run forever. For this reason, we prefer requiring 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. The compiler can itself write the procedure *typegen* from the definition of the type of X.

Each mode of a procedure will in general assign different modes to the calls in the body of the procedure; this applies whether the procedures called are built-in or not. To distinguish these, the compiler emits separate code for each mode of the procedure. As the calls in the object code are also qualified by a mode of the called procedure, the runtime system has no notion of multi-mode procedures; the compiler takes care of everything in this respect.

In the rest of the thesis, we will often say *procedure* when we mean *procedure in a given mode*. We will rely on the context to specify what mode we are talking about; this context will usually be the body of some other procedure. In this case, the mode we mean is the mode assigned to the call of the first procedure in the body of the second procedure.

2.4. Variations on a theme

2.4.1. 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 (i.e. the mode of that call is N, *List* ? *List*. *element*; not N, *List* => ! *List*. *element*). Modes created this way are *implied modes*. Modes not implied by other modes are *principal modes*.

Example 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.

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*(*List*, 5) to go ahead when the elements of *List* are not instantiated, *whether or not they are going to be*.

One can also shift nodes from INFREE to INBOUND. This usually 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. The exception occurs when a call to the procedure has no producer for the node; this is OK for INFREE nodes but not for INBOUND nodes. On the other hand, it is never 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.

We can compile a procedure in one of its implied modes in one of two ways. One way is to simulate what would happen if this mode were a principal mode: do a mode analysis for the implied mode and compile the body according to this mode analysis. The other way is to substitute for the call to the procedure in its implied mode a conjunction consisting of a call to the procedure in one of the implying principal modes and a test unification (e.g. we can implement $length(List, N)$ (both inputs) by $length(List, N1)$ & $N1 = N$, where *length* produces $N1$). Both alternatives can find failure early; for the second alternative this requires that the test unification be at higher priority, so that it will be woken up whenever the other call produces any data (the usual generate-and-test paradigm [Nais85b]). The first alternative can yield faster code because it has fewer overheads in doing this. As for space, the first is preferable if the program text contains lots of calls to the implied modes, while the second yields shorter programs if there are only a few such calls.

Both implementation methods ensure that if a procedure would return n answers for a particular set of input data in a principal mode then it will return at most n answers for that set of data in any of the modes implied by that principal mode. Thus in our system all procedures are *steadfast* according to Richard O'Keefe's definition of that word.

2.4.2. Mode-dependent code

Our mode system allows the programmer to attach more than one mode declaration to a procedure. In a way, this is just syntactic sugar; the compiler will analyze each mode separately and will generate separate code for each mode; in the object code every procedure has only one mode. However, this syntactic sugar is quite useful, for two reasons. One is that it allows us to view the procedure as a unit; we need not repeat the procedure declaration N times (where N is the number of modes) and think up a new name for each one. The second is a consequence of the first; when one multi-mode procedure wishes to call another, this syntactic sugar saves us the effort of writing multiple copies of the calling procedure, copies that differ only in which version of the other procedure they call.

Sometimes, however, code that works OK in one mode may not work satisfactorily or at all in another mode; the most common reason for this is the use of sequencing constructs, which cannot be driven backwards. In such cases we obviously cannot avoid writing down more than one version of the code. However, we would still like to call these variants by a single procedure name, so that we can keep on using a single version of the procedures that call this one. To allow us to do so, we use *when* clauses, whose syntax is demonstrated by example 2.3. The declaration in that example tells the compiler that for the first mode, it should generate code that executes $p1$ first, and that for the second mode it should generate code that executes $p2$ first (we assume that both $p1$ and $p2$ can themselves be driven backwards). We will encounter the need for this kind of differential sequencing in section 4.7. Note that if procedure p did not require strict sequencing, then we would not need separate procedure bodies; the mode analysis algorithm would by itself generate code that calls $p1$ and $p2$ in their reverse modes.

Programmers who use this feature should take care to ensure that the bodies of the various definitions actually represent the same logical relation. This should not be a problem if the bodies have the same logic and differ only in the control aspects; but unlike IC-Prolog [Clar79], we allow the bodies to have different logic as well. This can be useful when one wishes to employ different and possibly even unrelated algorithms for the different uses of a procedure. For example, while sorting algorithms are quite efficient at sorting lists in their forward modes (that's their job), they are not very efficient when generating permutations in their backward modes; they may even go into infinite loops [Shoh84]. The simplest way to prevent this is to make the body of

```
procedure p(X, Y: integer)
when mode X => Y;
def
        p1(X, Z) &> p2(Z, Y)
when mode Y => X;
def
        p2(Z, Y) &> p1(X, Z)
end append
```

Example 2.3: mode-dependent code.

the sort predicate in its backward mode a call to a specialist permutation generation program (presumably after checking that the input is in fact sorted).

2.4.3. Producer selection

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. Our syntax for this is a list of node names in brackets after a subexpression: this means that the subexpression should be the producer of the nodes named in the list.

The classic example is query optimisation for database programs. Suppose that p and q are database procedures that can produce all their arguments without any input, 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.

This is the reason why bare producer annotations in the style of the Relational Language [Clar81] do not work with multi-mode procedures. Therefore we allow programmers to associate a name with each mode declaration; they can then quote the appropriate name(s) with their producer annotations. If the X => Y mode of our code fragment is named *forward* and the Y => X mode is named *backward*, this annotation could take the form

```
p(X, Z) [forward: Z] & q(Z, Y) [backward: Z]
```

where the annotation follows the subexpression to which it applies. This scheme has the advantage that it extends easily to arbitrary first-order formulae. For example, the code

```
{ p1(X, Z) & {p2(X, Z) or p3(X, Z)} } [Z] & q(Z, Y)
```

tells the compiler that Z may be generated either by $p1$ or by both $p2$ and $p3$ in the arms of the disjunction, but should not be generated by q.

A program with producer annotations can simulate join AND-parallelism in our stream AND-parallel system. Given two procedures $p(A, B)$ and $q(A, B)$, both with mode => A, B, 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 AND-parallelism.

Producer annotations can be supplied directly by the programmer, or they can be derived automatically from other information. An automatic tool should be able to generate producer annotations if it knows the number of solutions typically produced by each mode of each production; this is an old optimisation technique for relational databases. The programmer is one source of the required information. However, there is a much better source: traces from instrumented runs of the program under consideration. The instrumented object code of course has to find out the number of solutions generated by each possible producer, so one way is to let them all generate the variable and then check the consistency of these solutions, just as in the join AND-parallelism case above. Another way is to compile procedure bodies according to more than one dataflow graph, time how long each takes (this can be difficult in a parallel system), and return the solutions produced by one of them (any one will do, since they should produce the same results). The technique of basing optimisation decisions on statistics derived from real runs of the program is already used in some of the more sophisticated compilers for conventional languages, e.g. [Chow86].

All of the above discussion is concerned with static producer selection. In some rare cases, the selection of producers may need to be based on information that is only available at run-time. We accommodate such cases by a simple technique illustrated by example 2.4.

2.4.4. Unrolled type trees

The definition of mode-correctness in our system applies only to type-correct programs, so the compiler should do mode analysis only after it has finished type-checking. The mode analysis algorithm should thus be able to assume that the types of arguments in procedure calls are compatible with the types of the formal parameters of the called procedure. However, this is not enough; the mode analysis algorithm requires a one-to-one correspondence between the type tree nodes of the actual and the formal parameters.

Consider the unification *List* = *cons*(*Head*, *List*1) 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*(*Head*, *List*1) it can easily derive the type tree shown in figure 2.3 for the right side of the unification. By the rules of infinite trees this type tree is equivalent to the standard one in figure 2.1. 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. 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.

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

```
select_producer(Runtime_data, Which) &
{
        Which = left  &> p(X) [X] & q(X)
or
        Which = right &> p(X) & q(X) [X]
}
```

Example 2.4: run-time producer selection.

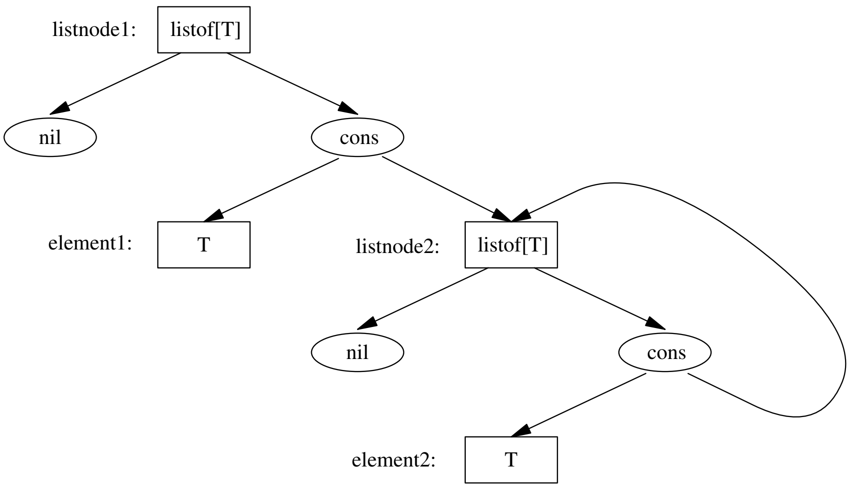

Figure 2.3: unrolling a type tree.

two arguments of the unification procedure). The compiler now issues instructions to perform a test for *listnode*1, assignments for all the *listnode*2's, and void unifications for *element*1 and the *element*2's, just as we said in section 2.3.

The rule for doing unrolling in procedure calls is quite simple. If the actual parameter is not a variable, but is a manifest functor, then do a single unrolling of the formal parameter, giving to the new nodes of its type tree the same tag as the nodes they were derived from; recursively repeat the process for each argument of the manifest functor. The unrolling rule for unification is similar, but unification has no fixed set of modes, so the tag manipulation is not necessary. Since actual arguments can have only finitely many manifest functors, the size of the unrolled tree must be finite. In practice it tends to be quite low as well; most actual parameters are in fact variables.

2.4.5. Circular modes

The mode assignment algorithm we gave in section 2.3.1 tries to find a *partial order* in which the calls in a conjunction may be conceptually evaluated, even though in reality the calls may be run in parallel, with the partial order representing only a communication constraint. For some programs, however, there is no such order; in particular there is no such order for programs that exploit the paradigm of incomplete messages.

Example Consider the code of Example 2.5, which contains a client procedure and a toy "fileserver" with one atomic piece of data. A typical invocation of these procedures would be like this, where *init* produces *P* and *D* and *final* consumes the output of the client:

```
... init(P, D), client(P, S, O), server(S, D), final(O) ...
```

This expression is well-moded. Every part of every variable has exactly one producer: the client generates commands and the server carries them out. However, the mode analysis algorithm we presented above cannot find the valid mode assignment, because the correct dependency graph contains a circularity: it apparently shows that the server cannot execute until given a command and the client cannot generate commands until it is given the value of $S.readdata$ by the server. Intuitively, the second half of this is false, but how can the system know this ?

```
procedure client(P: param; S: listof[iocmd[T]]; O: output)
% this mode declaration will be superseded in the text
% later in this section by a more accurate one
mode
        P, S.readdata => S, S.element, S.writedata, O;
def
        decide_what_to_do(P, D),
        {
        D = doread &>
                S = read(R).S1,
                next_param_r(P, R, P1),
                client(P1, S1, O)
or
        D = dowrite(W) &>
                S = write(W).S1,
                next_param_w(P, P1),
                client(P1, S1, O)
or
        D = dostop &>
                S = nil,
                compute_result(P, O)
        }
end client

procedure server(S: listof[iocmd[T]]; D: data)
% this mode declaration will be superseded in the text
% later in this section by a more accurate one
mode
        D, S, S.element, S.writedata => S.readdata;
def
        S = read(D).S1   &> server(S1, D)
or
        S = write(D1).S1 &> server(S1, D1)
or
        S = nil
end server
```

Example 2.5: client and server.

The answer lies in the fact that *S. readdata*, which is an INBOUND node of the client, has as its parent *S. element*, which is an OUTBOUND node. The compiler can notice this *tag inversion*, and figure out by itself that the client does not need *S. readdata* to start executing. If the compiler *assumes* that the node *S. readdata* is known and lets the client execute, the client will generate commands, and *then* the server will instantiate *S. readdata*, thus fulfilling the assumption.

The problem arises in the first place because mode declarations, as we have presented them so far, say only *what* arguments the procedure can generate, but do not have anything to say about *when*. The only reasonable assumption the compiler can therefore make is that a procedure may need all of its inputs before producing any of its outputs. This is frequently not the case (e.g. it is not the case for the client procedure) and the default assumption leads to apparent circularities.

If we wish to avoid the problems caused by these apparent circularities, we should allow mode declarations to be more precise about what inputs are actually required to compute each output. To this end, we introduce the notion of *mode segments*. Mode segments look like mode declarations, the difference being that they may not give a tag to every node of the type forest. A mode declaration, as we now redefine it, consists of one or more mode segments separated by a hash # sign. If a type tree node appears in more than one segment, those segments must give it the same tag; and every node that has an OUTBOUND or OUTFREE tag should appear in exactly one segment.

Mode segments also require a modification of our abbreviation rule; if a node of the type forest appears in none of the segments, then not only is it given the same tag as its most immediate ancestors (which must have the same tag or the program is in error), but it will be inserted into all the segments that its immediate ancestors appear in (if the immediate ancestors are also absent from all segments, we of course apply this algorithm to them recursively).

We will make mode segments, not whole mode declarations, the basic schedulable entities of our new mode analysis algorithm; when we schedule one mode segment of a called procedure, we will say that we are scheduling a *virtual procedure call*.

Example Consider the procedure p defined by the following code:

```
procedure p(A, B, C, D: T)
mode
        A => C # A, B => D;
def
        p1(A, C), p2(A, B, D)
end p
```

Example 2.6: a segmented mode.

The mode declaration states that p can compute C given only A, and that it needs both A and B to compute D. This behaviour presumably reflects the modes of $p1$ and $p2$. Consider the conjunction

```
... p(A, B, C, D), q(C, B) ...
```

where the mode of q is $C \Rightarrow B$, and where we assume that A is already known. The first feasible mode segment is the first mode segment of p; the second is the only mode segment of q, and the third is the second mode segment of p. In this case the mode segments of p correspond to single procedure calls in the body of p, so the virtual procedure calls correspond to real procedure calls one level down from the calling conjunction, but in general this need not be so; $p1$ and $p2$ could have been more complex expressions performing the same functions.

In cases like the one above, the notion of mode segments allows us to find the valid mode assignment without making any assumptions. However, it cannot by itself help us with code that conceptually demands an unbounded number of mode segments, such as the procedures *client* and *server*. To allow us to describe such procedures, we introduce the notion of *iterated mode segments*. An iterated mode segment is a description of an unbounded number of ordinary mode segments that all follow the same pattern. For example, the server's original mode declaration states that it can compute $S.readdata$ from all the other parts of its input; however, the server does not really need to know what the second command is going to be before carrying out the first

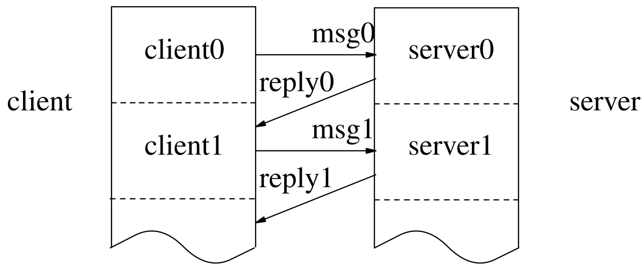

Figure 2.4: virtual calls.

command. We can describe this behaviour using iterated mode segments as follows:

```
foreach i: D, S/i, S.element/i, S.writedata/i => S.readdata/i;
```

Similarly, the mode of the client should be:

```
foreach i: P, S.readdata/i-1 => S/i, S.element/i, S.writedata/i
# P, S.readdata => O;
```

The iterated mode segments refer to the infinite tree obtained by unrolling the type tree of S an infinite number of times. What follows the slash is an expression denoting a *depth number*, which refers to the number of times one has to traverse the recursive arc of the type tree of *listof* in order to reach the node in question. These depth numbers start at zero and are cumulative, so that any reference to (say) $S.element/i$ also refers to $S.element/j$ for all $j < i$. Thus the mode declaration of the server states that for all values of i from zero upwards the server can compute the values of the *readdata* nodes of S down to depth i given the original value of D and the values of the S, $S.element$ and $S.writedata$ nodes at all depths down to depth i. The mode declaration of the client states that it can decide what the next message is going to be given the original parameter P and the server's reply to all previous messages, and that it can compute the value of O given the value of P and the values of *all* the replies from the server. When i is zero, $i-1$ is negative and is thus an invalid depth number, so $S.readdata/i-1$ corresponds to no node; this is what allows the client to start generating messages before the server starts producing $S.readdata$. Figure 2.4 shows the flow of data between generic clients and servers; in this case, *msg* stands for the nodes S, $S.element$ and $S.writedata$ while *reply* stands for the node $S.readdata$. From this diagram, one can easily see the sequence of virtual calls that the mode analysis algorithm should schedule: *client*0, *server*0, *client*1, *server*1 and so on.

That diagram also shows that although the conjunction that calls the client and the server has a circularity when viewed at the granularity of procedure calls, it has no circularity when viewed at the finest level of granularity (that of individual functors) because no piece of data depends on itself; one cannot arrive back at one's starting point if one follows the arrows in the diagram. This is good, because it implies that the implementation is justified in leaving out the occur check; it is never needed for code without data-level circularities (see section 3.5).

In this thesis we consider only those iterated mode segments whose non-constant depth numbers are of the form $i-1$ and i on the input side and i on the output side, or equivalently i and $i+1$ on the input side and $i+1$ on the output side. Mode analysis for this subset is relatively simple, while it could become quite complicated if we allowed more general forms. Fortunately, our subset includes all programs based on the client-server paradigm, and a good many besides.

If one did wish to extend our subset, the work on weighted graphs [Devi88] would come in useful.

We believe that the compiler's input should contain segmented mode declarations whenever appropriate. If the compiler finds it necessary to use a segmented mode, then the program text should contain this segmented mode; if it doesn't, then the mode declaration doesn't do its job as documentation. However, just as ordinary mode declarations can sometimes be automatically inferred from the whole program, so one should be able to derive segmented mode declarations from ordinary mode declarations and the code of the procedure. In fact the algorithm required is precisely the mode analysis algorithm; to discover mode segments one would give it strict subsets of the set of input nodes as the initial set of known nodes.

2.5. The full mode analysis algorithm

We now present the full version of our mode analysis algorithm. This algorithm assumes that a number of things have already been done when it is invoked.

Type checking. Our algorithm assumes that the procedure for which it is called has already been proven to be type-correct. It also assumes that the type trees of the left and right sides of unifications are identical and that the type trees of actuals and formals are also identical (see section 2.4.4). The latter may require us to use specialised copies of the mode declarations of called procedures, the specialisation consisting of unrolling the type trees of arguments, giving names to the new nodes and labelling them with the tags of the nodes from which they are derived.

Implication elimination. We assume that procedure bodies contain no implications operators; they should be eliminated as we discussed in section 1.2.

Quantification and scoping. Our algorithm assumes that all universal quantifiers have been eliminated by transformation into existential quantifiers, and that existential quantifiers themselves are represented only implicitly in the input. One part of the input of the algorithm is a representation of the expression that is the body of the analyzed procedure. This expression is represented by a tree whose interior nodes are labelled with various connectives: AND, OR, NOT, SEQ, GUARD, ALT, and whose leaves are labelled UNIFY (for calls to the special procedure =) or CALL (for calls to all other procedures). Each node of this tree may have associated with it a list of variables that are existentially quantified so that their scope is the subexpression whose root node they are attached to. We assume that local variables appear in exactly one of these lists; the names of local variables should be standardised apart, and variables that are not explicitly quantified by the user are quantified by the system at the smallest subexpression that contains all occurrences of the variable. To simplify the handling of GUARD and SEQ nodes, we also assume that no variables are quantified at OR and ALT nodes. If a variable would otherwise be quantified at a disjunction, we transform the tree to push the quantification into the arms of the disjunction, so that the tree $\exists X$*: {p(X) or q(X)}* would be transformed into $\exists X$*: {p(X)} or* $\exists Y$*: {q(Y)}*.

Producer selection. We associate with each node of the expression tree a set of nodes of the extended type forest of the procedure definition; this set enumerates the type tree nodes that the given subexpression is not allowed to generate. We use these *maynotgen* sets to implement producer selection annotations. A node may of course appear in only one producer selection annotation for each mode; we include the node in the *maynotgen* sets of every sibling of the subexpression to which the annotation applies, as well as to every sibling of every ancestor of that subexpression, except the ones that are outside the scope of the annotated variable. Since the annotations may be different for the various modes, our algorithm requires a separate copy of the procedure body for each mode; this is also needed for the proper handling of mode-dependent code (see section 2.4.2).

Our algorithm is written in a pseudocode that is intended to be a close approximation to how the algorithm would look like when expressed in Ptah. The heart of our algorithm is procedure *expr*. Its job is to take an expression and a list of known nodes and find out which parts of the expression can be feasibly scheduled and what other nodes they would thereby make known. We again remind the reader that our schedules represent only communication constraints and that they may have nothing to do with the actual order in which calls are started (they are more closely related to the order in which calls may *finish*). *Expr*, the first argument of procedure *expr*, represents of course the expression to be analyzed. The second, *Maygen*0, lists the type tree nodes visible from outside *Expr* that the expression is allowed to generate (it may of course generate any of its local variables). *Assumed* is a list of type tree nodes that we assume to be known for the present so that we can find valid mode assignments even for expressions with circular dataflow graphs. *Free* and *Void* are needed by procedure *unify*; they contain respectively a list of the nodes of the type forest that must not have producers and a list of the void variables in the body of the procedure being analyzed. *Known*0 is a list of the nodes of the extended type forest that are known upon entry to procedure *expr*; *Known*9 is a list of the nodes of the extended type forest that are known upon exit from procedure *expr*. *Known*0 and *Known*9 are employed as a difference list, with *Known*9 having *Known*0 as a final subsequence with any newly known nodes added to its front. The same is true for *Sched*0 and *Sched*9; they contain the starting and final versions of the list of scheduled mode segments.

The mode segments in the various versions of *Sched* may have to be constructed on the fly from the original mode declarations in the program. Consider the expression

```
p(A, X, Y) or { q1(A, X) & q2(B, Y) }
```

where procedure p has mode A => X, Y, procedure $q1$ has mode A => X and procedure $q2$ has mode B => Y, and assume that A is known but B is not. In this case we can make progress only by noting that the mode A => X, Y is equivalent to the mode A => X # A => Y, and scheduling the first of these mode segments along with $q1$. The disjunction thus generates a value for X; some other part of the procedure body will then compute B (perhaps from X itself), allowing us to schedule $q2$ and the other half of p. To simplify the handling of disjunctions, we will consider *all* mode declarations to consist of mode segments that each produce only one OUTBOUND node (they may produce OUTFREE child nodes as well).

2.5.1. Procedure expr

The body of procedure *expr* is essentially a recursive case analysis. We present the base cases first: the leaves of the expression tree.

For a segment of a call to be feasible, both its input and its outputs must satisfy certain conditions. The inputs must be already available; they may be in the set of type tree nodes that have already been assigned a producer (the set *Known*0) or they may be in the *Assumed* set, implying that some call will later make them known (this is checked in procedure *toplevel*). Alternatively, an input node may correspond to a functor textually present in the call (we call these manifest functors), or its occurrence may be made impossible by a manifest functor. For example, if a procedure requires as input both the *listnode* and *element* nodes of a list-valued argument, and the call specifies the functor *nil* as the argument, then the caller lives up to the producer's side of the bargain for the *element* node: if and when the value of the node is required, the producer will supply it. In this case, the caller "produces" the *element* node by making sure that its value will never be required. Formally, a node is *impossible* if one of its AND-node ancestors is a sibling of a functor manifest in the call; in this case the AND-node ancestor is the

```
procedure expr(Expr, Maygen0, Assumed, Free, Void,
               Known0, Known9, Sched0, Sched9)
def
Locals  = local(Expr)
Maygen1 = (Maygen0 - maynotgen(Expr)) ∪ Locals

case label(Expr) of

CALL:   choose one of the modes of the procedure being called
                such that the mode includes all the mode segments
                of the call that are already in Sched0
                and such that the mode has at least one segment
                whose input nodes are manifest or impossible
                        or in Known0 or in Assumed
                and whose output nodes are in Maygen1 but not in Known0
                        or are impossible
                % this is a backtrackable choice
        Known1 = Known0 ∪ (outputs of these mode segments)
        Sched1 = Sched0 ∪ (these mode segments)
        if Known1 = Known0 then
                Known9 = Known1
                Sched9 = Sched1
        else
                expr(Expr, Maygen0, Assumed, Free, Void,
                     Known1, Known9, Sched1, Sched9)
        end if

UNIFY:  unify(left_term(Expr), right_term(Expr), Maygen1, Assumed,
              Free, Void, Known0, Known1, Sched0, Sched1)
        if Known1 = Known0 then
                Known9 = Known1
                Sched9 = Sched1
        else
                expr(Expr, Maygen0, Assumed, Free, Void,
                     Known1, Known9, Sched1, Sched9)
        end if

AND:    expr(left(Expr),  Maygen1, Assumed, Free, Void,
             Known0, Known1, Sched0, Sched1)
        expr(right(Expr), Maygen1, Assumed, Free, Void,
             Known1, Known2, Sched1, Sched2)
        if Known2 = Known0 then
                Known9 = Known2
                Sched9 = Sched2
        else
                expr(Expr, Maygen0, Assumed, Free, Void,
                     Known2, Known9, Sched2, Sched9)
        end if
```

```
SEQ:    expr(left(Expr), Maygen1, Assumed, Free, Void,
             Known0, Known1, Sched0, Sched1)
        if all of left(Expr) is scheduled or covert in Sched1 then
                expr(right(Expr), Maygen1, Assumed, Free, Void,
                     Known1, Known9, Sched1, Sched9)
        else
                Known2 = Known1
                Sched2 = Sched1
        end if
        if Known2 = Known0 then
                Known9 = Known2
                Sched9 = Sched2
        else
                expr(Expr, Maygen0, Assumed, Free, Void,
                     Known2, Known9, Sched2, Sched9)
        end if

GUARD:  expr(left(Expr), Locals, Assumed, Free, Void,
             Known0, Known1, Sched0, Sched1)
        if all of left(Expr) is scheduled or covert in Sched1 then
                expr(right(Expr), Maygen1, Assumed, Free, Void,
                     Known1, Known9, Sched1, Sched9)
        else
                Known2 = Known1
                Sched2 = Sched1
        end if
        if Known2 = Known0 then
                Known9 = Known2
                Sched9 = Sched2
        else
                expr(Expr, Maygen0, Assumed, Free, Void,
                     Known2, Known9, Sched2, Sched9)
        end if

OR:
ALT:    % assert Locals = empty
        expr(left(Expr),  Maygen1, Assumed, Free, Void,
             Known0, Known1a, Sched0, Sched1a)
        expr(right(Expr), Maygen1, Assumed, Free, Void,
             Known0, Known1b, Sched0, Sched1b)
        if Known1a ∩ Maygen1 = Known1b ∩ Maygen1 then
                Known9 = Known1a ∪ Known1b
                Sched9 = Sched1a ∪ Sched1b
        else
                Maygen2 = (Known1a ∩ Known1b) ∩ Maygen1
                expr(left(Expr),  Maygen2, Assumed, Free, Void,
                     Known0, Known2a, Sched0, Sched2a)
                expr(right(Expr), Maygen2, Assumed, Free, Void,
                     Known0, Known2b, Sched0, Sched2b)
                % assert Known2a ∩ Maygen1 = Known2b ∩ Maygen1
```

```
                Known9 = Known2a ∪ Known2b
                Sched9 = Sched2a ∪ Sched2b
        endif

NOT:    expr(left(Expr), empty, Assumed, Free, Void,
             Known0, Known9, Sched0, Sched9)

end case
end expr

procedure unify(Left, Right, Maygen, Assumed, Free, Void, then
                Known0, Known9, Sched0, Sched9)
def
if Left is variable and Right is variable then
        if for some nodes Left is in Known0 or in Assumed
        and Right is in Maygen but is not in Known0 then
                Known1 = Known0 ∪ Right's nodes
                Sched1 = Sched0 ∪ (these assignment segments)
        else
                Known1 = Known0
                Sched1 = Sched0
        end if
        if for some nodes Right is in Known0 or in Assumed
        and Left is in Maygen but is not in Known0 then
                Known2 = Known1 ∪ Left's nodes
                Sched2 = Sched1 ∪ (these assignment segments)
        else
                Known2 = Known1
                Sched2 = Sched1
        end if
        if for some nodes both Left and Right are
        in Known0 or in Assumed then
                Known3 = Known2
                Sched3 = Sched2 ∪ (these test segments)
        else
                Known3 = Known2
                Sched3 = Sched2
        end if
        if for some nodes Left is in Free and Right is in Void then
                Known4 = Known3
                Sched4 = Sched3 ∪ (these void segments)
        else
                Known4 = Known3
                Sched4 = Sched3
        end if
        if for some nodes Right is in Free and Left is in Void then
                Known5 = Known4
                Sched5 = Sched4 ∪ (these void segments)
        else
                Known5 = Known4
```

```
                Sched5 = Sched4
        end if
        Knonw9 = Known5
        Sched9 = Sched5
else if Left is manifest and Right is variable then
        if root(Right) is in Maygen but is not in Known0 then
                Known1 = Known0 ∪ root(Right) ∪
                        (the nodes in Right made impossible by Left
                        except the ones not in Maygen)
                Sched1 = Sched0 ∪ (this mode segment)
        else if root(Right) is in Known0 then
                Known1 = Known0 ∪
                        (the nodes in Right made impossible by Left
                        except the ones not in Maygen)
                Sched1 = Sched0 ∪ (this mode segment)
        end if
        unify_args(args(Left), args(Right), Maygen, Assumed,
                   Free, Void, Known1, Known9, Sched1, Sched9)
else if Left is variable and Right is manifest then
        symmetrical to the case above
else if Left is manifest and Right is manifest then
        test that root(Left) = root(Right)
        Sched1 = Sched0 ∪ (this test unification segment)
        Known1 = Known0
        unify_args(args(Left), args(Right), Maygen, Assumed,
                   Free, Void, Known1, Known9, Sched1, Sched9)
end if
end unify

procedure unify_args(Left, Right, Maygen, Assumed, Free, Void,
                     Known0, Known9, Sched0, Sched9)
def
if Left = Lhead.Ltail and Right = Rhead.Rtail then
        unify(Lhead, Rhead, Maygen, Assumed, Free, Void,
              Known0, Known1, Sched0, Sched1)
        unify_args(Ltail, Rtail, Maygen, Assumed, Free, Void,
                   Known1, Known9, Sched1, Sched9)
else
        % assert Left = nil and Right = nil
        Known9 = Known0
        Sched9 = Sched0
end if
end unify_args

procedure toplevel(Proc)
def
foreach Mode in Proc
        Expr = body for Mode in Proc, with maynotgen sets
                to implement producer annotations
        compute Void from scan of Expr
```

```
        get Allinbound, Allinfree, Alloutbound, Alloutfree from Mode
        Free = Allinfree ∪ Alloutfree
        compute Mayassume from tag inversion, directives
        foreach Segment in Mode
                get Inbound, Infree, Outbound, Outfree from Segment
                choose a subset of Mayassume, call it Assumed
                % this is a backtrackable choice
                expr(Expr, Alloutbound, Assumed, Free, Void,
                     Inbound, Known, empty, Sched)
                test that all elements of Outbound are in Known
                test that all elements of Assumed are in Known
        end foreach
        test that the various values of Sched are compatible
                and that their union contains all the segments
                of the chosen modes of the called procedures
end foreach
end toplevel
```

The mode analysis algorithm.

cons node and its manifest sibling is the *nil* node in the type tree of lists. Note that due to the unrolling of type trees whenever manifest functors occur in calls, OR-nodes descended from the AND-node of the manifest functor cannot be made impossible by this definition.

The other requirement that feasible calls must satisfy is that the call must be allowed to instantiate the outputs of the chosen mode. This means that the segment's outputs must correspond to type tree nodes that have not been assigned a producer yet (are not in $Known0$) and that the procedure is otherwise allowed to generate (and therefore are in $Maygen1$). (A different mode analysis algorithm may delete nodes from *Maygen* sets as they are put into *Known*, but we have not chosen to do so.) Alternatively, the outputs of the mode may be nodes made impossible by manifest functors in the call; e.g. a procedure *iota* that takes as input the skeleton of a list and generates the elements of the list itself (by binding them to consecutive integers beginning with zero) fulfills the producer's side of the bargain for the *element* node even if the call specified *nil* as the actual parameter.

Another constraint on the scheduling of mode segments is that we are really trying to find a mode for the called procedure; once we have scheduled some segments from a mode, we should not schedule any segments that do not belong to that mode. Note that this does not mean that we should not schedule segments that belong to other modes, because some segments may legitimately belong to more than one mode. For example, as we discussed above, the mode $A => X, Y$ has two segments, $A => X$ and $A => Y$. Having scheduled $A => X$, we must of course be able to schedule $A => Y$ later, even though it also belongs to the implied mode $A, X \# A => Y$.

Once we are sure that a call has some mode segments that satisfy the necessary requirements, we may add them to the set of scheduled segments and their outputs to the set of known nodes. These additions to the set of known nodes (if there are any) may allow us to schedule some more segments, so *expr* must call itself recursively with its partial results to ensure that it schedules all the mode segments that can possibly be scheduled. For example, a procedure $p(A, B, X, Y)$, with the mode segments $A => X$ and $B => Y$, may be called as $p(a, X, X, Y)$. Scheduling the first mode segment puts X into the *Known* set; the recursive call is necessary to schedule the second mode segment and put Y into the *Known* set as well. Note that the recursive call passes *Maygen*0 instead of *Maygen*1 to maintain the invariant that the second parameter contains only nodes visible from outside the expression in the first parameter; but the recursive call will nevertheless set its own copy of *Maygen*1 to the same value as its parent.

UNIFY nodes conceptually require the same processing as CALL nodes, but we need to handle them separately because the special procedure = doesn't have a fixed set of modes as such; the modes that apply at the various nodes of the type trees of the arguments are completely independent. We will discuss this further in section 2.5.2.

The handling of AND nodes is quite simple: recursively process the two subexpressions, passing partial results from the left side to the right side, and recurse on the whole expression if there any new known nodes in case the left side can schedule some segments based on values made available by the right side.

Sequential conjunctions are more complex: we must make sure that no deadlock will arise from the delaying of the right subexpression until the completion of the left subexpression. This means that we must not schedule any segments from a right half until we are sure that the left half can proceed to completion. We must therefore wait until all segments of the left half have been scheduled, except in cases that involve what we call *covert unification*. The unification $S = read(D).S1$ in procedure *server* of example 2.5 contains an example of covert unification. The mode of procedure *server* tells us that all parts of S are already known except for the *readdata* node. The value of the known variable D supplies the value for this node for the first element of the list S, but the values for the other elements have to come from $S1$. The values for this node of $S1$ are generated on the other side of a sequential AND operator, so if we had to wait until all of the left side was scheduled before scheduling the right side then we would wait forever. However, the unification of those nodes of S and $S1$ is a covert unification, because $S = read(D).S1$ can run to completion without instantiating the *readdata* nodes of the later elements of S. The code emitted for the unification binds $S1$ to the tail of S, and the flow of data from the *readdata* nodes of $S1$ to S takes place later, through the sharing of memory locations and without the execution of further instructions; hence the name.

If it weren't for covert unifications, we would not need to recurse on the whole expression itself; because then there would be no communication from the right subexpression to the left subexpression. Communication from the left to the right subexpression is taken care of by passing *Known*1 to the second call to *expr*; any values the subexpressions produce for themselves are taken care of inside the calls to *expr* for those subexpressions. However, because of covert unifications, we must try to see whether we can schedule covert segments in the left side after the right side generates some new values. In the case we have been discussing, when the right side generates $S1.readdata$ we can go back and schedule the covert segment that produces the *readdata* node for the later elements of S.

Our handling of GUARD nodes is almost exactly the same as our handling of SEQ nodes, with one important exception: the left subexpression of a GUARD node is allowed to generate values only for variables local to the GUARD expression itself. This restriction ensures guard safety: the left child of the GUARD node (the guard itself) may export values only to the right child (the body), and not directly to the code surrounding the GUARD expression. This allows an implementation to execute the guards in (pseudo) OR-parallel without any need for a multiple environment mechanism. Note that our analysis of SEQ nodes assumes that their expressions will be evaluated OR sequentially. If we wish to evaluate in OR-parallel expressions in which OR and SEQ nodes take the place of ALT and GUARD nodes, then we would have to modify our handling of SEQ nodes to reflect this. This modification would make the handling of SEQ nodes identical to the handling of GUARD nodes.

Our approaches to OR and ALT nodes are identical. Their main problem is that if one arm of the disjunction produces values that the other arm cannot produce, then the disjunction as a whole does not behave in a strongly-moded fashion: sometimes it will produce the value, sometimes it won't. (This problem exists regardless of whether the implementation is OR-parallel or OR-sequential.) We must therefore avoid any such situation. First we see what nodes the two arms of the disjunction can produce. If they produce the same set of visible variables then the disjunction behaves well, and we can just take the union of the scheduled segments. Note that $Known1a$ and $Known1b$ may each contain variables local to their subexpressions which therefore will not appear in the other; the disjunction is nevertheless well-moded as long as $Known1a$ and $Known1b$ contain the same set of variables from $Maygen1$.

If the two arms of the disjunction produce different sets of visible variables, then we must content ourselves with the disjunction producing only the intersection of these two sets. We thus analyze the two subexpressions again with a smaller *Maygen* set that reflects this constraint. This time, *Known*2*a* and *Known*2*b* *must* contain the same set of visible variables: they must both consist of *Maygen*2 and a set of variables local to their subexpressions.

As we mentioned above, we assume that no local variables are quantified at OR and ALT nodes. If we did not push the variables that would be otherwise so quantified into the left and right subexpressions, then we would need to add another argument to *expr*, to distinguish type tree nodes that may be instantiated by any part of the first argument from those that may be instantiated in the body part of GUARD nodes but not in their guard part.

Procedure *expr* handles negation by a single recursive call to analyze the negated expressions. The only constraint is that the negated expression is not allowed to instantiate any variables not local to it; this is a safety condition necessary to ensure the soundness of negation. This constraint has to apply even to variables that the user explicitly quantifies at the negation node, as in $\exists X$: *˜p(f(X))*; this expression can be well-moded only if p does not care about the X part of its argument (i.e. declares it to be INFREE). In that case, $p(f(X))$ can be true for one X only if it is true for all X, and hence the semantics of $\exists X$: *˜p(f(X))* is the same as that of $\forall X$: *˜p(f(X))* and therefore that code is equivalent to ˜$\exists X\colon\ p(f(X))$.

The performance of our algorithm could possibly be improved by checking at the start of procedure *expr* to see whether all segments of the argument expression *Expr* have already been scheduled (are in *Sched*0). If the answer is yes, then we can set *Known*9 to *Known*0, *Sched*9 to *Sched*0, and return without any further processing; this will happen reasonably frequently because of the checks for backward flow of information in the conjunctive cases. Whether an all-scheduled test is worthwhile depends on how fast the test is. One fast implementation passes around a tree that shadows the structure of *Expr*, with leaves corresponding to completely scheduled subexpressions; this would require us to add to procedure *expr* a pair of arguments representing the old and new values of this tree.

2.5.2. Procedure unify

The arguments of *unify* are analogous to the arguments of *expr*, with the exception that the code to be analyzed is given not as one argument representing an expression but as two arguments representing a unification. Like procedure *expr*, procedure *unify* is also a case analysis: the four cases are distinguished by whether the left and right terms are manifest functors or not. Procedure *unify* is organised as a recursive procedure whose base case occurs when neither argument is manifest. When analyzing the unification *List* = *cons*(*Head*, *List*1) in procedure *length* (which we discussed back in section 2.2), we first call *unify* with the arguments *List*. *listnode*1 and *cons*(*Head*, *List*1), then with the arguments *List*. *element*1 and *Head*, and then with the arguments *List*. *listnode*2 and *List*1 (see figure 2.3 in section 2.4.4 for the type tree from which these node names are taken). The last two calls involve no manifest functors, and so they lead to no more recursive calls to *unify*.

Calls that do not involve manifest functors are quite simple to handle, although the code is rather long. We simply look at what nodes can be input on one side and output on the other, and schedule assignment unifications at those nodes; we look at what nodes can be input on both sides, and schedule test unifications at those nodes; and we look at what nodes are free on one side and void on the other, and schedule void unifications at those nodes and (conceptually) at their descendants. We call the segments whose inputs are not required for the completion of the emitted code *covert segments*; in practice this should mean all segments except possibly the one for the root of the tree, whose input(s) the implementation may require to be known (e.g. for optimisation reasons, or for use as the basis of pointer direction rules).

The second case, where the left argument is a manifest functor and the right corresponds to a node that is part of a variable, is more complex. If the root node of the right argument is not yet known, then this unification will generate it; otherwise we schedule a test unification instead. In both cases, the unification can (and will if it is allowed to) conceptually generate all the nodes in *Right*'s type tree whose occurrence has been made impossible by the unification. E.g. the unification *List* = *nil* can conceptually generate the node *List*. *element*, as we discussed in the previous section. After handling the root node, we recurse on the arguments.

The third case is a mirror image of the second. The fourth case, where both arguments are manifest functors, is in a way the simplest one. We test whether the two functors are the same; if they are, we just do what we would do for a conjunction of unifications that equate pairs of the arguments of the functors. However, if the two functors are in fact different, then the program contains an error and we allow procedure *unify* to fail. The rest of the compiler should catch this and issue an appropriate error message after diagnosing the trouble. (A version of the mode analysis algorithm intended for production use would probably do much more error handling than the version we present.)

2.5.3. Procedure toplevel

The task of procedure *toplevel* is to supply procedure *expr* with the parameters it requires. It derives its information from the text (or other representation) of the procedure it is asked to analyze. The most basic information is the list of modes of the procedure; *toplevel* analyzes each of these modes separately.

The first thing we do when we analyze a mode is extract from the text of the procedure the expression that implements the procedure for that mode. Most of the time, the body expressions will be the same for all modes, but not always. In some cases, the programmer may give different and possibly even unrelated expressions as the bodies for the various modes; we discussed this in section 2.4.2. Also, the programmer may require us to create variants of procedure bodies by specifying mode-dependent producer annotations (see section 2.4.3); we discussed how to modify expressions to handle these annotations at the beginning of section 2.5. Once we have the body expression, we scan it to find all of its void variables.

We view each mode as consisting of a number of mode segments, though modes usually have only one segment. As we discussed in section 2.4.5, the segments must agree in what tags they assign to the various nodes of the type forest; our first task after finding the body expression is to compute these tags. We treat INFREE and OUTFREE tags the same in our analysis: the only thing the expression may do with a node with either tag is unify it with a void variable. Since these nodes are not in *Alloutbound*, they will not be in any version of *Maygen* in procedure *expr*, and hence will not be given producers. As for their appearances in procedure calls, the tags of their parent nodes (INBOUND and OUTBOUND respectively) ensure that INFREE nodes appear only in INFREE positions and that OUTFREE nodes appear only once in an OUTFREE position.

The next step is to determine which nodes we should allow procedure *expr* to assume to be known. In theory, we may allow any output node and any node of a local variable to be assumed; we will later check whether the assumed nodes in fact have generators. However, if we follow this course then we would accept as well-moded many programs with circularities that lead to deadlock. E.g. if $p(X, Y)$ can compute Y from X and $q(X, Y)$ can compute X from Y, then $p(X, Y)$, & $q(X, Y)$ is theoretically well-moded, but the two calls will most likely delay forever, each waiting for the other to produce data. Our purpose in restricting the set of nodes that may be assumed is to detect such bugs at compile time. We put a node in the *Mayassume* set only if we find that it is involved in a tag inversion (as we discussed in section 2.4.5), or if the programmer explicitly directs us to do so via a compiler directive or *pragma*.

Most of the work of processing a mode segment is done by procedure *expr*; procedure *toplevel* merely sets up its inputs and checks its outputs. The most important input is the set of initially known nodes; we put the INBOUND nodes of the mode segment in this set. The most important output is the set of nodes that the procedure body can compute from these. This set, *Known*, must include all the OUTBOUND nodes of the mode segment under examination and all the nodes that procedure *expr* assumed in its analysis. It is because of the latter constraint that we do not necessarily pass all of *Mayassume* to procedure *expr*; we want to make sure that whatever nodes are consumed by the part of the body that is scheduled for the mode segment are also produced in that part of the body. Note that we allow this part of the body to produce output nodes that are named in other mode segments; this is necessary because some outputs may be needed as inputs in subcomputations that generate other outputs.

Our algorithm as we presented it gives each invocation of procedure *expr* the empty set as the set of initially scheduled segments in the body. It then checks that the sets of body mode segments scheduled by the various invocations of *expr* are in fact compatible (i.e. they choose the same mode for each procedure in the body). For efficiency, this constraint should be applied earlier so that procedure *expr* never picks a mode that conflicts with mode segments assigned earlier. We did not write our algorithm to do so because we wish to emphasise the unordered nature of the set of mode segments that make up a mode.

The other check that we must make at the end of procedure *toplevel* is to see that the union of the scheduled body mode segments exhausts the set of body mode segments available to be scheduled in the chosen modes of the called procedures. A procedure can fail this test only if it contains irrelevant code, since when we apply this test we have already scheduled producers for all of the procedure's outputs.

One may verify by inspection that if our preconditions are satisfied our algorithm only ever produces mode assignments that are valid by the rules of section 2.3. The rules about the appearances of FREE nodes are enforced indirectly, by restricting the appearances of their parent nodes. Our algorithm terminates because the number of mode segments in an expression is finite (see the next section for why this is true even for iterated segments), and because an invocation of procedure *expr* recurses only if it could schedule some segments.

Our algorithm will find a valid mode assignment if one exists if all the output nodes of the mode declaration are assumed initially to be known. However, such an assumption causes our algorithm to pass as correct programs that will always lead to deadlock, so by default we assume as few nodes as possible to be initially known. This ensures that programs that may lead to deadlock are flagged as erroneous at compile time unless the programmer explicitly directs the compiler to assume some nodes to be known in its mode analysis, thereby indicating that he or she is aware of the possibility of deadlock.

2.5.4. Handling iterated mode segments

Our algorithm can handle iterated mode segments with minor changes in housekeeping detail; the main change is that we must keep track of the depths of nodes.

We analyze iterated mode segments by splitting them into two if necessary: one segment should correspond to the case (where output depth numbers are zero) and the other to the induction step (where output depth numbers are greater than zero). For example, we analyze procedure *server* from example 2.5 of section 5.4.2 as if its mode declaration had been written as

```
D, S/0, S.element/0, S.writedata/0 => S.readdata/0;
# foreach i: D, S/i+1, S.element/i+1, S.writedata/i+1 =>
        S.readdata/i+1;
```

instead of the original

```
foreach i: D, S/i, S.element/i, S.writedata/i => S.readdata/i;
```

Procedure *toplevel* will analyze the two segments in turn. The second analysis is a superset of the first, because the inputs of the second segment include all the inputs of the first (remember that depth numbers are inclusive). The first analysis verifies that given D, $S/0$, $S.element/0$, and $S.writedata/0$, the code of the server can in fact generate $S.readdata/0$. The first clause generates $S.readdata/0$ by assigning it the known value of D; the others conceptually generate it because they guarantee its non-occurrence in S by testing for the presence of functors from which $S.readdata$ is not reachable. The third clause can in fact conceptually generate values for all the $S.readdata$ nodes, not just the first one, by making them all impossible, but the other two clauses cannot, and so the disjunction as a whole cannot.

The second analysis schedules all the segments scheduled by the first, and then it goes further: it can schedule the recursive calls to *server* in the first two clauses. The mode of the called procedure (which does not in general have to be *server* itself) assures procedure *expr* that it can generate $S1.readdata/i$ given $S1/i$, $S1.element/i$, $S1.writedata/i$, and the second argument (D or $D1$), and all of these are in fact available. The second argument was known even in the first analysis; the nodes $S1/i$, $S1.element/i$, and $S1.writedata/i$ get their values from $S/i+1$, $S.element/i+1$, and $S.writedata/i+1$ respectively, through the unification of the tail of S with $S1$. The same unification also generates the *readdata* node of later elements of S via a covert unification with the *readdata* node of $S1$.

The first analysis proves that *server* can generate the *readdata* node of the first element of S using only the input nodes for that element (not needing the values of the input nodes for other, later elements). The second analysis allows us to conclude that this is so for all the elements of S. The recursive call serves us as the induction hypothesis, and our analysis shows that if *server*'s mode declaration is valid down to depth i, then it is also valid to depth $i+1$, completing the induction. In the case of mutually recursive procedures, the picture looks a bit more complicated to humans, but we get the same results without changing the algorithm.

The case of the client procedure is a bit more complicated, but the same principles apply. The declared mode of the client is

```
foreach i: P, S.readdata/i-1 => S/i, S.element/i, S.writedata/i
# P, S.readdata => O;
```

According to our rule, we analyze it as if it were written

```
P => S/0, S.element/0, S.writedata/0
# foreach i: P, S.readdata/i =>
        S/i+1, S.element/i+1, S.writedata/i+1
# P, S.readdata => O;
```

explicitly deleting the node $S.readdata/i-1$ when i is zero. The analysis of the first two segments goes pretty much as the analysis of procedure *server*, except that we can only schedule the segment *foreach* i: P, $S.readdata/i-1$ => S/i, $S.element/i$, $S.writedata/i$ for the recursive calls. We can schedule the segment P, $S.readdata$ => O in the recursive calls, together with the call to *compute_result*, only when we analyze that segment in the definition of *client*.

We now turn to the other side of the coin: how do we handle calls to procedures with iterative modes in the bodies of other procedures that do not care about the iteration ? The answer is we analyze such calls as if the iteration were not there, except for pre- and post-processing.

The pre-processing consists of adding nodes to the *Mayassume* set. The nodes we add are those that appear in the input side of the mode declaration of one of the called procedures with a depth number less than the (uniform) depth number of the output side. There are no such nodes in procedure *server*, but there is one in procedure *client*: the node $S.readdata$. When processing the conjunction

```
... p(P, D), client(P, S, O), server(S, D), q(O) ...
```

where p generates the initial values of P and D, we analyze *client* and *server* as if their mode declarations were written as they were in example 2.5: without any iterated segments (in this case without any segmentation at all). By putting $S.readdata$ in the *Mayassume* set, we allow *client* to go ahead and generate the other parts of S; the call to *server* will then fulfill the assumption by generating $S.readdata$.

The post-processing takes as its input the output of the main mode analysis algorithm, and tests whether conjunctions involving calls to procedures with iterated modes contain any data-level circularities. (Circularities in the dataflow graph must involve conjunctions, although not necessarily the conjunction of calls; disjunctions do not allow the exchange of information between their alternatives.) The post-processing consists of simply running the basic mode analysis algorithm of section 2.3.1 on the base cases of the iterative modes involved in circularities, restricted to the nodes involved in the circularities. In the above call, the base cases are

```
S/0, S.element/0, S.writedata/0 => S.readdata/0
```

for the server and

```
=> S/0, S.element/0, S.writedata/0
```

for the client. Obviously, we schedule the base case of the client first, yielding a schedule without circularities at this level. The regular pattern of iterative modes tells us that the pattern of the base case must be repeated at deeper levels as well; we can therefore conclude that the conjunction contains no data-level circularities at all.

Chapter 3

Software engineering issues

This chapter starts out with a discussion of debugging, covering both static and dynamic tools as they apply to Ptah. The next four sections consider the broader implications of our approach; they cover in turn the implications of the type system, the implications of the mode system, implications for program length and theoretical implications. Although everything we say is derived from our own experience, we include some quotes from other people to show that our point of view is shared by others in the field. We then compare our mode system with other related mechanisms both outside and within logic programming.

3.1. Debugging

The main tools for debugging programs in conventional languages (such as C) are program analyzers such as *lint* and *cchk*, and the tracing and breakpoint facilities of debuggers such as *adb* and *dbx*. While such tools can be used to debug Prolog, their application to logic programming is not without problems.

3.1.1. Static debugging tools

Few logic programming environments provide static debugging tools of even the most rudimentary kind. The basic reason for this is that such systems function much better when the programs they process have a reasonable degree of redundancy, and such redundancy has so far been missing. Most logic programming systems provide few ways for the programmer to declare the properties of his or her programs. With the exception of the descendants of IC-Prolog (the Relational Language and Parlog) and of Trilogy, no logic programming language we know of actually *requires* such declarations. The majority of programmers do not bother to write declarations even when the language allows them to do so. This state of affairs has persisted even though its folly was pointed out long ago [Bruy82].

Some classes of errors in logic programs can be detected without help from the programmer. The errors reported by *nit*, the so-called NU-Prolog Incompetence Tester [Thom86], include: predicate called but not defined; predicate defined but not called; and variables that occur only once in a clause. Other classes of errors, though, especially the ones more related to semantics than syntax, cannot be reliably detected in the absence of declarations. Type inference systems, for example, cannot derive the intended types of the second and third arguments of *append*, [Mish84, Zobe87].

The basic problem is that the code in some clauses of a Prolog program (e.g. the non-recursive clause of append) works even if the input is ill-typed according to the programmer's intuition (i.e. the code computes a superset of the intended model). In the absence of declarations, the type inference system can at most make a worst-case assumption (e.g. that the programmer really intended *append*([], *a*, *a*) to succeed) and propagate the effects of that assumption to the rest of the program. Different type systems handle such situations in different ways. Some make the assumption, thus assuming that the program is well-typed unless it can be proven otherwise. Others insert checks to be executed at run-time [Zobe87, Nais87]. We prefer a third approach: declaring the program to be ill-typed unless the type system can prove it to be well-typed [Mycr84]. This may require a bit more up-front work from the programmer, but it will rule out the possibility that the program will keel over one day with a type error.

The problems of type inference systems working on languages without declarations pale into insignificance beside the problems that beset mode inference systems [Mell81, Debr86b, Marr88a] on the same languages. Since they cannot in general count on the analyzed program being well-typed, they cannot rely on any patterns being present in the set of possible values of each variable. To ensure termination, they must therefore be content with even more crude approximations to binding states. Whereas the approximation domain for type inference systems is quite rich (usually the set of regular trees), the corresponding domains for mode inference systems contain only a small number of elements. In [Debr86b] the domain is the set {empty, closed, free, don't-know}. Even the richest domain used in mode inference (that we know of) has only six elements; it is the one in [Mell85]:

o	nothing known about the mode
-	always free
+-	always bound to a structure whose arguments are all free
++	always bound to a ground term
+	always bound to a term (any internal structure)
?	sometimes bound, sometimes not

It would be difficult to augment this set of states in the absence of reliable type information. A term can have only one main functor, but this functor may in general have any arity; a term may therefore contain any number of non-main functors. Since the analyzer doesn't know where in the term these functors may be, it cannot reason about the binding states of the corresponding parts of variables. The best it can do is reason about some predefined structure imposed upon terms from the outside (e.g. "the functors of this term will be bound to at least depth k").

None of these problems arise in Ptah, because of the information supplied by declarations. When working on a procedure, type and mode checkers may assume that the declarations for other procedures are correct; if they are not, the error will be discovered later, when the errant procedure is checked. This is true even when the type or mode checker uses an inference algorithm to carry out its checks, by comparing the declared type/mode with the inferred one. The difference is that a pure inference algorithm cannot rely on the correctness of its approximations to the types or modes of other procedures, as they may depend on the type or mode of the procedure it is working on. An inference algorithm thus requires at least two passes over the program, whereas a checker requires only one pass. Both checkers and inference systems can work on a single module as long as they have information about the procedures declared in other modules. However, if this is to work for every module, we must have information about every module; if this information were available in an inference system, then one should be using a checker instead of an inference system anyway. The one exception we make to this principle is that we use the declaration of other procedures (whether or not they are defined in the current module) to infer the types of local variables and save the programmer some typing.

3.1.2. Dynamic debugging tools

The earliest dynamic debuggers for Prolog were direct equivalents to debuggers for conventional languages (such as *adb* and *dbx*). Like their models, they provided facilities for the tracing of execution and for the control of breakpoints. However, their operation is conceptually complicated by the potential presence of backtracking. Four-port debuggers [Byrd80] control this complexity by providing a clean description of backtracking behaviour, but tracing a logic program is still not as easy as tracing (say) a C program, due to the presence of "blind alleys", paths of the computation that do not contribute to the final solution.

The cleaner semantics of logic programs, however, compensates for this by a permitting a completely new style of dynamic debugging. This style, nowadays called *declarative debugging*, was first introduced by [Shap82] under the name *algorithmic debugging*, and has since developed into an active research area [Lloy86a, Lloy86b, Yan88, Av-R84, Take86, Nais88b, Ferr87, Hunt87, Pere86, Nich88].

The basic idea is that if a call computes an incorrect success set (that is, succeeds with a wrong answer or misses a solution) then either the definition of the called procedure is in error or one of the calls in that definition computes an incorrect success set. If the debugger has access to an *oracle* who knows the intended semantics (success set) of every procedure, then it should be able to locate the incorrect procedure/clause by traversing the computation tree. (This works only if the program terminates; the diagnosis of errors leading to infinite loops requires somewhat different techniques.) In most systems, the oracle is the programmer, though in some cases the role could be automated [Drab88]. Since programmer time is expensive, the aim of the debugger should be to ask the oracle as few questions as possible. Several of the papers above have presented heuristics to try to achieve this end.

One of these heuristics, the one presented in [Pere86] under the name *rational debugging*, is particularly well suited to moded systems. In our (biased) opinion it is also one of the most useful heuristics published so far, because it tends not to ask irrelevant questions. The algorithm asks the user to point out a term in the computed solution as being wrong, and then follows the dependencies of this term back in the computation tree. The idea is that if a variable is bound wrong, then either there is something wrong with the procedure that computed its value or the inputs of that procedure are themselves wrong. (Notice the resemblance of this idea to the basic idea of declarative debugging.) In Pereira's original approach, the dependency information is generated by a modified unification algorithm; in a declarative debugger for Ptah it would of course be derived from mode declarations. This would eliminate the run-time overhead, and reduce the amount of memory required to store the dependency information. The reason is that in Ptah the dependencies must have a regular structure, whereas in Prolog different calls to a procedure may have different dependencies. (This problem also plagues the application of the dependency-tracking idea to conventional languages [Kore88].)

Ptah also has some other characteristics that should make it easier to implement a declarative debugger for it. The first one is the lack of hard side-effects. A declarative debugger for full Prolog needs to cope with the possibility that the program may be modified during execution. To be sure that it executes every goal with respect to the correct version of the program, the debugger must traverse the computation tree in the standard Prolog execution order, although it may relax this order within subtrees that do not modify the program. The more calls to *assert*/*retract* the program contains, the closer the declarative debugger must approximate the behaviour of the trace/spypoint/breakpoint debuggers, and the less effective it becomes.

An alternative implementation technique for a declarative debugger for full Prolog is to execute the program once, yielding a computation tree whose nodes contain not just goals but also the contents of the database at the time the goal was invoked. This approach is clearly infeasible for long-running programs even if the implementation does considerable storage optimisation. (The need to keep many copies of the environment is one of the reasons why declarative debugging cannot feasibly be adapted to traditional programming languages.) In contrast, due to the lack of hard side-effects, a debugger for Ptah has no need to store anything with most goals, because all the necessary information can be reconstructed on the fly. The exceptions to this rule are goals that invoke guards; their execution cannot in general be reconstructed from scratch. The reason of course is that we cannot tell which candidate clause was committed to, unless either we have recorded this information or the algorithm that selects among candidate clauses is deterministic.

Recording information about each commitment is clearly infeasible for the committed-choice languages: there are simply far too many process reductions in a typical run to think about recording a clause-indicator token for each. A variant of this approach is to omit the token if there is only one candidate clause [Nich88], but it may be expensive in CPU time to discover this information. In Ptah, the problem largely disappears, because single-candidate-clause type computations can be described with sequential ANDs rather than guards, and the execution of expressions involving sequential ANDs can be reconstructed without any trouble. The debugger can afford to record a few bytes of information per guard expression if they are the exception and not the rule. Unfortunately, guards occur with a high dynamic frequency in some classes of programs (such as operating systems); however, these classes of programs tend to be difficult to debug in any language.

The other alternative, a deterministic candidate selection algorithm, is undesirable in general, as it may lead to delays that are needed only to ensure determinacy. If our rule is to select the first true guard, and the second guard is known to be true but the first is delayed, then we have to wait until the variables that the first guard is waiting on have been instantiated. However, it *is* feasible to use deterministic clause selection during the test phase of program development. A compiler switch (similar to the -g option of Unix compilers) that engages deterministic candidate selection would help the user to reproduce error symptoms and hence to find their causes. Still, this switch by definition prevents the user from even observing the effects of some bugs related to candidate selection, and cannot help in tracking them down; this is unfortunate if the normal mode of execution uses a nondeterministic candidate selection algorithm. One should therefore always turn off this option for further testing after all other errors appear to have been found.

3.2. Implications of the type system

Prolog, as a language, doesn't know about types. Recently, however, type systems for Prolog have become an active research area, with a flood of publications [Zobe87, Dart88, Nais88c, Yard87, Gogu84, Voda84, Voda88, Xu88, Redd88, Bruy82, Bruy88, Kana85, Mish84, Mycr84, Fruh88, Debr88a, Nais87, Nils83, Diet88, Hank87]. Most of these papers advocate that types be viewed as we view them: as rational AND/OR trees, with the OR-nodes representing types and the AND-nodes representing functors. Others advocate more general systems, systems in which types are described by arbitrary predicates [Nais87], on the ground that rational trees cannot describe all the sets of values that the programmer might want to collect together into a type. However, types are useful precisely *because* they are not arbitrary sets; if they were, we couldn't reason about them conveniently enough. We would like to guarantee that type checking can be done tolerably efficiently; whether a value belongs to a type defined by an arbitrary procedure is of course undecidable. Rational trees seems to be a good compromise: they are expressive enough in most cases yet their type-checking algorithm is of low complexity.

Most of the papers in the above list discuss weak type systems, i.e. systems in which type information is advisory and systems that are administered by separate tools, not by the compiler. Given that most of these type systems are intended for use with Prolog, a language whose syntax has no natural place for declarations, this is perhaps inevitable. We however are not burdened by backward compatibility considerations, and we are free to choose whatever type system we like. As we related in chapter 1, we prefer strong type systems because two decades of software engineering research have shown us that strong types provide redundancy that helps in the production of reliable programs. True, most of this research was based on conventional languages, but experience with logic programming languages seems to confirm the above conclusion. For example, in more than one version of Prolog, the system predicate *findall* had a bug that could have been easily detected by a strong type system. However, most Prolog implementations have no type checkers at all, so this bug survived in many systems (and in some books) even after it was found and corrected in others. When discussing the lessons to be drawn from this, Richard O'Keefe says in [Keef87]:

> How can we avoid the bug? By systematic design in the first place. There are two conditions we want to represent on the stack:
>
> - there is a marker here
> - there is a solution X here
>
> How do you do that? Always, Always, Always, the right way to represent cases in Prolog is to have a different constructor function for each. (You may have to deal with "anything else" cases in the input of your programs, but you shouldn't design your own data structures that way.) These two cases map DIRECTLY onto
>
> - marker
> - solution(X)
>
> What is really important here is this systematic approach to designing data structures. If you want to represent situations
>
> s1 with data d11,...,d1k1
> ...
> sn with data dn1,...,dnkn
>
> use terms
>
> s1(D11,...,D1k1)
> ...
> sn(Dn1,...,Dnkn)
>
> to do it, so that you can tell later on what you meant by looking at the principal functor. If there is a most common case, it is very tempting to ex-Lisp programmers to make that the "else" case of an if-then-else (or a 'caseq') and avoid the "inefficiency" of putting a wrapper around that case. DON'T DO IT. It will actually make your programs LESS efficient (because there will be a 'variable' case in your predicates, which will need cuts in all the preceding clauses, which take time to execute, and you'll probably get temporary choice-points you can do without).

The approach recommended by O'Keefe imposes discipline on the programmer. Such discipline can be useful by itself, but it becomes even more useful if the compiler enforces it, as it ensures that deviations from the discipline are caught immediately. These deviations can be very difficult to find without the help of a strict type checker. They may be caused by typing mistakes, by forgetting to modify some modules after changing a data structure, or by other similar careless errors. Some programmers work slowly and make careless mistakes only infrequently; others work faster and make more careless mistakes as a result. The two populations have been named "humble" and "assertive" programmers by [Wein71]. We ourselves belong to the second category; hence we believe strong types (and strong modes) to be even more necessary. However, we believe that "humble" programmers need these tools too. For example, when a data structure changes, all code in the program that refers to the data structure will have to be inspected and perhaps changed. A visual inspection of the code is virtually guaranteed to miss some references to the data structure, especially in large programs. The compiler, on the other hand, should have no trouble in tracking down all the type errors that result from the change in the type of the data structure.

To assist the compiler in its task, and to reduce the potential for confusing human readers, every function symbol ought to be used in exactly one way in exactly one type. The reasons are laid out by [Keef88]:

> What you do is invent suitable symbolic names for the situations (if you must use an abbreviation, *always* abbreviate that word that way throughout your whole program). Then each situation is represented by a compound term with the situation name as the function symbol and with the associated data items as arguments.

> If some of the situations have different numbers of arguments, you can give them the same name. But this is not a good idea, because if you ever add data items to one of the situations you may make two distinct situations conflict. I have failed to follow this rule in the past. When I used to design trees, I used to think it was clever to use the same function symbol to identify empty nodes, leaves, and internal nodes. I was wrong.

This does not quite mean that we cannot reuse the same name in two different types; it merely means that we cannot reuse them *by themselves*. To have one's cake and eat it, one need only follow the following convention: use the same functor name for the same concept in different types, but to add a prefix or suffix to the name to indicate what type the functor belongs to. For example, one can use the functors *s_empty* and *t_empty* to denote the empty set and the empty tree respectively. This convention yields a good compromise between compiler checkability and human readability; it has been used for years by C programmers to manage *struct* and *union* field names.

One further advantage of a compiler-administered type system is that it can lead to increased execution speed (as alluded to in the last paragraph of O'Keefe's first quote). User defined types tend to have few alternatives, i.e. each OR-node has only a small number of child AND-nodes; we can therefore implement functors as small, consecutive integers. This speeds up indexing, as we can do a simple indirect jump through a table, whereas conventional systems need to use a hash table to convert a functor name into an address.

3.3. Implications of the mode system

We now turn our attention to the implications of the mode system itself. As we discussed in the introduction, our mode system does require some discipline from the programmer, and it does lead to a certain loss of flexibility. 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 most good Prolog programs *are* well-moded even if their modes are not written down, and many others can be made well-moded with minor transformations. Of the rest, a large fraction are traps just waiting to spring on some unsuspecting programmer. To substantiate this claim, we refer to [Drab87]. That paper describes an experiment in which a set of existing Prolog programs was tested to see whether they were *simple*. By Drabent's definition, an annotated logic program is simple if every variable in every clause has a defining occurrence and if every non-defining occurrence is preceded by a defining occurrence. The annotations are primitive versions of mode assignments, and simple programs are a restrictive subclass of mode-correct programs.

Drabent tested 23 existing Prolog programs and found 13 of them to be simple, and a further 7 to be non-simple because of one clause alone. He analyzed the causes of non-simpleness in both these 7 and the remaining 3 programs, and came up with a list of reasons. Many of his reasons reflect the restrictive nature of the definition of simple programs: they describe non-simple programs that are nevertheless well-moded. This is the case with the reasons Drabent calls "always failing clause", "insignificant value", "multidirectional use of a procedure", and "glued arguments". This is probably also the case for another reason, "variable as a tag", although Drabent does not give enough detail for us to be sure. Yet another reason, "variables as unique objects" can be simply eliminated by replacing unique variables by unique integers (or in Prolog, by unique atoms; Drabent himself suggests this approach). Integers and atoms have the advantage that they allow uniqueness to be tested without recourse to non-logical predicates such as *var*.

Drabent's next reason, "goal as an argument" is concerned with the direct evaluation of terms as goals. Such a view cannot be usefully adopted in a strongly moded environment. We believe that the proper framework for the dynamic construction of code and for metaprogramming is the context of higher-order functions/procedures. In this view, the compiler is merely a procedure-producing procedure, and programs invoke it just as 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 too onerous a restriction: von Neumann languages have lived with it for years.

Although meta-circular interpreters cannot be well-typed or well-moded because they have to deal with procedures of arbitrary types and modes, they do not really need to be well-typed and well-moded by themselves. Meta-interpreters are widely acknowledged to be too inefficient for production use, so systems that rely on them significantly usually evaluate them partially with respect to the program they want to run. In the partially evaluated meta-interpreter each part of the meta-interpreter appears many times, each copy being specialised to a procedure or clause of the input program. If this input program and the meta-interpreter's use of its own data are both well-typed and well-moded, then the partially evaluated meta-interpreter should be well-typed and well-moded as well. (We discuss higher-order types, the compilation process and meta-programming in chapter 4.)

The next reason, "natural use of variables", is related to the previous one: it is concerned with the use of Prolog variables in system programs to represent Prolog variables in user programs. This simply does not arise in Ptah, since we handle meta-programming differently: variables in the input of the Ptah compiler are represented by text, not by Ptah variables.

Two reasons, "variables as pointers" and "open data structures" are related to the issue of destructive update. Programs that are non-simple for either of these reasons should be well-moded if they have a regular enough flow of data and would be non-well-moded if they don't. A case in point is the paradigm of difference lists [Clar77]. The standard definition of *dappend* (*append* for difference lists) is the unit clause $dappend(A - B, B - C, A - C)$. When *dappend* succeeds in Prolog, the skeletons of the right halves of the arguments will in general contain a number of *cons* functors and a single variable at the end of the chain. Since the positions of these variables cannot be known at compile time (it depends on the length of the lists), the usage of *dappend* prevalent in Prolog cannot be well-moded.

The basic explanation for this is that a difference list is not an object in the Herbrand universe. *A.B - B* is not a term; it represents a set of terms, whose members should be (but aren't) treated identically. In Prolog one cannot define a predicate *dlength* to find out the length of a difference list without either using non-logical "predicates" such as == or instantiating the tail variable, thus rendering the difference list useless for further appending. The basic problem is that *dappend* is incomplete: after appending $1.X - X$ and $2.Y - Y$, one can no longer concatenate $1.X - X$ and $3.Z - Z$, because X has been bound to $2.Y$, a term that cannot be unified with $3.Z$. *dappend* is therefore *incomplete*: sometimes it fails when we would wish it to succeed. However, in Prolog any attempt to correct this incompleteness would also eliminate the performance advantages of *dappend*. The reason for this is that the efficiency of *dappend* depends on a regular pattern of use: a difference list must be appended to only once because its tail variable can be instantiated only once.

Given these facts, disciplined programmers must of course make sure that each difference list is appended to only once; if they wish to avoid non-logical predicates they will convert difference lists to normal lists before using their values (for purposes other than appending) by supplying other normal lists (e.g. *nil*) as their tails. These two conventions are exactly what is required to make *dappend* well moded. Given the type declaration

```
type dlistof[T]: minus(front: listof[T], tail: listof[T]);
```

and the use of an infix minus sign to denote the functor *minus*, we can define a well-behaving *dappend*; it definition appears in example 3.1.

The mode declaration of *dappend* states that the caller must supply values for A, B and C through the front of the first and the second arguments and the tail of the third argument respectively, corresponding to the forward mode of *append*. *dappend* can also get its values for A, B and C from other parts of its arguments to implement some, though not all, of ordinary *append*. The appearance of $L1$, $L2$ and $L3$ in the output list indicates that *dappend* can generate the main functors of all its arguments, though this functor would normally appear textually in the caller anyway.

The upshot of all this is that although we do not allow the arbitrary use of difference lists, we do allow their *disciplined* use. The Parlog version of quicksort in example 3.2 (taken from [Clar86]) is well-moded both by Parlog's standards and by ours. It is well-moded because there is an orderly flow of data from *remainder* to *frontsortedlist* through *remainderplus*. In our syntax, and using explicit calls to *dappend*, this code would look like example 3.3. The code of that example is well moded by our standards; the flow of data is the same as in the previous version except for a few trips through *dappend*. However, it is *not* well moded by Parlog's standards, because Parlog's mode declarations are attached to arguments instead of type tree nodes, and thus they are adversely affected by transformations that group together the arguments of a procedure. We call this problem the *argument composition paradox*; it is shared by other

```
procedure dappend(L1, L2, L3: dlistof[T])
mode
        => L1, L2, L3
        # L1.front => L3.front
        # L2.front => L1.tail
        # L3.tail  => L2.tail;
def
        L1 = A - B, L2 = B - C, L3 = A - C
end dappend
```

Example 3.1: procedure dappend.

```
mode sort(list?, sortedlist^).
sort(list, sortedlist) <-
        sortd(list, sortedlist, []).

mode sortd(list?, frontsortedlist^, remainder?).
sortd(head.tail, frontsortedlist, remainder) <-
        partition(head, tail, list1, list2),
        sortd(list1, frontsortedlist, head.remainderplus),
        sortd(list2, remainderplus, remainder).
sortd([], list - list).
```

Example 3.2: quicksort in Parlog.

systems that also attach mode information to arguments (e.g. the Relational Language and MU-Prolog but not NU-Prolog) By contrast, we allow the arguments of procedures to be grouped together at the programmer's pleasure. We do not care whether difference lists are represented by two arguments or by the two substructures of one argument, because type tree nodes are not affected by the grouping transformation.

Drabent's other example of "variables as pointers" is the *serialise* program from the Warren Prolog benchmark, which appears in example 3.4.

The flow of data in this program is somewhat devious, but quite regular. In the intended use of *serialise*, *L* is input and *R* is output. *pairlists* takes *L* and produces a list of variables *R* of equal length, as well as a list of pairs *A*. The first half of each pair is an element of *L*; the other is the corresponding free variable in *R*. The calls *arrange* and *numbered* process the first halves of the

```
procedure sort(In, Out: listof[T])
mode:
        In => Out;
def
        sortd(In, Out - nil)
end sort

procedure sortd(In: listof[T], Out: dlistof[T])
mode
        In, Out.tail => Out, Out.front;
def
        In = nil,
        Out = List - List
or
        In = Head.Tail,
        partition(Head, Tail, List1, List2),
        sortd(List1, Sortlist1),
        dappend(Sortlist1, Head.X - X, Sortlist1prime),
        sortd(List2, Sortlist2),
        dappend(Sortlist1prime, Sortlist2, Out)
end sortd
```

Example 3.3: quicksort in Ptah.

```
serialise(L, R) :-
        pairlists(L, R, A),
        arrange(A, T),
        numbered(T, 1, N).
pairlists(X.L, Y.R, pair(X, Y).A) :-
        pairlists(L, R, A).
pairlists([], [], []).
```

Example 3.4: procedure serialise.

pairs and instantiate the second halves accordingly, and this instantiates the elements of R as well. The modes of the procedure *pairlist*(*Left*, *Right*, *Pairs*) that would most naturally occur to a programmer are:

```
Left, Right => Pairs;
Pairs => Left, Right;
```

With these modes, *serialise* would not have a valid mode assignment. However, the unification of the third argument of *pairlists* with $pair(X, Y).A$ could involve data flows other than the two straightforward ones. Because unification can have one mode for one node of the type tree of its arguments and another mode for another node, and because these unifications can be delayed, more modes are in fact possible. The one that allows us to give *serialise* a valid mode assignment is (specifying all the type forest nodes in a redundant form to increase readability):

```
Left, Left.element => Right, Pairs, Pairs.element, Pairs.p_object
# Pairs.p_number => Right.element;
```

assuming the type definitions and declarations

```
type listof[T]:  nil or cons(element: T, listof[T]);
type counter[T]: pair(p_object: T, p_number: integer);

procedure serialise(Init: listof[T], Final: listof[integer])
procedure pairlists(Left: listof[T], Right: listof[integer],
                    Pairs: listof[counter[T]])
```

The code works because *arrange* and *numbered* can together compute *Pairs*. *p_number* from the other parts of the variable *Pairs*. Although the correct mode declaration of *pairlists* could not be considered straightforward, it is as straightforward as *serialise* allows it to be. Given the mode declaration, programmers can relatively easily verify that it is correct; in its absence, they have to discover it (which is much harder) if they wish to convince themselves that *serialise* would work.

The paradigm of "open data structures" is to use an unbound variable to represent one state and to instantiate this variable to represent a transition to another state, the new value of the variable giving the identity of the new state. Drabent's example is the fast-update binary search tree that was mentioned in [Warr80]: a binary search tree whose leaves are unbound variables. When a new value is inserted, the appropriate leaf variable is bound to a term containing the new value and two other unbound variables.

Although this paradigm frequently leads to fast Prolog programs, we do not believe that its use should be encouraged, for several reasons. One reason is that the applicability of this paradigm is restricted to problems in which each state changes only once: a binding, once established, cannot be broken except by backtracking. One could conceivably represent a state by a list whose tail is always an unbound variable, the last element of the list before this unbound variable giving the current state of the object, but the overhead of scanning down the list would probably wipe out the gains from destructive update. The other, much more important reason is that open data structures encourage the use of non-logical predicates. For example, one cannot test whether an item is in a fast-insert tree without either modifying the tree (e.g. inserting the item) or using *var*. The irony is that such non-logical practices are said by many to exploit the power of the "logical variable".

We would prefer to stay within the logical framework if at all possible. In many cases this is trivially possible: programs that use destructive update only in a regular manner are often mode correct, e.g. procedure *sortd* above. They can be viewed as programs with unconventional (i.e. not strictly left-to-right) flow of data, and the mode declarations encourage the reader to view them so. In cases without any regularity, e.g. fast-insert trees, we would prefer to rely instead on language primitives that encapsulate destructive update in a logical manner, e.g. the multi-version structures of [Cohe84], the mutable arrays of [Erik84], or the hash tables of [Bark87]. These primitives could even use fast-insert search trees as the implementation mechanism. Alternatively, we could use eager garbage collection methods, such as those described in [Chik87] and [Goto88] or their refinements, whose efficiency may in time approach the efficiency of destructive update. And of course there is always a possibility that a high level program transformation can eliminate the need for destructive update by eliminating intermediate data structures. Wadler's work on functional languages seems especially promising in this respect [Wadl84, Wadl88], since strongly moded logic programming languages are closely related to functional programming languages. Our work on stability (see section 4.4) may also help in this area.

The above discussion should have convinced the reader that the loss of expressive power accompanying strong modes is not serious enough to pose a threat to the usefulness of our system; most of the code examined by Drabent turned out to be mode-correct and we have presented ways of handling the rest. We now turn to the other side of the coin: the *benefits* of strong mode systems.

The most important benefit is increased reliability. A typical program in a strongly moded language should be more reliable than (say) its Prolog counterpart, because the compiler has extra sources of redundant information that it can use to find errors. This is confirmed by our own experience, as well as by Drabent's study: the last of his reasons for non-simpleness was "program bugs". Specifically, he found two bugs in what were thought to be debugged programs. Although he does not state which programs had the bugs, one can deduce from the information he does give that at least one of the buggy programs was in regular use.

Another confirmation of our thesis is the following quote; it is from a paper on Flat Concurrent Prolog [Klin88b], but it also applies to all logic languages without strong modes (i.e. all concurrent logic languages except the Relational Language and Ptah):

> A common question a frustrated FCP(?) programmer finds himself asking is: "who the hell has instantiated this variable?".

When two calls try to instantiate the same variable at the same time but to different values, concurrent logic languages have no option but to cause one of the calls to fail. If this call has not yet committed and has other clauses that do not likewise lead to a clash, then everything should be all right; but the alternative is the failure of the entire computation. (In committed choice languages, all the body calls in a program are conjoined, and so the failure of one means the failure of all. Note also that arranging to delay commitment until one is sure that the export of bindings will succeed is a reasonably complex problem in its own right for which good solutions have only recently become available [Burt88, Klin88b].) Hence the effect of small synchronisation errors may not be merely a local deadlock but can be the equivalent of a system crash. This is not conducive to reliability, especially when one considers that these bugs cannot be found except by executing the code in the "right" context, and that in a large program this is usually quite hard to do - companies that sell test coverage analysis tools will vouch for that. This is not to say that our system prevents all bugs - far from it. But *it does prevent a large and important class of bugs, and programmers need all the help they can get on this score.*

Besides helping static debugging tools to do their job, strong modes also help dynamic tools. As we have shown in section 3.1.2, declarative debugging algorithms converge on bugs faster when they are based on knowledge of binding information, and a strong mode system is the best possible source of this information.

Another contribution of strong modes to reliability is their role in the control of complexity. They encourage simple communication patterns between procedures, and they only allow complex communication patterns if they can be described by a regular structure (the type structure). Further, whether these patterns are simple or complex, they must be explicit; this kind of documentation makes programs more readily understandable [Bruy82].

The second benefit of strong mode systems is that they enable us to combine don't-know nondeterminism and stream AND-parallelism. Previously, this combination has been thought to be possible but infeasible, but an implementation can achieve acceptable efficiency if it has access to the information present in a strong mode system (see chapter 5 for further details). In our opinion, the expressive power we gain from our ability to exploit don't-know nondeterminism more than compensates for the expressive power we lose through our inability to handle irregular instantiation patterns. This argument gathers even more force when we consider that most of the other logic languages used for systems programming do not even allow user-defined goals in guards; in many respects they bear a closer resemblance to CSP [Hoar78] than to Prolog, or to logic itself.

The third benefit is the possibility of greatly increased execution speed. One source of this is stream AND-parallelism; this source is of course shared by the other concurrent logic languages. The second source, which is not applicable to these other languages, is intelligent dependency-directed backtracking (the subject of chapter 5). The third source is that strongly moded systems need not worry about the possibility of two calls trying to instantiate the same variable at the same time. The mechanisms necessary to avoid this problem (e.g. atomic tell unification [Sara88]) are quite complex not just from the programmer's but also from the implementor's point of view, and their employment brings inevitable runtime overheads. A fourth source is the possibility of more effective optimisers. Most optimisation techniques are based on detailed knowledge of the possible behaviours of the program to be optimised, and our type- and mode-systems can provide optimisers with high-quality operational information. Program analyzers (especially ones based on abstract interpretation) can also derive much of this information automatically, but by their nature such tools can only provide approximations that the optimiser needs to handle on a worst-case-assumption basis. Languages without strong types and modes thus discourage attempts at extensive optimisation by making it unnecessarily difficult.

We believe that our bargain is a good one. We lose the ability to handle programs with irregular dataflow patterns, but this ability is needed only rarely and we can compensate for the loss using other techniques. On the other hand, we gain reliability, a much expanded area of application, and probably some execution efficiency; these benefits are always useful.

Note that all three of these benefits depend crucially on the rules of the mode system being cast in iron: the mere possibility of a violation drastically lowers the usefulness of a mode system, because it destroys the programmer's confidence in the system and necessitates the re-inclusion of elaborate run-time mechanisms for write access arbitration. Given this notion as a premise, our mode system is almost forced as a conclusion, because as long as the scope of modes is bigger than a single functor, programs using the paradigm of incomplete messages will find it *necessary* to violate the mode system.

3.4. Implications for program length

One undisputable consequence of our requirement for explicit type and mode declarations is an increase in program size. We do not deny this. We believe that the extra information in the program text makes the program more readable; however, opinions on this subject are divided, as the following two quotes demonstrate. The first is from [Bruy82]:

> Usually, people do not take the time to write this information [types and modes] as comments in their programs. This [practice] makes programs rather unreadable for everyone except their author.

The second is from [Azzo88]:

> We think that [the] explicit approach makes the program unreadable for every one except their authors. The declarations cause an explosion of the program size.

We are inclined to agree with Bruynooghe, although we intend type and mode information to be more than just comments. Experience has shown that hard-to-read programs are most frequently written in languages that encourage people to write short programs. One such language, APL, has even been called a "write-only language". In the absence of declarations, readers have to expend mental energy, perhaps much mental energy, to deduce information that declarations would have handed to them on a platter.

A possible source of misunderstandings in arguments on this point is that they tend to be conducted from different viewpoints. As Brooks says in "The mythical man-month" [Jr75] a *program* and a *programming product* are two very different things; the latter has to be well-documented, well-tested, easily maintainable, easily adaptable, robust etc. Accordingly, a programming product costs three times as much as a program that performs the same function but does not need to have all these extra attributes. One can look at type and mode declarations as part of this extra cost; as we have shown before, they do help programs achieve the properties required of programming products. And because they mandate the clear description of at least some aspects of procedure interfaces, they ought to help in the building of *programming systems products* as well.

The following quote from [Hoar81] reinforces this message:

> I was eventually persuaded of the need to design programming notations so as to maximise the number of errors which cannot be made, or if made, can be reliably detected at compile time. Perhaps this would make the text of programs longer. Never mind ! Wouldn't you be delighted if your Fairy Godmother offered to wave her wand over your programs to remove all its errors and only made the condition that you should write out and key in your whole program three times ? The way to shorten programs is to use procedures, not to omit vital declarative information.

Programs written in logic programming languages of course already use may short procedures; in this context, the way to shorten programs is to use higher-order procedures more extensively.

Although we do not recommend it, one could possibly arrange things so that programmers type in only the code of a program and then use type- and mode inference systems to supply the type and mode declarations. We do not think this would work well enough, because the inference systems tend not to discover all types and modes unless given declarations for at least a few key procedures. If programmers have to think about just which procedures these are, then their job is not much easier than that of other programmers who think about (and type in) the types of the arguments of procedures. (As we said earlier, we let the compiler deduce the types of the local variables from the declarations of the procedures called in the body of the procedure, but this consideration does not apply to that case.)

3.5. Theoretical implications

The most obvious theoretical implication of our mode system is that it justifies the elimination of the occur check in almost all cases. Based on the sequencing done during mode analysis, one can trivially prove that the occur check can be omitted from any program if the compiler can prove that no conjunction in the program has data-level circularity. (A unification $X = f(X)$ needs the occur check only if this unification is the producer of X (our mode analysis algorithm allows this), but if it *is* the producer then the unification itself represents a data-level circularity.) In cases where the compiler suspects a data-level circularity, the occur check can be implemented by program transformation. This transformation would take a circular conjunction like $p(X, Y)$ & $q(Y, X)$, where both calls take their first argument as input and produce the second as output, and turn it into $p(X, Y)$ & $q(Y, X1)$ & $X =_{oc} X1$, where $=_{oc}$ is unification with the occur check. Unfortunately, the usefulness of this transformation is limited by the fact that many circularities involve perpetual processes; in such cases the runtime of the introduced call may grow without bound just as the size of the term X grows without bound. This is usually not acceptable.

Another implication of strong modes is that they allow us to relax the safety condition on computation rules. The standard safety condition, as defined in [Lloy87], requires that negated calls be fully ground before called. We can replace this rule by one which states that negated subexpressions may have no outputs. Negated subexpressions may start execution at any time; if they need the value of some input, they will wait for it. Since expressions may not need all their inputs in order to succeed, the subexpression being negated may succeed before all its declared inputs have been produced. If the negated subexpression has any INFREE inputs that actually have no producer in the calling clause, then our relaxed safety condition avoids floundering (unlike the original strict one). Indeed, mode-correct programs cannot flounder: every variable that is consumed has a generator, and the generator cannot terminate without producing values for all its outputs.

The set of mode-correct programs is a superset of the set of allowed programs. In particular, the concept of well-moded programs allows what seems to be a natural characterisation of evaluable predicates. We believe that most results that apply to allowed programs actually hold also for well-moded programs. Dart has proven some of these results for the class of *connected* programs [Dart87], a class that includes allowed programs and is itself included in the class of mode-correct programs. Mode-correct programs may also have desirable properties (particularly properties concerning completeness) beyond those of allowed and connected programs because of the relaxation of the safety rule.

3.6. Comparisons with related notions

Although the concept of modes would at first sight seem restricted to the domain of logic programming, some related notions have in fact been proposed in the context of conventional programming languages: these notions are concerned with whether variables have been initialised or not. In the paper [Dijk79], Dijkstra distinguishes between initialisation and subsequent (re)assignments; he separates variables that may be updated from constants that may be initialised but may not be reassigned. He also establishes controls over the scope of variables that are reminiscent of parameter passing controls. Global variables and constants (*glovar* and *glocon*) are inherited from the surrounding environment and are already initialised; they correspond to inputs. Virgin variables and constants (*virvar* and *vircon*) are inherited from the surrounding environment in the uninitialised state; the local block is obliged to initialise them, hence they correspond to outputs. Private variables and constants (*privar* and *pricon*) correspond to local variables.

We know of no widely used programming language that mandates such controls at the block structure level. Some languages (e.g. Ada [1%Q83]) do impose some controls at the procedure level, by requiring the programmer to state whether parameters are input, output, or both (corresponding roughly to Dijkstra's *glocon*, *virvar* and *glovar* classifications).

The paper [Stro86] extends this notion to a concept named *typestate*: just as the (static) type of an object determines the set of operations that may ever be performed on it, its typestate at any point in the code determines what subset of that set of operations may be performed on it at that point. As operations are performed on the object, the compiler keeps track of its changing typestate, and prevents the invocation of inappropriate operations. One application of the typestate concept is to keep track of whether a variable has been initialised; while it remains uninitialised, the compiler prevents any attempt to read its value.

The authors of all three of these proposals have indicated in their papers that their goal was increased program reliability; this supports the main line of our argument in section 3.3.

Strongly moded logic programming languages resemble functional programming languages quite closely: a procedure in a given mode is computing a function from its inputs to its outputs. We therefore expect some research from the area of functional languages to be also applicable to our system. However, we believe that strongly moded logic languages have more expressive power than functional languages. Two sources of this extra expressive power is the natural ability to define multi-mode procedures and the ability to handle partially instantiated data structures. The second of these is very important in the area of operating systems: a server that can be accessed by several clients is quite awkward to write in a functional programming language [Hend82], but can be implemented in a straightforward manner in a logic programming language using the paradigm of incomplete messages [Shap83b].

The field of relational databases also has notions related to modes: the notions of functional and multi-valued dependencies. The relationship is quite weak, however; modes and dependencies have completely different uses (control of running programs and static database design respectively).

Closer parallels can be found in deductive databases, in areas dealing with bottom-up computation and magic sets: the notions of adornments and sideways information passing strategies or *sips* [Beer87]. An *adornment* of a predicate with *n* arguments is a string consisting of *n* symbols from the alphabet *{b, f}*, where *b* stands for *bound* and *f* stands for *free*. The *n*th symbol designates whether the *n*th argument is bound or free at the time of a call to the adorned predicate. As the bottom-up computation of an adorned program always computes ground answers (in the approaches we are considering), any arguments that are free at the time of the call will be bound when the call is completely evaluated. Adornments therefore are essentially mode declarations restricted to the function-free case; sips are the similarly restricted versions of mode assignments.

Most of the current research in bottom-up computation and recursive query optimisation is based on adorned programs, because the information present in adornments is essential to the optimisation techniques. The main limitation of the concept of adornments is that adornments are fully authoritative only in the context of function-free databases. The concept *has* been extended to databases with function symbols, by interpreting *b* and *f* as applying to the main functors of arguments only [Rama88], but this leads to loss of information as to where the subterms of those main functors will be bound. This may in turn cause the "optimised" program to waste time performing redundant computations. We believe that our mode system, as a strength-preserving generalisation of adornments, can avoid this problem, and allow the full application of magic sets and related optimisation techniques to logic programs with function symbols. As a concrete example of the advantages of modes, we note that the approach described in [Rama88] requires expensive subsumption checks at each iteration of the bottom-up computation. These checks can be replaced by simple equality tests in a system based on strong modes; a relation that can produce two solutions that are unifiable but not identical (apart from variable renaming) is not well moded and will be rejected by our mode system at compile time.

3.7. Comparisons with notions in logic programming languages

In the first implementations of logic programming languages, the flow of data was determined solely by the order of execution of the calls of the program. Although this is still true of some versions of Prolog, most logic programming languages these days have additional instantiation control systems as well, usually in the form of declarations or annotations. These instantiation control systems have a number of uses. The important ones are:

(1) to prevent bugs;

(2) to document the program;

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

(4) and to gain efficiency.

Curiously, the least important use was exploited first. The mode declarations of DEC-10 Prolog [Warr77] specify for each argument of a predicate whether the programmer expects that argument to be bound or unbound at the time of calls to the predicate, or whether the argument will be bound in some calls and unbound in others. The compiler exploited this information by specialising the code used to perform argument unification. This specialised code was probably larger than a call to the general-purpose unification routine, but it had the potential to execute faster (although in Poplog it did not always do so [Mell85]).

The chief drawback of this scheme is that if a mode declaration is wrong, then the generated code will be wrong as well. As the validity of a DEC-10 Prolog mode declaration depends on the ways in which the procedure is called, and since those ways potentially depend on the entire program, a bug in any part of the program can invalidate any mode declaration. If twenty years of software engineering should have taught us anything, it is the extreme unwisdom of reliance on the infallibility of humans; we cannot really expect programmers to write correct DEC-10 Prolog mode declarations. (They may make mistakes when writing Ptah mode declarations as well, but these errors can never cause incorrect code to be generated.)

The literature contains a number of algorithms for automatic mode inference [Mell81, Debr86b, Debr88a, Bruy88, Debr87, Debr88b, Warr88b, Marr88b, Chan85b]. Some of these go to considerable lengths in an effort to obtain precise binding information, yet only very rarely can they locate all the "interesting" (not don't-know) modes. In particular, they have trouble with the meta-logical and extra-logical features of Prolog (such as *call*, *not*, *arg* and *assert*/*retract*).

The aim of mode inference systems is usually program optimisation. Their usefulness during debugging is limited by two factors. One is that they require as input the calling patterns of the procedures that may be invoked from the top-level loop of the interpreter. During debugging, Prolog programmers will typically want access to all predicates, so supplying this information amounts to declaring the modes of all procedures. Another reason is that not only do mode inference systems increase compile time by factors of 2 or 3 [Warr88b], but they usually rule out incremental compilation as well.

DEC-10 Prolog mode declarations are precise (they apply to only one functor), but they are definitely not strong. Some researchers have taken the opposite track: they investigated stronger but less precise schemes [Bruy82, Demb85, Shoh84, Drab87, Sick79, Dera85]. These systems associate an annotation with each procedure, stating which arguments are input and which are output: the input arguments must be ground on procedure entry and the output arguments will be ground on procedure exit. Many programs can be annotated in this fashion (as witness Drabent's study). Unfortunately, however, the discipline imposed by the annotations also rules out many desirable practices (e.g. the use of partially instantiated data structures), so to the best of our knowledge they have not been incorporated in any Prolog implementation.

The notions of both DEC-10 Prolog modes and annotated logic programs are intimately bound up with the computation rule: the order of goal execution determines what variables will be bound when. This descriptive approach to modes works for AND-sequential languages, but it breaks down in AND-parallel languages, since in those languages there is no single order of execution. Parallel systems thus seem to demand a different, *prescriptive* approach: modes should control the order of events, not the other way around.

The first implementations of prescriptive modes were in Prolog dialects that simulated AND-parallelism by coroutining. IC-Prolog [Clar79] was the earliest such system. It allowed the programmer to provide more than one version of a clause, the versions differing only in annotations and the order of the body literals; it relied on annotations in the heads of these alternatives to choose one of them. These annotations basically specified which parts of the call had to be instantiated and which parts were not allowed to be instantiated, by specifying which variables in the head had to be input and which had to be output. (IC-Prolog and its descendants also used their annotations to control priority and buffering, but we are not concerned with those aspects here.)

The main purpose of the *wait declarations* of MU-Prolog [Nais85c] and the *when declarations* of NU-Prolog [Thom86] is to stop infinite loops caused by insufficiently instantiated calls. They work by delaying the evaluation of calls that the declaration says are not sufficiently instantiated. Because they suspend whole calls, not just primitive unifications, they are subject to what we call the *procedure composition paradox*. Suppose we have two calls, $p1(X)$ and $p2(Y)$, where $p1(X)$ fails and $p2(Y)$ waits forever because the producer of Y is in an infinite loop. A clause containing a conjunction of these two calls would fail. However, if we form a procedure $p1p2(X, Y)$ by a Cartesian product of the clauses of $p1$ and $p2$, then the equivalent call $p1p2(X, Y)$ waits forever. Being based on procedure arguments, *wait* declarations also suffer from the argument composition paradox; *when* declarations do not.

Like DEC-10 Prolog modes, *wait* and *when* declarations can be generated automatically [Nais85b]. They have some value as program documentation, though this value is not as high as we would like, because their use is not compulsory, and because they document *how* the procedure does its job rather than *what* it does. 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; instead it reflects the implementation of *append*. If the order of the arguments of *cons* were reversed, the Ptah modes would still be the same, but the NU-Prolog when declaration would have to read *when* B *or* C. For many well-behaved procedures, the *when* declaration reflects the inputs required by the head unification, and by the built-in procedures called in the body of the procedure; the inputs required by the called non-built-ins (including any recursive calls) are reflected in the *when* declarations of the called procedures.

The triggers of Epilog [Port82, Wise86] state that the execution of a call should be delayed until a certain number of its arguments are instantiated. As such, they are a primitive form of *wait* declaration. Their usefulness is strictly limited, as is their flexibility.

Both wait and when declarations can be used to implement the *freeze* predicate of Prolog II [Boiz86a, Colm82]. Freeze is exclusively a control mechanism: it directs that the evaluation of a call should be delayed until a given variable is instantiated. Its semantics is actually somewhat 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 is not a declaration but occurs in the body, thus making it hard to write multi-mode procedures.

Programs written in Parallel NU-Prolog (also called PNU-Prolog) include mode information as part of their *lazyDet* and *eagerDet* declarations [Nais88a]. This mode information states for each argument whether that argument is input or output, as in the annotated logic program schemes we discussed above. The difference is that in PNU-Prolog the modes do affect program execution, although not directly: they are used to guide a preprocessor that transforms the code of the program. The generated code involves many calls to non-logical predicates; PNU-Prolog provides the lazyDet and eagerDet declarations as syntactic sugar to allow programmers to avoid writing such code. However, they can if they wish to, so the instantiation control system of PNU-Prolog, like that of MU-Prolog, NU-Prolog and Prolog-II, is a weak one.

The main control mechanism of Concurrent Prolog (besides the compulsory guards) is the *?* read-only annotation. The purpose of the annotation is to prevent the instantiation of variables in inappropriate places. The original definition of read-only variables was [Shap83a]:

> The unification of read-only term X? with a term Y is defined as follows. If Y is non-variable then the unification succeeds only if X is non-variable, and X? and Y are recursively unifiable. If Y is a variable then the unification of X? and Y succeeds, and the result is a read-only variable. The symmetric algorithm applies to X and Y?.

This definition has a number of flaws [Levy84, Sara86, Rama86]. The most important are that it causes unification to become order-dependent and that variables can acquire read-only annotations at run-time even if they have no such annotations in the program text, confusing both human programmers and program analysis tools. The team at Weizmann has developed several other revisions of Concurrent Prolog (most of them restricted to *flat* clauses, whose guards may call only built-in procedures) such as FCP(|, $\downarrow$), FCP(:) and FCP(:, ?) to solve these and other problems that have cropped up over the years, e.g. those related to the atomicity of commitment. Nevertheless, they have kept the read-only annotation as the basic synchronisation mechanism, despite their admission that it leads to subtle bugs (see the quote from [Klin88b] in section 3.3) and that its use often leads to unclear code, besides being hard to implement efficiently, as shown by another quote from [Klin88b]:

> ... it is clear that in most cases static specification of input, as is done in all other languages except FCP(?), is preferred over dynamic specification. The former is clearer, more modular, and easier to implement efficiently.

The apparent motivation of the Weizmann group in doing this is that they wish to preserve FCP(?)'s ability to exploit paradigms such as test-and-set, duplex streams, multiple writer streams, and short-circuit termination detection [Klin88b]. We believe that there are better ways of doing each of these things, and that the programs using these ways are in fact well-moded; one should be committed to *ends*, not *means*. Test-and-set and other mutual exclusion mechanisms should be implemented via a call to an arbiter (this view is also advocated in [Burt88]). Multiple writer streams should simply be single-writer streams merged together. The implementation of the merge operator may use destructive operations to achieve good efficiency as long as these are hidden behind a logical interface; this was proposed by researchers at Weizmann itself [Shap85]. Duplex streams should be implemented as two streams, one for the data and one for the consumer's feedback; the use of one variable for two purposes has long been known to be a false economy in the world of conventional languages. If efficiency really is very important, then one should add an extra argument to the messages of the stream and let the consumer instantiate it when it has finished processing the other parts of the message. As for the short-circuit technique, it can be implemented as well-moded code, but it should simply not be needed in Ptah programs; in the absence of data-level circularities the distributed termination problem just disappears.

Saraswat's ↓ input-only annotation [Sara86] and Ramakrishnan and Silberschatz's % annotation [Rama86] were both devised to avoid the problems of the read-only annotation, and to a large extent they are successful in that task. Ramakrishnan and Silberschatz's system even has some compile time checks (these enforce the syntactic restrictions that are needed to prevent the instantiation of ostensibly non-writeable variables). However, neither of the proposed languages can support *protected embedded channels*, i.e. if they allow a call to instantiate the top level of a data structure then they cannot guarantee that the call will not instantiate any of the lower levels. This makes it hard to write code for the client side of client-server systems.

In GHC [Ueda86b] and in Flat GHC, the role of the instantiation control system is played by the *rule of synchronisation*. This rule states that procedures are not allowed to instantiate variables appearing in the call while they are executing within their guards. This rule is purely a control mechanism; one can understand it only operationally, not declaratively.

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 (and those of the Relational Language) are almost identical. Unfortunately, Parlog's *weak input arguments* (a mechanism that 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. 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 multi-mode procedures are hard to write in a language with a compulsory committed-choice mechanism.

A 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.
```

This unpredictability, which Parlog shares with GHC, the various FCP variants, with ↓- and with %-annotated languages, is a bountiful source of bugs (remember the quote from [Klin88b] in section 3.3). The designers of Parlog, however, attempted to minimise the risks by embedding weak input arguments in an otherwise strongly moded language (unlike the designers of these other languages, who made no such effort). Our approach eliminates the risk yet retains sufficient flexibility.

Summary. We now go through the four objectives of instantiation control systems that we listed at the beginning of this section. We evaluate how well the mechanisms we discussed above support each of the objectives.

To prevent bugs. The only languages that share our primary concern with the prevention of program errors are the Relational Language and (to a lesser extent) Parlog. Some of the other mechanisms have error prevention as a secondary aim: e.g. the compiler checks in Ramakrishnan and Silberschatz's proposed system and the infinite-loop preventing preprocessors of MU- and NU-Prolog. With the exception of the Relational Language, none of these mechanisms can guarantee programs to be free of problems caused by write-write conflicts.

To document the program. It is striking that none of the systems we discussed, with the exception of the Relational Language and Parlog, really try to use their mechanisms to help document the program. Even for Parlog, many published programs (e.g. those in [Clar86]) separate mode declarations from the procedures to which they refer, thus somewhat reducing their documentation value. The value is limited in any case because the code can override the mode declarations; this is a necessary consequence of the modes' lack of precision. We must also mention that both [Bruy82] and [1%Q86] advocate that programmers should document their procedures with mode declarations, contained in comments in the Quintus case. However, neither proposal envisages compiler enforcement of this convention, although Bruynooghe proposes a separate verifier to check user-supplied declarations.

To help control the execution of the program. This objective is the primary one for most of the instantiation control mechanisms, and a secondary one for most of the others. Some of these mechanisms control procedures: triggers, the annotations of IC-Prolog, the wait- and when-declarations of MU- and NU-Prolog, the ↓- and %-annotations of the CP-derived systems, and the mode declarations of the Relational Language and Parlog; others control calls, an approach that is much more error prone but permits somewhat more flexibility: the freeze predicate and the ?-annotations of FCP.

To gain efficiency. This was the only objective of Edinburgh Prolog. It was probably not the primary factor in the design of the other mechanisms, though it was always an important constraining factor; for example Parlog's modes allow a compile-time guard safety check in place of the expensive run-time mechanisms of GHC.

Our mode system, like the Relational Language's and Parlog's, addresses all four objectives. Our modes are a much sharper tool for the control of program execution: the Relational Language disallowed many useful constructs and Parlog programs have to breach the mode system to accommodate them. Because of the necessity for these breaches, our approach is superior to Parlog's in the error-prevention department as well; and because of its precise *and accurate* description of the possible behaviours of procedures we believe it to provide better documentation as well.

Chapter 4

Operating system issues

We start our discussion of operating system issues by describing how we handle code generated at run-time. Next we cover the mechanism we use to surmount the problems that the operating system environment poses for strongly typed and moded systems. We then tie these two threads together in our presentation of the compilation process. The main body of the chapter presents a solution of the conflict between logic programs' tendency to generate multiple solutions and real world devices' requirement that commands being sent to them should be unambiguous.

4.1. Higher-order types

As we have discussed in the background section (section 1.2), we do not allow running programs to be modified, but we do allow the dynamic construction of procedures and the invocation of such procedures. To use the new procedure, the program must know the types of its arguments and the modes in which it may be used, and declare a higher-order (i.e. procedure-valued) variable of that type and mode.

As an example of the definition of a higher-order type, we present the type *unixprogram*. It leaves out some things for the sake of brevity; a real Unix program would also need to be given an argument vector (including options), an environment vector, a standard error stream, and possibly other arguments as well (e.g. a stream of signals).

```
type unixprogram = procedure(
        In0, In9: listof[character],    % input as d-list
        Out0, Out9: listof[character], % output as d-list
        Os0, Os9: listof[syscall])      % OS stream as d-list
mode
        In0, Out9, Os0.kerneldata1..n, Os9.userdata1..m =>
        In9, Out0, Os0.userdata1..m, Os9.kerneldata1..n [stable];
end unixprogram
```

Example 4.1: type unixprogram.

This states that variables of type *unixprogram* have procedures as their values. The arguments of these procedures are input, output, and operating system streams, all represented as difference lists. The mode declaration may look complex, but with practice one can recognise it as a standard paradigm, as one recognises the C code $for(i = 0; i < N; i++)$ as a standard paradigm. The pattern *In*0 => *In*9 shows that Unix programs *consume* the initial portion of their input d-list; the reverse pattern *Out*9 => *Out*0 shows that processes *produce* the initial portion of their output d-list. These two patterns are both special cases of a more general pattern that applies to difference lists used for the transmission of incomplete messages, a pattern that manifests itself directly in the parts of the mode declaration involving *Os*0 and *Os*9. This pattern says that the declaration of the left half of a difference list is the declaration that one would give if the list were represented as normal (non-difference) list, and that the declaration of the right half is the same except for the transposition of input and output. To see the intuition behind this transposition, consider the conjunction of *P* and *shell* in the body of the first alternative of the extract from the shell process that appears in the next section. The variables $In2$, $Out1$ and $Os2$ both appear once as the left and once as the right side of a difference list. Since each part of each of these variables must have exactly one producer, parts that are input in one must be output in the other.

A final note on the mode declaration above. The interface between processes and the operating system is defined by the structure of the type *syscall*. The root OR-node of this type has one AND-node child for every type of system call that processes can make. The arguments of these AND-nodes in turn are the parameters of the system call concerned. As these parameters may themselves have complex types, the full definition of type *syscall* would take up a considerable amount of space; accordingly we omit it. Some of the OR-nodes in this type are intended to be instantiated by user processes, while some are intended to be instantiated by the kernel. In the mode declaration, we represent OR-nodes of the first kind with $Os0.\,userdata1\,..\,m$ and those of the second kind with $Os0.\,kerneldata1\,..\,n$. We will discuss the *[stable]* annotation in section 4.4.

The names of procedure-valued variables may appear in expressions wherever a procedure name may. Calls to procedures represented by variables are of course delayed until the procedure-valued variable gets its value. Because such execution may begin as soon as the variable is instantiated, such instantiation must be delayed until the value of the variable has been completely defined. (We can use the atomic unification technique we discussed in section 1.2 to ensure this.) If the definition of the type of this variable says that the procedure represented by the variable has only one mode, then this value is a pointer to the start of the object code of the procedure. If the procedure has two or more modes, then it is a pointer to an array of pointers to object code, one for each mode of the procedure. In this latter case, the code emitted by the compiler for the call will access the proper version of the procedure by branching through the array element that corresponds to the mode assigned to the call, as determined by the usual mode analysis.

Procedure-valued variables may be passed as (parts of) arguments in procedure calls just like other variables; in such cases their type must of course be declared in the called procedure as must the types of all arguments. However, we also require programmers to declare the types of procedure-valued local variables (something we do not do for ordinary variables) if the body of the procedure in question contains a call to the procedure-valued variable. Without such declarations, the compiler's type inference algorithm cannot reliably find out the types of variables that appear only in calls to procedure-valued variables. (It can do so only if the procedure-valued variable, besides being called, is also passed as (part of) an argument to a procedure of known type. In that case the type of that procedure gives us the types of the arguments, including the argument corresponding to the procedure-valued variable, and the type of this argument gives us the types of the arguments of the procedure-valued variable.) In any case, the mode analysis algorithm needs to know the modes in which the procedure-valued variable can be invoked, and the stability analysis algorithm (section 4.6) needs to know the stability annotations on those modes.

Programs may generate values for a higher-order variable P in any one of three ways. The first is to assign to P the value of another higher-order variable Q by executing the unification $P = Q$. Such a unification is non-logical when performed in test mode, as we only compare the addresses of the two procedures (we do not support higher-order unification). The second is to make P refer to an already existing procedure p by executing the unification $P = p$. The third way is to invoke the compiler. But before we discuss this method, we need to cover some other ground.

4.2. The universal type

In a logic operating system, the most straightforward way to view the file system is as a perpetual process providing persistent storage of named objects. To allow us to keep objects of more than one type in one file system, while still guaranteeing complete type safety, we need a way to convert objects to a common type; to keep type safety, we need be able to convert them back again to their original type *and to no other type*. To be able to perform this check, we obviously need to keep the original type of the object with the object in its converted form. We call the type of objects in this converted form the *universal type*, which we write as *univ*. The name comes from the fact that objects of all types, including *univ*, can be converted to type *univ* by bundling them up with their types in what we call a *cocoon*. As far as we know, this idea was first put forward by Cardelli, whose name for the concept is the *dynamic* type [Card85].

Three primitive operations are sufficient to integrate *univ* objects into the rest of our system (Cardelli lists more, but these can be defined in terms of the three primitives we give). One can convert any object to the universal type through the call *make_cocoon*(*Orig_obj*, *Univ_obj*), and back again through the call *break_cocoon*(*Univ_obj*, *Recovered_obj*), both of which take their first argument as input and produce their second argument as output. The operation *break_cocoon* succeeds only if the type of *Recovered_obj* agrees with the type of *Orig_obj*. This type is stored in the cocoon along with the value of *Orig_obj*; the check is done by invoking the type-checking part of the compiler. The cocoon thus protects the objects it encapsulates against arbitrary manipulation. A third operation, *probe_cocoon*(*Univ_obj*, *Type_desc*) takes a universal object and returns a description of the type of the object stored in the cocoon.

Except for assignment unification, no operations other than the three primitives may be applied to universal objects. To be able to manipulate an object in any other meaningful way, a code that does the manipulation must know its type. (Even in as typeless a system as Smalltalk one must know what kind of messages one can send to an object if one wishes to avoid a core dump or similar calamity.) This does *not* mean that one cannot write programs that can manipulate arbitrary type objects. A program may use *probe_cocoon* to find out the type of the cocooned object, and use this information to construct, compile and invoke procedures that can break the cocoon and manipulate its contents. The only constraint is that the program must know in advance the types of the these procedures; these types will presumably state that some of the arguments are of type *univ*. The structure of the type of the second argument of *probe_cocoon*, type *type_desc*, is derived directly from the grammar rules that describe the structure of types in the programming language (in our case Ptah). Note that type *type_desc* differs from the type *type* that is present in some languages such as Russell [Dona85] and EL1 [Wegb74] (where it is called *mode* in the Algol 68 tradition) in that declarations cannot give as the type of a variable the value of another variable of type *type_desc*.

The following code fragment illustrates the way a shell would retrieve a program from the file system and then execute it (the operating system would proceed similarly when it wants to execute a named shell). This code assumes that *P* and the shell itself both have type *unixprogram*, and that the shell's arguments in its procedure header are *shell*(*In*0, *In*9, *Out*0, *Out*9, *Os*0, *Os*9). We also assume that the shell has already read in a program name, consuming part of *In*0 and leaving *In*1 in its place. Again we stress that this is a sketch only: in real life the user program would have command line arguments, the *fs_lookup* system call may not find the named object, the shell would have some internal state, and so on.

```
Os0 = fs_lookup(Name, Obj).Os1,
break_cocoon(Obj, P),
P(In1, In2, Out0, Out1, Os1, Os2),
shell(In2, In9, Out1, Out9, Os2, Os9)
```

Example 4.2: executing a user program.

4.3. The compilation process

In our system the compilation process is based on four types of objects: modules as text, modules as parse trees, object modules, and procedural objects. They are transformed into each other by the parser, compiler, linker and binder, according to the scheme presented in figure 4.1. A module starts out life as a list of characters (type *text*). The parser converts this into a parse/syntax tree (type *src_module*), which the compiler transforms into object code (type *obj_module*). The linker resolves references between multiple object modules, taking several of them as input and producing one as output. The binder extracts procedural objects from fully-resolved modules.

We have separated the parser from the rest of the compiler with the intention of making parse trees a standard form of program representation. As in Lisp, we intend meta-level programs to use this representation [Sand78]. We also intend that some tools, e.g. syntax-directed editors, should produce and store programs in this form. This cannot be done in most conventional systems, since they cannot store structured data, nor pass it between processes. The few that can do so, e.g. Multics, Apollo Domain and SunOS 4.0, rely, as we do, on the concept of uniform virtual memory [Bens72].

One class of tools that can particularly benefit from the use of parse-tree representations is the class of program transformation systems. Besides their usual uses, in our system these tools also give us the functionality that in other systems would be provided by meta-interpreters. (Our system cannot support the vanilla meta-interpreter [Ster88] or its variations because they cannot be well-typed and well-moded statically.) We believe that given a meta-interpreter, one can automatically derive from it an equivalent meta-compiler, i.e. a program that takes a user program as input and produces another program that does whatever the original program would have done when evaluated by the meta-interpreter. In fact, the output of the meta-compiler should be identical with the output of a partial evaluator that is applied to the same meta-interpreter and user

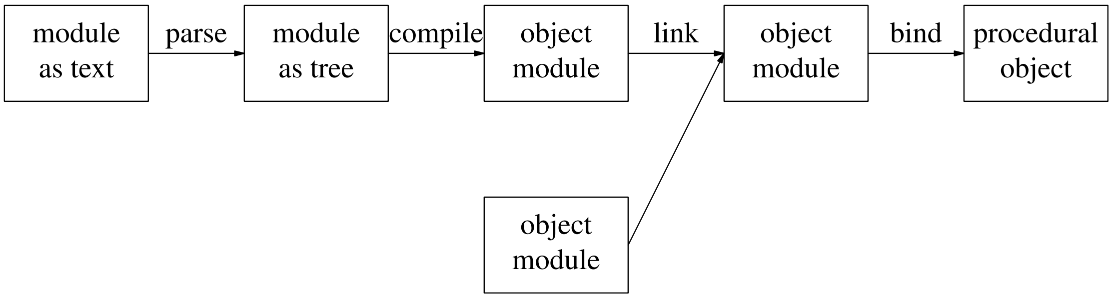

Figure 4.1: the compilation process.

program. This is good, since without partial evaluation the extensive use of meta-interpreters (including the use of composite meta-interpreters) is not really viable [Safr86].

When used to enhance the functionality of programs, meta-compilation (and the partial evaluation of meta-interpretation) usually adds arguments to procedures and goals to clauses, and may add the definitions of completely new procedures [Ster88]. If the resulting program is to make sense, the new parts of the program must be well typed; in our system the original program should also be well-typed. Barring conflicts between the names of old and new variables, functors and procedures, which can be avoided by renaming, the output of the meta-compiler should also be well-typed. The same argument also applies to modes.

Our equivalent of the composition of meta-interpreters is the application of one program transformer to the output of another. One can obtain the same effect in traditional systems by applying a meta-interpreter to object programs that have already been partially evaluated together with another meta-interpreter. To us this process appears to be simpler than the usual composition of meta-interpreters via clause fusion and similar methods [Ster88], even allowing for the fact that we need to keep track of arguments, calls and procedures introduced by earlier meta-compilers so that later applications of meta-compilers can avoid instrumenting them.

In general, we believe with [Safr86] that it is better to enhance the functionality of a system through program transformation than through the run-time system (we rely on meta-compilers while Safra advocates partially evaluated meta-interpreters, but this is a relatively minor issue). Some of the things we can use meta-compilers for are the implementation of declarative debuggers and the introduction of memo functions [Mich68], goal caches [Fagi86] or extension tables [Diet87] into programs (the three terms mean much the same thing). Based on experience with conventional systems, we may also expect to use this technology for the collection of statistics on runtime events (see e.g. the *pixie* program from MIPS) and for implementation of profilers [Wein84].

The independent status of the parser has two further good consequences. First, in our system the parser can be used not only by the compiler but also by programming environments, and by other programs such as cross-reference generators [Wegb71, Sand78]. Second, users can now customise the language by extending or even completely replacing the parser. New parsers can extend the language by mapping new constructs into old ones. Some of the constructs that can implemented this way are macros [Chik84, Kond88] and object-oriented constructs such as classes [Chik84, Clar87a].

The job of the compiler proper is to check the static semantics of the program, including type-, mode- and stability analysis, and to generate executable code. The job of the linker is to combine object modules, resolving any references between them. We require modules to have explicit lists of the procedures they export and import, including the types of these procedures (remember that the type of a procedure includes the types of its arguments, its modes, and its stability annotations). To allow us to do this the export and import lists must also contain the necessary type definitions. These lists should be generated by the compiler according to the direction of the programmer. The programmer should specify which modules to import external types and procedures from; the compiler should fetch the full interface definitions of those types and procedures from the indicated modules.

The linker takes two or more object modules and generates a new one that contains all the type and procedure definitions in the original modules and exports all the definitions that are exported in any of the original modules. The main job of the linker is to see whether procedures that are imported into one of the original modules are exported in another. If yes, then the linker should check that the types of the procedures are the same in the import and export lists and resolve all references to that procedure (a type mismatch should cause the linking to fail). Note that the compiler will also check that the types of imported procedures are compatible with their uses in the module being compiled; when compiling more than one module at a time, some of these redundant checks should be optimised away.

The import list of the output of the linker is the union of the import lists of the original modules minus the set of resolved references. An object module whose import list is empty represents fully resolved executable code. From such a module, one can extract procedural objects using the *bind* operation, which takes as inputs an object module and a name and generates as output a procedural object, providing that the object module has an empty import list, that it has a procedure whose name is given by the argument and that the type of the local variable that is to receive the output is identical to the type of the named procedure in the given module (this last check is identical to the one used by *break_cocoon*).

So, to add new code to a running program in our system, we go through the following steps. First, we create the new procedure either in text form or as a term representing a parse tree, and wrap a module declaration around it. We give this to the compiler; if the compiler finds no errors, it will give us back an object module. If this module has unresolved references, we need to give it to the linker together with modules that supply the missing procedures. If there are still no errors, we can then try to extract a procedural object from the new module and assign it to a higher-order variable; if we have anticipated the type of this object correctly, the *bind* operation will succeed. Since this sequence is somewhat tedious to write down all the time, one should supply some syntactic sugar in the form of a procedure that takes (say) a parse tree and returns a procedural object, and which goes through the necessary steps internally. (Note that this is almost exactly what completely compiled Prolog systems do. Some Prolog systems cannot compile dynamic code and must interpret it instead; this exposes the programmer to the perpetual danger of the compiler and the interpreter producing different results.)

To allow new procedures access to all the procedures in the running program, the running program must have access to the module from which it itself was extracted. The type *unixprogram* should therefore be extended with an argument of type *obj_module*; the intention is that to execute a program, the shell retrieves the appropriate universal object from the file system, converts it to type *obj_module*, extracts procedure *main* of type *unixprogram*, and invokes it with the *obj_module* as one of its arguments. The program may thereafter create new modules, and link them with its own module, giving a new extended module from which the executable forms of new procedures may be extracted.

In our discussion on stability, we will assume that our compiler, linker and binder can all be trusted to generate correct code, and that only they are allowed to create procedural objects. This is in fact quite easy: objects of type *obj_module* are simply not representable in source code. The types *text*, *src_module*, *type_desc* and their cousins should be standard, but they need not be protected.

Given that our compiler is capable all by itself of guaranteeing the integrity and security of a computer, we might model our system on the example of the Burroughs machines [Hauc68, Maye82] and rely totally on the compiler to the extent of dispensing with all hardware protection mechanisms, such as the user/kernel distinction and the storage of protection bits in segment/page table entries. This scheme is very attractive: it allows us to lower the cost of the hardware and it yields quite good security [Wilk81]. For pragmatic reasons, however, we would prefer to keep some of the hardware protection mechanisms, specifically the ability to run separate jobs in separate address spaces. This should allow us to handle programs written in other languages and compiled with other, untrusted compilers. The role of our trusted compiler would be to take untrusted object code and emit instructions that cause the execution of the untrusted code to take place in a separate address space, to arrange communication between that address space and the rest of the system, and to protect the rest of the system against malfunctions in the untrusted code.

These functions must in any case be present in every operating system that does not rely on a trusted compiler; see for instance the protection filter mechanism of PIMOS [Chik88]. (We will not consider foreign code in the rest of this thesis.)

4.4. Stability

Research in operating systems for logic machines has so far concentrated on two approaches. One is based on versions of Prolog enhanced with the necessary primitives for process handling and interfacing to the hardware. The resulting systems (e.g. SIMPOS [Taka84]) rely on side-effects even more extensively than other Prolog programs. The second approach is based on parallel languages such as Parlog, (Flat) Concurrent Prolog, and (Flat) GHC as described in [Clar84, Shap84b] and [Kusa86, Fost86, Fost87a, Silv86]. These systems communicate with the world via streams whose values are monitored by device drivers; the operating system itself can therefore be much more declarative.

Research on systems of the second kind has so far been based on languages that do not support don't-know nondeterminism, thus preventing programmers from writing search-type programs in a natural way. This is an important loss; the search paradigm is one of the major advantages of logic over functional programming.

There are two reasons for this restriction. One is that these systems need stream AND-parallelism to represent concurrent communicating processes [Emde79], and the integration of don't-know nondeterminism with stream AND-parallelism poses some difficult implementation problems that have only recently been addressed (see chapter 5 and the papers [Yang86, Clar87b, Bahg88a, Bahg88b, Hari88, Nais88a]). The other reason is that the connection of don't-know nondeterministic logic programs and deterministic devices presents three persistent problems. These problems all involve the variables whose values are monitored by device drivers; we call these *real-world-state variables*, or state variables for short.

(1) Alternative branches of the computation may give conflicting values to state variables. For example, the logic program may tell the printer `print the next character, which is 'a' or 'b'`.

(2) The computation may retract the value it has given to a state variable. The program may tell the printer `print 'a'`, and then tell it `erase that 'a', print 'b' instead`.

(3) The program may fail to give a value to a state variable in any of the branches. It may tell the printer `print UNDEFINED`; or even worse, the program may simply exit with failure.

These scenarios are examples of what we call *instability*; a stable computation is one that does not suffer from any of these problems. We give a formal definition in section 4.4.1 (note that this notion of stability has nothing to do with the notion of a stable clause selection strategy [Shap84a]). Our concern is with the stability of programs written in don't-know nondeterministic languages. In the next few sections we present a mechanism that can prove the stability of a program written in such a language at compile time and for all possible input data. We make four claims for its significance.

(1) Concurrent Prolog, GHC and Parlog disallow don't-know nondeterminism in order to prevent scenarios 1 and 2. This restriction is *sufficient* for the purpose, but, as we show, not *necessary*. Our mechanism works *even if large parts of the computation are don't-know nondeterministic*.

(2) To enforce their rule against don't-know nondeterminism, Concurrent Prolog, GHC and Parlog divide each clause into a guard and a body. Conceptually, every clause of their

programs contains a commit (alias trust) operator, though the syntax may allow it to be omitted from the textual form of the program. This implies that it is not immediately evident whether the declarative and operational semantics of a procedure coincide: they do if the guards are mutually exclusive and have single solutions, they don't if the commit operator can ever discard solutions. We do not require all clauses to have a guard. In our system, programmers need to use guards only when they want to throw away some solutions, explicitly choosing the operational over the declarative semantics. This helps both human readers of the program and automatic program transformation tools. In our system they need not perform complicated analyses to find out whether certain transformations, which depend on the equivalence of declarative and operational semantics, are applicable. (E.g. the transformation of $append(A, B, 1.2.nil)$, $A = 1.nil$, $B = 2.nil$ into $append(1.nil, 2.nil, 1.2.nil)$ is permissible only if one knows that *append* will not prematurely commit to another solution such as $A = nil$, $B = 1.2.nil$.)

(3) Of the three major committed-choice languages, only Parlog really tries to address the third scenario. Its mode system can ensure that calls do not terminate without instantiating the arguments they were supposed to instantiate (our mode system can do the same). Parlog's metacall [Clar84] also solves part of the other half of the problem: it can guarantee that a failing user program will not fail the entire operating system (the metacall always succeeds, but it may bind its status argument to *failed*). However, the metacall obviously cannot guard against the failure of procedures not called by it; therefore at least part of the operating system itself is vulnerable to bugs that cause its procedures to fail. Our mechanism is administered by the compiler; hence its protection extends to all code in the system, including the operating system.

(4) The languages Concurrent Prolog, GHC and Parlog each rely on the atomic update of variables to allow them to be connected to real-world devices. In each of these languages the implementation of atomic update requires run-time tests, in Parlog's case helped by compile-time checks. By contrast, our mechanism operates entirely at compile time.

Note that the above points compare our system only with the committed choice languages, and not with the new proposals for the integration of stream AND-parallelism and don't-know nondeterminism. The reason is that none of the proposed systems is suitable for the implementation of logic operating systems. P-Prolog [Yang86], Andorra [Hari88] and Pandora [Bahg88b] all implement don't-know nondeterminism via OR-extension, which is the virtual split of the *entire* computation into separate OR-branches (the three systems vary on how these splits are controlled). The problem is that the very possibility of these splits make it very difficult, if not impossible, to connect the output of a program written in one of these languages to a real-world device: how can the device choose between the alternative solutions ? Also, this form of OR-parallelism (which does not fit the AND/OR process model [Cone83]) seems to have a considerable amount of overhead. Though this overhead can be reduced considerably, it is still prohibitive for heavily don't-know nondeterministic programs [Bahg88a].

Another language that supports both stream AND-parallelism and don't-know nondeterminism is Parallel NU-Prolog [Nais88a]. Although PNU-Prolog implements don't-know nondeterminism via conventional backtracking, it has some other features that make it inadvisable to program parallel logic operating systems in it. Probably the most obvious is that all user programs would have to reside in one global database. Other problems involve PNU-Prolog's restrictions on communication between stream AND-parallel and don't-know nondeterministic code, and the difficulty of implementing its sequential side-effect predicates in a parallel environment.

Yet other papers propose to provide the functionality we desire by interfacing a stream AND-parallel environment (e.g. Parlog) with a don't-know nondeterministic one (e.g. Prolog).

This can be done by interfacing the runtime systems [Clar87b, Bahg88a] or by compiling the language of one environment into the language of the other [Ueda86a, Codi86]. However, all these systems restrict the ways in which the two environments can interact; these restrictions are generally intended to prevent a don't-know nondeterministic part from making any uncommitted bindings available to any other don't-know nondeterministic parts. This implies that when given the query *generate*(X), *test*(X) where *generate* has multiple solutions, most (though not all) of these systems must wait for *generate* to finish before invoking *test*. They therefore cannot exploit the parallelism available, and they waste CPU time computing a complete solution if *test* can reject a partial solution. We judge these to be significant limitations; we designed our system to overcome them.

Another problem with these systems is that they require programmers to know two similar but subtly different languages, a situation that encourages confusion and tends to inhibit the full use of either language [Wein71; p. 236]. (Up till now this effect may not have been noticed because the users of such systems have tended to be either their designers or other researchers who were already familiar with the two languages.) On top of this, they also require programmers to manage the interface between the two languages, and if they want a predicate to be available in both environments, they must write it in two languages. This last imposition is especially onerous because many procedures would most likely be needed in both environments.

For all these reasons we prefer a single unified environment, with just a single execution algorithm that is capable of both stream AND-parallelism and don't-know nondeterminism, We now consider how we can solve the stability problem in such an environment; we will discuss its implementation in chapters 5 and 6.

4.4.1. Definitions

We wish to know what conditions we should impose on don't-know nondeterministic programs that are to be connected to the outside world; we wish to know what makes such programs *stable*. One can easily find sufficient conditions: all computations of the form *Variable* = *constant* are stable (although they are not very useful by themselves). Finding necessary conditions, on the other hand, is impossible: the decision problems for the three scenarios are all unsolvable (we prove this later in section 4.6). What we need are sufficient conditions that are general enough to permit the most common paradigms of parallel logic programming but specific enough to be quickly decidable.

Our solution is to define three properties of computations, each designed to exclude one of the three scenarios. We call these properties *functionality*, *monotonicity* and *totality*. They resemble the mathematical notions they are named after, but the analogy cannot be exact; our primary interest in this section is the *operational* behaviour of programs.

Definition A procedure p in mode m is *functional* if it never (i.e. for no set of input bindings) has more than one solution. It is *monotonic* if it never retracts a binding it makes to any part of any of its output variables unless a consumer compels it to do so by rejecting the current set of bindings, causing backtracking to p. It is *total* if it never fails without first producing a solution.

Definition We say that a procedure p is *stable* in mode m if it is functional, total and monotonic. In that case it always has exactly one solution and it never makes an output binding that is not part of this solution.

Example The ideal implementation of *append*(A, B, C) is stable in the mode in which it constructs C from A and B, but it is only monotonic and total in the mode in which it splits C into A and B, since that split can be done in many ways. It is functional and monotonic in the mode in which it checks that a given C is the concatenation of a given A and B pair. The canonical example of a nonmonotonic procedure is the naive 8-queens program *queens* that produces a permutation of the integers from 1 to 8 and then tests whether the permutation is safe. If this test fails (as it usually will), the outside world will see *queens* retract part of the binding of its output variable without a rejection by an outside consumer.

Example An example of the syntax of stability declarations:

```
procedure append(A, B, C: listof[T])
mode
        A, B => C       [stable];
        C => A, B       [monotonic total];
        A, C => B       [functional monotonic];
        B, C => A       [functional monotonic];
        A, B, C         [functional monotonic];
```

We do not intend that the compiler should analyze programs to determine if they are functional, monotonic or total. Instead, we rely on declarations supplied by the programmer. For each mode of each procedure these state whether that procedure is functional, monotonic or total in that mode. The role of the compiler is merely to check these declarations. This arrangement results in better documentation and a more efficient compiler.

Definition An occurrence of a variable X in a call in the expression defining procedure p is a *functional occurrence* if in each invocation of p it will see at most one value of X. It is a *monotonic occurrence* if in no invocation of p will it ever observe any part of the value of X being retracted. It is a *total occurrence* if in all invocations of p it will see at least one value of X. We say that an occurrence of X is a *stable occurrence* if it is functional, total and monotonic.

Example The intuition behind these definitions is that a variable occurrence is e.g. functional provided that if and when its call is activated, the variable appears to be produced by a functional procedure. In $p(X) \,/\, q(X, Y)$, where p produces X and q produces Y, the occurrence of X in $q(X, Y)$ is stable even if $p(X)$ is not stable. This is because the guard throws out all of p's solutions except one (hence X is functional at q), and because q will not be started unless and until p has succeeded (hence X is monotonic and total at q). Note, though, that any occurrences of X outside the expression $p(X) \,/\, q(X, Y)$ itself would not necessarily be stable; if such occurrences exist then the guard is *unsafe* [Clar86].

In the rest of the thesis we will often talk about *a procedure* being stable (functional, etc) but this is only shorthand for *a procedure in a given mode* being stable. The mode should be obvious from the context: in the case of a call to a procedure, it is the mode assigned to the call by the compiler. In effect we consider each mode of a procedure to be a different “procedure”. Similarly, we say that a variable is functional (or monotonic, or total, or stable) if it is functional (or monotonic, or total or stable) in the occurrences in which we are interested.

Because of the inherently operational nature of our definitions, whether a particular procedure or variable occurrence has any of the three properties depends on the implementation. The details of how such questions should be answered are presented and discussed in section 4.6. First, a few comments on the definitions themselves.

4.4.2. Notes on the definitions

Even though we defined the notions of functionality, monotonicity and totality with state variables in mind, the definitions themselves do not refer to state variables. It makes perfect sense to say that a procedure is stable or unstable even if that procedure has nothing to do with any state variable.

A particular computation must of course be functional or total (or both). For procedures, though, the notions of functionality and totality are orthogonal. All four combinations are possible: a procedure may be non-functional for one set of inputs and non-total for another. Similarly, monotonicity is orthogonal to functionality and totality.

If an expression has no outputs, then it should always be functional, as the only solution it can have is the empty binding. It must also be monotonic: since it cannot make any output bindings, it cannot retract any. It should not be total; if it is, then it is equivalent to the atom *true* and can be deleted from the program. (In theory the functionality of such expressions is implementation dependent, but every implementation based on a strong mode system should be smart enough to compile these expressions as if they were followed by a cut, guard or whatever mechanism the language uses to eliminate unwanted search paths.)

In general, as the number of their inputs increases, procedures are more likely to be functional and monotonic and less likely to be total. If a procedure is functional (or monotonic) in a mode m, it is also functional (or monotonic) in any modes implied by m; and if it is total in mode m, it is also total in any modes that imply m.

The standard Prolog notion of *determinacy* [Mell85, Nais85b] is similar to our notion of functionality. The difference between them is nevertheless significant. The definition of determinacy relies *only* on implementation considerations, i.e. on whether there is a choice point left on the stack. It is not related to the program text in any clean way. From the text itself, a procedure in (standard) Prolog can be called determinate only if it has cuts in all of its clauses, except possibly the last one [Sawa85]. Functionality is a cleaner concept in that it does not rely on non-logical primitives. When a functional procedure succeeds, we *know* that the remaining clauses (if any) cannot contribute to another success; we can therefore discard any remaining choice-points. In this it is similar to the notion of functionality as defined in [Debr86a] and to the notion of *logically deterministic* procedures as defined in [Naka86]. The definition of *binding determinacy* in [Nais88a] is roughly equivalent to the combination of our functionality and monotonicity: in PNU-Prolog, the syntactic sugar of *lazyDet* and *eagerDet* declarations hides the non-logical primitives used to implement them.

Our notion of functionality differs from the one in [Smol84]. In Smolka's system, a procedure declared to be functional may in fact have multiple solutions, but all solutions following the first will be discarded. In our system, the compiler checks that a procedure declared to be functional will have only one solution; if programmers want to throw away solutions, they have to use guards explicitly. Smolka's paper has no notion of monotonicity, but it has a notion of totality, and this notion is very close to ours. Smolka's totality assertions specify that a procedure is total if certain arguments are free and distinct; this corresponds to declaring the procedure total in a certain mode. The difference is that in our system declarations of totality are verified by the compiler whereas in Smolka's system they are not.

4.5. Guaranteeing stability

From the point of view of an operating system, the world is a well-defined sequence of states. A don't-know nondeterministic program has no trouble with these states as input. However, when asked to compute the next state, it may give many solutions (effectively asking us to make copies of the world !), or none at all. Our objective is to show that

(1) one can build a stable computation out of unstable parts without a *compulsory* guard mechanism,

(2) and that one can guarantee the stability of such a computation at compile time even though the computation may call procedures constructed at run-time.

We address (2) in sections 4.5.2 and 4.6. As an existence proof of (1), consider the following code fragment. It prints a solution of a problem, or nothing if there is none:

```
{
        solve(P, S), test(S)          ▯ append(S, Otail, Out)
alt
        ˜∃ S: {solve(P, S), test(S)}  ▯ Out = Otail
}
```

solve is a procedure that produces a solution *S* given a parameter *P*; *test* tests whether the *S* is an acceptable solution. We assume that *S* is of type string and that strings are implemented as lists of characters. (As we will show in section 4.7, the common subexpression can be eliminated.) Procedure *solve* need not have any guards in its definition; it may have any number of solutions, and it may backtrack any number of times before producing one that is acceptable to *test*. The code as a whole is nevertheless stable: it will lead to stable computations for all values of *P*. The instability of the first guard is not visible from outside the braces. We know that the guards are safe; if *S* were consumed outside the braces then the program would have a mode error, as the second alternative does not produce *S* for outside consumption.

We achieve the side-effect of printing the solution not by invoking a built-in non-logical primitive, but by instantiating a variable whose value is monitored by a device driver. In this case, the variable is *Out*, and it is a list of characters to be printed. Conceptually, our code produces the difference between *Otail* and *Out*. It takes *Otail* as input and produces *Out*, though obviously it cannot (and does not) wait for *Otail* to be instantiated: that would take literally forever.

4.5.1. Regions of the computation tree

We view computations as trees whose nodes are expressions: the children of each node are the active (running) components of the expressions. This implies that the tree changes as expressions start and finish execution. In the example above, the procedure in whose body the above code occurred would at first have two children, one for each guard. The first guard would have two children, one for *solve* and one for *test*; the second guard would have one, for a negation process (which would itself call *solve* and *test* one level down). When one or other of the guards succeeds, it kills the processes active in the other, so that the parent would momentarily have no children; then it will create a new process to execute whichever body is selected.

The computation we are concerned with is rooted at a call to the procedure *operating_system*, whose arguments are streams of messages to and from devices. We divide the tree of this computation into stable and unstable regions. Regions are rooted at nodes that represent unstable calls whose parent is stable or vice versa (for our present purposes, we consider calls to be unstable if they are not *known* to be stable). For example, while the guards are executing in the above example, each of the child processes is the root of an unstable region. On the other hand, the call that executes the body (whichever body that is) is stable, and furthermore it can't tell which if any of its inputs were produced in an unstable region; it knows only that all its inputs *appear to be* produced by stable procedures. The guards hide the instability of their left hand sides.

A node of the tree is *stable with respect to* one of its ancestors if its outputs are stable whenever that ancestor's inputs are stable. This implies (a) that the node must represent a stable expression (a call to a stable procedure or a subexpression of the body of a stable procedure that has been found to be stable by the analysis of section 4.6) and (b) that the inputs of that expression must themselves be stable whenever the ancestor's inputs are stable. A region is *locally stable* if its root is locally stable with respect to itself; it is *globally stable* if it is locally stable and its root is the root of the whole tree. In figure 4.2, the region labelled GS is globally stable, the regions labelled LS are locally stable, and the regions labelled US are unstable.

4.5.2. The enforcement mechanism

The parts of the operating system dealing directly with real-world devices of course have to execute in the globally stable region. Command interpreters ("shells" in Unix terminology) and user programs deal with the world through the operating system, but the parts that give commands to the operating system would likewise have to execute in the globally stable region. The contribution of our scheme is that the operating system, shells, and user programs may all rely on unstable parts, as long as their stability as a whole is preserved. For example, the operating system may rely on unstable procedures supplied by the user to decide whether to permit a program to access that user's files, but it must know what to do when the test fails, indicating denial of access.

The compile-time nature of our stability mechanism forces us to confront the following dilemma. How can we ensure that the operating system is stable when we don't know what shells it may have to invoke, and how can we ensure that the shells are stable when we don't know what user programs they may have to invoke ? It can happen, for example, that a shell is stable only if the user programs it invokes in a particular place are themselves stable (it is also possible that the shell's stability does *not* depend on the stability of the user programs it invokes in other contexts). The answer of course is to do some checking at run time; and the checking we need is exactly what is provided by the delayed compiler invocation of *break_cocoon*.

The code of example 4.3 retrieves a program from the file system and then executes it; it is a more realistic version of the code in example 4.2. The two changes are both necessary to ensure that the code as a whole is stable. First, the operating system may not be able to return the named file; it may not exist, or the user on whose behalf the request is made may not have access to the file, or somebody else may have an exclusive lock on the file, or some other error may be present. To accommodate these possibilities, we modify the type of the second argument of *fs_lookup* so

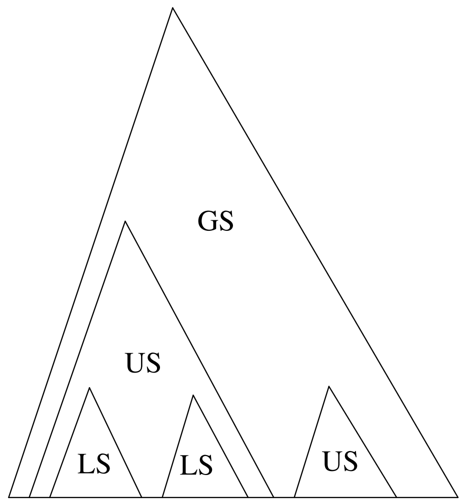

Figure 4.2: regions of the computation tree.

```
Os0 = fs_lookup(Name, Response).Os1,
{
        Response = error(Err) &>
                append(Err, Out1, Out0),
                shell(In1, In9, Out1, Out9, Os1, Os9)
or
        Response = found(Obj) &>
        {
                ~break_cocoon(Obj, P) &>
                        append("error message", Out1, Out0),
                        shell(In1, In9, Out1, Out9, Os1, Os9)
        or
                break_cocoon(Obj, P)  &>
                        P(In1, In2, Out0, Out1, Os1, Os2),
                        shell(In2, In9, Out1, Out9, Os2, Os9)
        }
}
```

Example 4.3: executing a user program (refined).

that the operating system may bind it to either *found*(*Obj*) if it can return the file or to *error*(*Err*) if it cannot. The code must handle each case in a stable manner. The other change is that we now check whether the *break_cocoon* operation succeeds or not; again the code must be stable for both cases.

As we shall see in the next section, the above code is stable as long as *P* is bound to a stable procedure. However, *P* cannot be bound to anything else, because its type (which we have shown in section 4.1) specifies that it must be stable; if *Obj* happens to encapsulate an unstable procedure (or a procedure of the wrong type or mode, or a non-procedural value), then the *break_cocoon* operation will fail. The *break_cocoon* operation thus serves as our enforcement mechanism.

4.6. Stability analysis

The decision problems of the functionality, monotonicity and totality properties (and hence of the three scenarios of section 4.4) are all unsolvable. To prove this informally, we present three small

```
p1(X) :- X = a.                 % program 1: functionality
p1(X) :- tm, X = b.

p2(X) :- tm, X = a, fail.       % program 2: monotonicity
p2(X) :- X = b.

p3(X) :- tm, X = a.             % program 3: totality
```

Example 4.4: proof of undecidability.

programs (in Prolog syntax), in example 4.4. In each case, *tm* is a call to a procedure that simulates an arbitrary Turing machine on arbitrary input. The first program proves that we need to be able to solve the halting problem to solve the decision problem of functionality; the second and the third do the same for monotonicity and totality. (Our first program also proves the undecidability of determinacy, a property closely related to functionality; this "proof" is simpler than the one in [Sawa85].)

However, the undecidability of these problems need not prevent us from achieving our objective: to be able to guarantee a program's stability at compile time. We could possibly make the compiler call a stability inference system (along the lines of type- and mode inference systems), and accept the possibility that the inference system returns "maybe" as an answer. We adopt what we feel to be a better solution: to cut the Gordian knot by requiring programmers to declare the stability properties of each mode of each procedure. These declarations state whether the programmer expects the procedure to be functional, monotonic and/or total in that mode; the default answer is *no* to all three questions. This prevents the compiler from wasting time on trying to prove the stability of unstable procedures; it can concentrate on verifying the stability of the procedures the programmer expects to be stable. (We gave an example of the syntax of these declarations in section 4.4.1.)

Another advantage of our approach is that it decomposes the problem of declaration verification into small, independent subproblems. The compiler checks each mode of each procedure independently to see whether the procedure body satisfies its declarations. (Here we are interested only in stability-related declarations, but in practice the compiler would be checking type- and mode declarations at the same time.) While checking one mode of one procedure, the compiler needs to know only the declarations of other procedures; it assumes that the bodies of those procedures satisfy their declarations. If this assumption turns out to be incorrect, the compiler will find out when it checks the errant procedures. This decomposition increases the speed of the compiler, and, more important, leaves us free to make the compiler operate incrementally. Note that if the system supports separate compilation (as it should), some of the checks may have to be done by the linker.

When checking whether a procedure conforms to its stability declarations in one of its modes, the compiler analyzes expressions according to their structure, as we show in the rest of this section. Given that a completely error-free analysis is impossible (no-one can prove the totality of a Turing machine simulator), we prefer to make the compiler's analysis a simple one. Other tools can still make more complicated analyses and transform the program so that its stability is visible to the compiler; we cover this idea in more detail in section 4.7.

Since functionality, monotonicity and totality are inherently operational notions, one cannot lay down rules that will work for every implementation. The analyses in the rest of this section apply to "typical" implementations.

We assume that no variable can be both input to and output from an expression; any circularities in the dataflow graph must involve at least two conjoined subexpressions. For example, we can calculate the Fibonacci sequence using procedure *seqplus*, which takes two lists of integers and produces a third, each element of which is the sum of the corresponding elements in the first two lists. To simplify some of the analysis below, we analyze the code `seqplus(1.1.X, 1.X, X)` as if it were written `seqplus(1.1.Y, 1.Y, X), X = Y`. This type of transformation can be done quite trivially by the compiler, and it helps the mode analysis algorithm as well.

After the covering the analysis rules in the rest of this section, we will present an example of how the analysis works in section 4.7.

4.6.1. Primitive expressions

The base cases of our checks are the built-in procedures. The most important of these is the unification procedure $U = V$. It has three normal modes. In the assignment modes U => V and V => U, unification is stable, while in the test mode, where both U and V are input, it is only functional and monotonic. Unification is also stable in its special modes, which apply when at least one of U and V is void. Each of these decisions apply only to one type tree node; a call to the unification procedure is stable only if it is stable for all the nodes of the type trees of its arguments.

We consider two other primitives, *plus* and *mult* (a complete language would of course have many more). Procedure *plus*(A, B, C) is stable in all three of its *principal* (non-implied) modes, which are A, B => C; A, C => B; and B, C => A. The analogous result holds for the first mode of *mult*(A, B, C), but not for the other two unless we make *mult* return a special error value on division by zero. In their implied modes (with all three arguments input) both *plus* and *mult* are functional and monotonic but not total.

4.6.2. Simple expressions

Negation An expression of the form ~*e* cannot be total unless there is something seriously wrong with the program (i.e. *e* always fails). However, it must always be functional and monotonic, because, to ensure soundness of negation, we do not allow the negated expression to have any non-void outputs. If it has a void output, i.e. it generates a variable that has no consumers, we analyze it as if it were written as ~$\exists X$: *e*.

Quantification The expression $\exists X$: *e* has whatever properties *e* has: it is functional if *e* is functional, monotonic if *e* is monotonic, and total if *e* is total. In addition, $\exists X$: *e* is functional and monotonic if X is the only output of *e*. The expression $\forall X$: *e* is equivalent to ~$\exists X$: ~*e* and we analyze it as such.

Implication We do not deal with this expression constructor; we deal with its expanded form instead. We analyze the expression *e1 -> e2* as if it were written *~{e1 & ~e2}*.

Conjunction The conjunction *e*1 & *e*2 is functional if both *e*1 and *e*2 are functional. and it is total if both *e*1 and *e*2 are total. For it to be monotonic, a more complicated condition has to hold: if *ei* contributes directly or indirectly to the value of a variable that is consumed outside the conjunction, then it must be monotonic; and if *ei* directly exports any variables then the *other* conjunct has to be total. All this implies that simple conjunctions cannot make stable computations out of unstable ones.

The first requirement for monotonicity expresses the transitivity of the monotonicity property: an expression cannot produce stable outputs if its inputs are unstable. To see the need for the second requirement, consider that on the failure of either conjunct the backtracking algorithm will either cancel the other or backtrack to it; both these actions can lead to the retraction of bindings. Failure itself also leads to retraction, of course, but by definition a monotonic expression cannot fail by itself after producing any bindings. This is why conjuncts exporting variables to the outside need not be total unless the other conjunct requires them to be.

Disjunction The disjunction *e*1 *or* *e*2 is monotonic if both *e*1 and *e*2 are monotonic, and it is total if either *e*1 or *e*2 is total. In the general case, we cannot say anything about its functionality, nor can we say anything more about its totality.

We can, however, say something in special cases. The most useful one occurs when each disjunct contains a unification of the form `X = fi(Yi1, ..., Yim)`, where X is input and the *Yi* are outputs (or void). We can then say that the disjunction is functional if the *fi*'s are exclusive (distinct), that it is always monotonic, and that it is total if the *fi*'s are exhaustive (the type of X will tell us what functors X may be bound to). Therefore the disjunction is stable if and only if each alternative functor for X appears exactly once.

Disjunctions of the form *e or ˜e*, where (to ensure mode correctness) neither occurrence of *e* can have any outputs, are also special: they are always stable. This is not surprising, since they are equivalent to *true*. However, we will find a use for them in the next two sections, where the "no outputs" restriction does not apply.

Other kinds of disjunctions also have useful properties. We do not, however, analyze them; instead we transform them into one of the above forms. We describe the transformation process in section 4.7.

4.6.3. Conditional expressions

To exploit the special forms of disjunction, we need either a sequential AND construct or guards. We explore the first alternative in this section and the other in the next.

The expression $e1$ & > $e2$ says that the execution of $e2$ will not be started until $e1$ has been completely evaluated; to avoid deadlocks, we do not allow $e1$ to consume variables produced by $e2$. The operational semantics of the sequential AND operator is, unlike that of guards, sound with respect to the declarative semantics of conjunctions, although it may be incomplete. $e1$ & > $e2$ has the same stability properties as the expression $e1$ & $e2$, with one difference. The failure of $e1$ cannot affect $e2$, because $e2$ cannot be executing at the time; hence $e1$ need not be total for the conjunction to be monotonic regardless of whether $e2$ exports variables outside the conjunction.

The most useful manifestation of the sequential AND operator is probably in expressions of the form $c1$ & > $e1$ *or ... or* cn & > en, where $c1$ *or ... or* cn is one of the special disjunctions of section 4.6.2. We call this a *conditional expression* if the variables produced by the ci are not consumed outside the disjunction, although they may be consumed in the corresponding ei. This requirement is analogous to the guard safety condition. Stability analysis must verify guard safety both for conditional expressions and for guard expressions.

A conditional expression of the above form is functional if the disjunction $c1$ *or ... or* cn and all the ei are functional; it is monotonic if all the ei are monotonic; and it is total if the disjunction $c1$ *or ... or* cn and all the ei are total. Therefore the whole expression is stable if the disjunction $c1$ *or ... or* cn is functional and total and all the ei are stable (the ci do not have to be stable by themselves, and they do not have to be monotonic at all). In that case, any variables produced by one of the ci for the corresponding ei will be stable at ei. The functionality and monotonicity parts of this are obvious; to see the totality part one should note that ei will be executed *only* if and when the corresponding ci has succeeded and hence produced values for the variables it shares with ei. In other words, only one of the ei will be called, and that one will behave as if its inputs were produced by a stable procedure.

Another form of disjunction also becomes special in the context of conditional expressions and in the context of guard expressions. An expression of the form c & > $e1$ *or* $\tilde{}\exists X{:}\ c$ & > $e2$ can be stable even if c has outputs, if those outputs are used only in $e1$, not outside the expression itself (guard safety again). This is the case in the example of section 4.5.

Conditional expressions bear a syntactic resemblance to sets of guarded commands, though semantically they are closer to *if* and *case* statements. (They also have the same effect as the all-single-neck procedures of P-Prolog [Yang86], although the underlying mechanisms have almost nothing in common.) Unlike guards, they do not involve don't-care nondeterminism. Their semantics can therefore be much cleaner; programmers should use them instead of guards whenever possible.

Another advantage of sequential ANDs over guards is that declarative debuggers need to be able to repeat any given execution history. For conditional expressions, this history can be reconstructed at any time. For guard expressions, traces must be stored, since otherwise there is no guarantee that a second execution of a guard construct would have the same effect as the first [Nich88]. A compiler analysis proving that the guards of a procedure are mutually exclusive can reduce the volume of these traces, but such analysis can be difficult to perform without the help of a strong mode system.

4.6.4. Guard expressions

Although sequential ANDs can replace guards in many cases, they cannot replace them in all cases. The canonical case is the *merge* procedure. When an operating system merges streams of requests to a server, it does not want to know all the possible ways this can be done. All it wants is one of those ways; it does not care which. Situations such as this (when one *wants* to throw away solutions) are the proper province of guards.

As one would expect, the functionality condition is not as strict for guard expressions as for conditional expressions. *g*1 **|** *b*1 *alt ... alt gn* **|** *bn* is functional if all the *bi* are functional; there is no need for *g*1 *or ... or gn* to be functional as well. The monotonicity condition, however, is the same as for conditional expressions, as is the totality condition and the condition for the stability of variables in the *bi*.

The replacement of & > by **|** has two effects: all but one solution of the guards is discarded; and all but one of the alternatives is discarded. The second of these effects can be avoided by writing `{ g1 ▯ b1 } or ... or { gn ▯ bn }`; the first cannot. To get the second effect without the first, a construct like Smolka's exit clauses is necessary [Smol84], though in principle it could be emulated using a primitive like *set_of* (see section 4.6.5 below).

GHC allows a body to start executing before its alternative is selected for commitment, subject to the condition that it should not instantiate the parent environment before commitment. If the corresponding guard fails, or another alternative succeeds in committing first, the time and resources spent on this body are wasted. Therefore GHC implementations should execute such bodies only when a processor is idle and has (literally) nothing better to do.

We believe that implementations of our system should not take advantage of this possibility, for two reasons. The first is that the gain is small: a parallel logic language should easily be able to generate more than enough work for any foreseeable multiprocessor. The second is that the cost is high. Without eager bodies, stable procedures may instantiate their outputs without any prior checks; if their inputs are stable, their outputs will be stable also. If, however, a stable procedure is invoked from the eager evaluation of a body, then it must suspend until its clause has in fact been committed to before instantiating any of its outputs. If it neglects to do so, and the system commits to some other clause, then the eager body evaluation destroys the monotonicity property. Our system has no other use for this kind of suspension mechanism, and its introduction would create significant new overhead. This is in contrast to (full) GHC implementations. They already need such a mechanism for guards, and their overhead is less significant since they already incur it for guards.

4.6.5. All-solutions expressions

An all-solutions expression is a goal of the form *set_of*(T, e, S), where e is an expression and T is a template, a tuple consisting of output variables of e. The call binds S to a set of tuples, each derived from a solution of e using the template T. The expression e is type- and mode-checked like other calls.

set_of should of course be total; even if *e* has no solutions, *set_of* should have one, with S being the empty set. (Unfortunately, in most implementations of Prolog the all-solutions predicate fails when *e* itself fails [Nais85a].) If S is represented as a list, the programmer can look inside it, and therefore it must be sorted if *set_of* is to be functional [Nais85a]. We can easily make *set_of* monotonic as well: we just wait until we have all the solutions before we bind S.

It may be useful, though, to make parts of S available before the computation of *set_of* terminates. This way, if the consumer of S needs only a part of the value (e.g. if it needs only to verify that there are at least two solutions), we can avoid computing the other solutions. We could implement this by instantiating S to $X.S1$ as soon as *e* succeeds with the T-tuple X, or, if *e* is monotonic, as soon as it instantiates any of its outputs. These techniques preserve monotonicity but preclude S from being sorted. To keep *set_of* functional, we must then build the notion of *set* into the language. One way is to put the solutions of *e* in the set in the order if which they arrive, but give to consumers of the set a random solution from among those produced so far, returning as a separate argument the remaining elements of the set (which need not have been computed yet). This scheme prevents the consumers from (reliably) extracting significant information from the ordering of the set.

Because of our mode system, *set_of* has no problems with variables either in the inputs or in the answers. The input variables of *e* will be given values by other calls. As for outputs, our modes guarantee that any variables in solutions are distinct (no links either within or among solutions) and that the pattern of their location in the structure is known at compile time (and hence is the same for all solutions). This second property implies that if two solutions are unifiable, they must be identical.

4.6.6. Metacall expressions

In Prolog, *metacall* refers to a goal *call*(*e*) that acts (or at least aims to act) as if the program text read just *e*, but with the difference that *e* may be supplied at run-time. In the context of logic operating systems, though, a variant of the metacall primitive that never fails is indispensable, so [Clar84] introduced two-argument metacalls. Their *call*(*e*, S) always succeeds, binding the status variable S to *succeeded* or *failed* according to whether *e* succeeded or failed. (That paper also introduced three-argument metacalls, while [Fost87a] introduced five-argument metacalls; these are versions of the two argument metacall with added control mechanisms. In this section we are concerned only with stability, not with control aspects.)

This arrangement makes perfect sense if *e* has no outputs. In a strongly moded system, a problem arises if *e* does have outputs, because success and failure are asymmetric: the former produces output bindings while the second does not. Our solution is to make the bindings part of the status, rather than exported directly from the *call* as in Parlog; thus we have no problems with bindings lingering after the failure of the call that generated them. Our metacall is of the form *call*(T, *e*, S) and it will bind S either to *succeeded*(X) or *failed*, where X is a T-tuple derived from a solution of *e* (as for *set_of*, T is a template containing output variables of *e*). If *e* has some solutions, *call* returns *succeeded*(X) for every solution X of *e*; if *e* does not have any solutions, *call* returns *failed*. One could say that *call*(T, *e*, S) returns what *set_of*(T, *e*, S) would return, but one solution at a time.

call(T, *e*, S) is of course always total. In its simplest implementation it is functional if *e* is functional, and monotonic if *e* is monotonic. However, we may wish to offer versions of the metacall that have different properties. For example, a primitive *mono_call*(T, *e*, S) could ensure monotonicity by delaying the binding of S to *succeeded* until *e* is completely evaluated. Another primitive, *func_call*(T, *e*, S), could ensure functionality by discarding all solutions of *e* except the first. A third primitive, *direct_call*(*e*), could act like *call* in Prolog, and would have the same properties as *e* itself (i.e. it would not guarantee totality).

For both all-solutions and metacall expressions, the compiler derives its knowledge about the stability properties of *e* from requiring the declaration of these properties for all procedure-valued variables, as we discussed in section 4.5.2.

Our mode analysis algorithm needs to be modified to handle metacall and all-solutions expressions. *direct_call*(*e*) should be handled exactly like *e*; the handling of *set_of*, *call* and its variations requires procedure *expr* to map nodes of T onto the nodes of S and vice versa. In the second case, procedure *expr* should also disallow the scheduling of any part of the called expression until all its parts can be scheduled, because even if the called expression can produce some of its outputs given a strict subset of its inputs, the metacall or all-solutions expression can't.

4.7. Program transformations and stability analysis

Our analysis, as presented in section 4.6, can detect stability only in a relatively small class of expressions. This is all we need, however, because we can transform other stable expressions into forms which fall into this class, and we can do so automatically. The main tools we use for this are macro expansion and common subexpression elimination.

Our approach is best illustrated by an example. Suppose the original expression is

```
        X < Y &> stable1(...)
or
        X = Y &> stable2(...)
or
        X > Y &> stable3(...)
```

The compiler, following the analysis of section 4.6, cannot prove that this expression is stable, because it has no way of knowing that the conditions are exclusive and exhaustive. So we transform the expression first into

```
        compare(X, Y, Cmp_r), Cmp_r = less    &> stable1(...)
or
        compare(X, Y, Cmp_r), Cmp_r = equal   &> stable2(...)
or
        compare(X, Y, Cmp_r), Cmp_r = greater &> stable3(...)
```

by macro expansion and then into

```
compare(X, Y, Cmp_r),
{
        Cmp_r = less    &> stable1(...)
or
        Cmp_r = equal   &> stable2(...)
or
        Cmp_r = greater &> stable3(...)
}
```

by common subexpression elimination. *compare* is the new comparison primitive, replacing <, =, and >; *Cmp_r* stands for "comparison result". The technique works for guard expressions as well as for conditional expressions. It can also be easily extended to handle code such as $X \geq Y$.

The macro expansion is a simple match-pattern-and-replace process; its complexity is linear in the size of the procedure and can easily be made independent of the number of patterns (by hashing on the macro name, e.g. <). Common subexpression elimination is just as straightforward. Its complexity is quadratic, but the typical procedure is small enough for this not to matter. In any case, if we wish we can reduce the size of the problem by comparing only the atoms introduced by macro expansion.

This transformation has other uses as well. It can avoid the timing problems associated with the use of the primitives *var* and *nonvar*. An apparently stable expression like

```
        nonvar(X) &> stable1(...)
or
        var(X)    &> stable2(...)
```

may fail if *nonvar*(X) executes first and fails, some other process instantiates X, and then *var*(X) executes and fails. The transformed program

```
bindingstate(X, Bst_r),
{
        Bst_r = bound &> stable1(...)
or
        Bst_r = free  &> stable2(...)
}
```

does not suffer from the same problem, and unlike the untransformed variant, can be proven stable by the rules of section 4.6. Again, *bindingstate* is a new primitive replacing *var* and *nonvar*.

A useful variation of this transformation changes

```
        e  &> stable1(...)
or
        ~e &> stable2(...)
```

into

```
call([T1, T2], e, Call_r),
{
        Call_r = succeeded(X1, X2) &> stable1(...)
or
        Call_r = failed            &> stable2(...)
}
```

This transformation does not improve stability (the original expression is already recognised to be stable), but it will remove the redundant computation of *e*, making the expression about as efficient as an *if* statement.

We apply a different transformation when conditions or guards have unifications that are more complex than the ones we discussed in section 4.6.2. For example, one of the conditions in the *server* procedure of example 2.5 contains the unification $S = cons(read(D), S1)$. In order to recognise the stability of procedure *server*, we must transform this code into a form that the rules of section 4.6.3 are applicable to. To do so, we simply apply anti-substitutivity and rewrite the unification as $S = cons(Msg, S1)$ & $Msg = read(D)$. Since unification is a top-down process, we are allowed to change the new & operator into & >.

We apply this transformation to two of the clauses of procedure *server*, and transform the third by the introduction of a null goal. We then apply common subexpression elimination, taking into account the associativity of the & > operator. The result is

```
        S = cons(Msg, S1) &>
        {
                Msg = read(D)   &> server(S1, D)
        or
                Msg = write(D1) &> server(S1, D1)
        }
or
        S = nil &> true
```

The rules of section 4.6 recognise the stability of this code.

The common subexpression recogniser and eliminator have to be part of the compiler. Most macros will come from a standard library that contains definitions for the built-in procedures (e.g. *compare* and *bindingstate*), but there is no reason why some cannot be defined by the user, perhaps using the mechanisms of [Chik84] and [Kond88]. The compiler analyzes the output of these macros, so the user cannot use them to subvert the stability checks (or the type- and mode checks for that matter). The metacall-introducing variation and the application of anti-substitutivity do not need macro expansion, but they require the support of other parts of the compiler. For example, the introduction of metacalls requires knowledge of the mode assignment, and so it cannot be performed completely syntactically.

Note that many of these transformations should probably be done by the compiler in any case, as their outputs are much more amenable to easy code generation. For example, the transformed version of procedure *server* corresponds directly to the best execution strategy for that procedure: wait until S is instantiated, dispatch on its value, and if that value is *cons* then repeat the process on its first argument (this scheme is very similar to the decision tree compilation algorithm of [Klin88a]).

If a procedure turns out to be unstable in a mode in which it has been declared stable, then the compiler has found a bug. These bugs are quite likely to be in the control part of the procedure; in some cases we can fix the control part automatically while leaving the logic intact. The simplest fix introduces monotonicity by mutating a standard parallel AND operator into a sequential AND, possibly permuting conjuncts in the process. This kind of change is probably needed in most programs; it is certainly needed for the standard definition of *append* (see example 4.5).

The body of example 4.5 does not match its declarations. For example, the code is not monotonic in the first mode: it could instantiate C to B, and later discover that A is not equal to *nil*. The problem is that $A = nil$ is not total when used as a test and it needs to be total if the first clause is to be monotonic. However, the compiler can notice that $A = nil$ would not need to be total if it were followed by a sequential AND operator instead of a parallel one. Picking up clues

```
procedure append(A, B, C: listof[T])
mode
        A, B => C       [stable];
        C => A, B       [monotonic total];
def
        A = nil, B = C
or
        A = H.AT, C = H.CT, append(AT, B, CT)
end append
```

Example 4.5: original version of append.

like this, the compiler could transform the above code into the one below. (It has to analyze the two modes of the procedure separately, so it is not too much to ask that it do its source level transformations separately, especially since it has to compile different object code for each mode to cater for their different synchronisation needs.)

The body of example 4.6 does match its declarations. However, programmers would find it inconvenient to write this kind of code themselves. The compiler should therefore accept the first form, and convert it automatically to the second if the user so requests. This request should be in the form of a *pragma*, a directive to the compiler contained in the program itself.

As an example of the analysis techniques of section 4.6, we now demonstrate that the second form indeed matches its declarations. We start with the first mode: A, B => C. The code for this mode is in the form of a conditional expression. As such it is stable if the disjunction of the conditions is functional and total and both bodies are stable. The disjunction of the conditions, $A = nil$ *or* $A = H.AT$, is one of the special disjunctions, as A is input and H and AT are outputs. It is functional because the functors *nil* and *cons* are exclusive and it is total because the functors *nil* and *cons* are exhaustive. The first body is a call to a stable procedure (unification in assignment mode), the other is a conjunction of calls to stable procedures (unification in assignment mode and *append* itself in its first mode),

```
procedure append(A, B, C: listof[T])
when mode A, B => C [stable];
def
        A = nil  &> B = C
or
        A = H.AT &> C = H.CT, append(AT, B, CT)
when mode C => A, B [monotonic total];
def
        A = nil, B = C
or
        C = H.CT &> A = H.AT, append(AT, B, CT)
end append
```

Example 4.6: modified version of append.

The second mode is a bit more complex. Its body is not a conditional expression, merely a disjunction. The first alternative is monotonic and total because it is a conjunction of two assignment unifications (in fact it is functional as well, but in this case we do not need to know that). Since the first alternative is total, the disjunction is total as well; but for it to be monotonic, the second alternative also has to be monotonic. In this case, both sides of the sequential AND operator contribute to the outputs A and B, so both must be monotonic; but only the right half does so directly, so neither side has to be total, although the right side does turn out to be total (the left side would need to be total if the AND operator were a parallel one). The left side of the & > is monotonic because it is a call to a monotonic procedure; the right side is monotonic because it is a conjunction of two calls to monotonic and total procedures, assignment unification and *append* itself in its second mode (the assignment to A is functional as well).

At this point a question naturally arises: can these techniques transform all stable procedures into forms which the compiler recognises as stable ? The answer is no. One can still write programs that (like the programs at the start of section 4.6) require the compiler to prove an arbitrary theorem to show that the program is stable. With more sophisticated analyses we might discover the stability of wider classes of stable programs, but we can never hope to do so for all stable programs.

Only actual experience with a system constructed along our lines will tell whether this incompleteness is a problem in practice, but we feel that the answer will almost certainly be *no*. After all, with appropriate macro libraries our system should be able to recognise the stability of the relevant constructs of the flat concurrent logic programming languages FCP [Mier85] and FGHC [Ichi87]. If it does happen that the compiler does not recognise stability in the programs of an important paradigm, then we should get the compiler to do a more sophisticated analysis (bearing in mind that this can be expensive even when not required) or write a tool that performs more sophisticated transformations. If after this a programmer still cannot convince the compiler that a given program is stable, he/she will probably also have a hard time convincing other humans of that fact. The simplicity of our stability conditions is deliberate: it benefits both man and machine.

Chapter 5

Execution algorithm issues

In this chapter we present a backtracking algorithm for stream AND-parallel logic programs; to the best of our knowledge it is the first such algorithm. To let this algorithm determine efficiently which goals to kill or restart when a goal fails, we rely on compile-time knowledge of the dataflow graph of each clause. This crucial information is implicit in the mode assignment of the clause; it was not available at compile-time in any previous stream AND-parallel system.

We show that modes can increase the precision of the backtracking algorithm, though our algorithm allows this precision to be traded off against overhead on a procedure-by-procedure and call-by-call basis. The modes also allow our algorithm to handle efficiently programs that manipulate partially instantiated data structures and an important class of programs with circular dependency graphs. On code that does not need backtracking, the efficiency of our algorithm approaches that of the committed-choice languages; on code that does need backtracking its overhead is comparable to that of the independent AND-parallel backtracking algorithms.

These algorithms are in fact our starting points [Lin86b, Woo86, Cone87]. Our main contribution is to show that these algorithms can be easily extended to handle stream AND-parallelism given a strong mode system, because such a system tells us at compile-time the producer of every variable. Without this information, we would not know how to order the calls in a clause (the notion of backtracking itself requires such an order), nor would we know which other goals to kill or restart when a goal fails (we have to kill the consumers of the variables produced by the failed goal). Having this information available at compile-time is crucial because of the great expense of collecting this information at run-time.

The backtracking algorithm is *intelligent* if it can select backtrack literals that are close to the true source of a failure. This intelligence is desirable because it prevents the system from wasting CPU time on the exploration of parts of the search space that would lead to the repetition of the same failure. The independence information collected by independent AND-parallel execution models can make their backtracking algorithms made more *intelligent* than the standard chronological algorithm used by Prolog: they can make minor improvements quite easily [Borg86, Herm86c], and more significant ones with only a little more effort [Lin86b, Woo86, Cone87]. It turns out that modes can also be used to increase the precision of the backtracking algorithm: they state at compile time the call the system should backtrack to in order to change the value of a given variable. Our backtracking algorithm is at least as intelligent as the independent AND-parallel ones and in some cases it can do better than they can.

Our backtracking algorithm interacts with the forward execution algorithm only through a narrow interface (which we describe in section 5.2.2). As a result, one can use our backtracking algorithm in conjunction with any forward execution algorithm that conforms to the requirements of this interface. In particular, our backtracking algorithm does not impose any restrictions on scheduling policies: it allows goals to be started in any order.

As we stated earlier, one of our aims is that one should be able to write an operating system in a language whose implementation incorporates our algorithm. This application area dictates two of the properties of our backtracking algorithm. One is that backtracking is localised: when executing $p(X)$, $q(Y)$, the internal backtracking of p cannot affect q. This is important because they may be executing jobs for different users: users would be rightly angry if they discovered that their jobs were continually being cancelled and restarted just because a goal failed inside some other user's job. The other requirement imposed by our application area is that we must allow circular dataflow graphs, since these are quite common in systems-type logic programs, and are indeed necessary for any implementation of the client-server model. We accordingly allow dataflow graphs to be circular in all situations where we can guarantee that there will be no backtracking along the circularities. This is quite a lenient condition: it permits one to write all the programs that one can write in the committed-choice languages (since these languages have no backtracking whatsoever).

As presented, our algorithm handles normal programs as defined by [Lloy87] (i.e. the definitions of procedures are disjunctions of clauses, each of which is a conjunction of literals). More general forms of logic programs, with arbitrary first-order formulae as procedure bodies, can be translated to this form [Lloy84], as we discussed in section 1.2. We must make special provisions for programs with guards, but these special provisions are naturally localised in the forward execution algorithm, concerned as they are with synchronisation and the killing of processes after the success (not the failure) of guards. Some methods for the implementation of guards need no support from the backtracking algorithm; others require from it an ability to handle multiple active AND-processes within a single OR-process. Although our algorithm does not allow for this, it can be modified to do so reasonably easily.

5.1. Backtracking operations in the AND/OR process model

Backtracking algorithms in the AND/OR process model are all based on knowledge of each clause's *dataflow graph* (see section 5.3.1 for a discussion of its derivation). The nodes of this graph are the calls of the clause (or of the query). An arc points from one node to another if the call represented by the first node produces a variable consumed by the second node. These arcs can be naturally labelled with the set of variables whose values flow along them. In the past, though, they have often been left unlabeled; previous backtracking algorithms just did not care about *which* variables connect a producer and a consumer. We, however, exploit this information to make our algorithm more precise. To do this, we need to modify slightly the set of basic operations of the backtracking algorithm. These operations are invoked by messages that flow up and down the branches of the AND/OR tree.

We are not in this thesis interested in OR-parallelism; for the sake of simpler exposition we would like to dispense with the OR processes. We do so by combining a call's OR process and its current AND process into a single process for that call, thus converting the AND/OR tree into an AND tree.

5.1.1. The old set of operations

Our work is based on the AND-parallel backtracking algorithms described in [Lin86b, Woo86, Cone87]. The main operations of these algorithms are the *redo*, *reset* and *cancel* messages, which travel from parent process to child process, and the *success* and *failure* messages, which travel in the opposite direction. The semantics of these messages are as follows (we will use the goal $p(X)$, $q(Y)$, $r(X, Y)$ as an example):

redo — This message, sent by the parent, asks the child to generate a new solution. It is sent to q when r fails, and to p if q fails after that.

reset — This message, sent by the parent, tells the child to start delivering its solutions from the beginning. It is sent to q when p generates a new solution, to make sure that r can test all combinations of X and Y. A *reset* message does not kill the child if it has a cache of results returned so far (see section 5.3.2 for a discussion of result caches).

cancel — This message, sent by the parent, kills the child. It is sent to r whenever p or q generates a new solution.

success — This message, sent by the child, notifies the parent of the success of the child and specifies the values of the child's output variables.

failure — This message, sent by the child, asks the parent to generate new values for the child's input variables.

5.1.2. The new set of operations

Our algorithm is based on just three types of messages: *get_next*, *get_first* and *here_is*. They are analogous to the operations *redo*, *reset* and *cancel* respectively, but each of our operations expects an argument. These arguments are sets of variables, and their purpose is to make the operations more selective. They also help our *get_next* and *here_is* messages to play the role of *failure* and *success* messages respectively.

We have three reasons for introducing new names for our operations. The first is to prevent the confusion of parameterised and non-parameterised operations. The second is to prevent the confusion of downward-only and bi-directional operations. The third is that we feel that the new names are more descriptive than the old ones.

The semantics of our operations are as follows.

get_next Vs — This message tells the recipient process to generate new bindings for at least one variable in the set *Vs*. A *get_next* message is analogous to a *redo* message when travelling downward and to a *failure* message when travelling upward.

get_first Vs — This message tells its recipient to start generating the bindings of the variables in *Vs* all over again. A *get_first* message is analogous to a *reset* message; it only travels downward.

here_is Vs — This message tells its recipient that all the variables in the set *Vs* have had their values retracted, and that they should soon get new bindings. These bindings should be different from the old ones unless a producer generates the same value twice in a row. A *here_is* message is analogous to a *cancel* message when travelling downward and to a *success* message when travelling upward.

In our moded stream AND-parallel system, a process will in general be started before all its input variables have been given values. Consumers of a variable may therefore need to wait for it to be instantiated. Since the delays themselves have no effect on our algorithm, it is easiest to ignore them, although the potential for delays is always present. In this context (which may take a little getting used to) the name *here_is* really means what it says.

The argument of a *here_is* message specifies *which* variables have changed, *but does not specify the new values*. The reason is that such information is not available at the time *here_is* messages are sent: a stream AND-parallel system executing $p(X)$, $q(X)$ need not wait for a *here_is* message from p before commencing the execution of q. This is an important difference between *here_is* and *success* messages: one should keep it in mind when reading the algorithms to follow.

Our motive for merging the *success/cancel* and *failure/redo* pairs of messages is the close correspondence between their intuitive semantics. Besides, the result is a uniform pattern of message traffic. *Here_is* messages travelling upward lead to other *here_is* messages being sent downward and to the right; *get_next* messages going upwards cause further *get_next* messages to be sent downward and to the left.

5.1.3. Examples

We now give some examples. We use conventional Prolog syntax, with the exception that our programs are in homogeneous form. In the homogeneous form of a clause unification is pushed into the body so that the head is composed of distinct variables. (This need *not* prevent us from using clause indexing [Warr83]; in fact, modes can reduce the size of the indexing code by eliminating the need for *switch_on_term* instructions.) To show the structure of the dataflow graph, in each clause one occurrence of each variable appears in bold: this is the occurrence that gives the variable its value (this notation is similar to the one in [Clar86]).

The usefulness of the extra precision of *get_next* is demonstrated by the following code.

```
?- p(X, Y), q(X), r(Y).
p(X, Y) :- X = a, p1(Y).
p(X, Y) :- X = b, p2(Y).
```

Example 5.1: precision of get_next.

If q fails, the algorithms of [Lin86b], [Woo86] and [Cone87] will all backtrack to $p1$, even though no solution of $p1$ can remove the cause of q's failure. This phenomenon is called the *clause boundary limitation* [Toh86]. Our operations, though, can tell p to get the next value of X, and thus backtrack directly to its second clause. In this case, the behaviour of our algorithm is as precise as the behaviour of the intelligent backtracking methods based on unification (e.g. [Pere80]). The difference is that our algorithm has much lower overhead, due to the *static nature of mode information*. Some later sequential algorithms, e.g. [Toh86, Kuma88], can also produce this behaviour with relatively low overhead.

A similar example demonstrates the extra precision of *here_is*:

```
?- p(X), q(Y), r(Z), s(X, Y, Z), t(Z).
s(X, Y, Z) :- X = a &> s11(Y), s12(Z).
s(X, Y, Z) :- X = b &> s21(Y), s22(Z).
```

Example 5.2: precision of here_is.

When t fails, we backtrack to r; we therefore need to reset s. We do so by sending it a *here_is {Z}* message. X and Y, remaining unchanged, are not included in this message. That being so, the process for s does not need to start over from scratch; it can just forward the *here_is {Z}* message to $s12$ or $s22$, whichever is active at the moment. This behaviour also surmounts the clause boundary limitation, this time going forwards.

5.2. The backtracking algorithm

Our algorithm is closest in spirit to those of [Lin86b] and [Woo86]. In low-level details it most closely resembles [Cone87], in that our algorithm is intended to be implemented using bit vectors. All four algorithms consist of alternating forward and backward execution phases. Forward execution is the normal mode of computation; backward execution (backtracking) is invoked on failure. In our algorithm, each backward execution phase handles one message plus any avalanche messages. We require that these phases be atomic; see section 6.2 for details on how this can be achieved in multiprocessors.

5.2.1. Definitions and static data structures

In the presentation of our algorithm we will consistently adopt the viewpoint of a process Q, executing procedure q. We will assume that its parent is the process P and that its children are the processes Ri for $i \in 1..n$, each created to solve a call in the current clause of q. We follow [Cone87] in representing the head of the clause (and hence indirectly the parent P) as two nodes or "processes" in the dataflow graph. One, $R0$, stands for the head in its role as producer of variables for the clause; the other, $Rn+1$, stands for its role as a consumer. Note that they play precisely opposite roles when viewed from P: whatever variables $R0$ produces for the calls in the body it imports from P, and whatever variables $Rn+1$ consumes from the calls in the body it exports to P.

We store the dataflow graph of the current clause in the data structures PV and CV. $PV(Ri)$ is the set of variables produced by the process Ri, and $CV(Ri)$ is the set of variables consumed by Ri. Here, i ranges from 0 to $n+1$. $PV(R0)$ is the set of Q's inputs while $CV(Rn+1)$ is the set of Q's outputs; $CV(R0)$ and $PV(Rn+1)$ are both empty. If $v \in PV(Ri)$ and $v \in CV(Rj)$, then the dataflow graph has an arc labelled v from Ri to Rj. In our system, all CV and PV sets are available at compile time, derived automatically from mode declarations written by the programmer. And as the number of both variables and calls in a clause is usually quite small, we can implement these sets of variables as bit vectors [Cone87]; we will have more to say about this in section 6.3.

We say that a variable $v1$ *directly depends on* variable $v2$ if $v1 \in PV(Ri)$ and $v2 \in CV(Ri)$ for some call Ri, and that a call Ri directly depends on a call Rj if $v \in CV(Ri)$ and $v \in PV(Rj)$ for some variable v. We also say that a call Ri directly depends on a variable v if $v \in CV(Ri)$, and that a variable $v1$ directly depends on a call Ri if $v1$ directly depends on $v2$ and $v2 \in PV(Ri)$. The *depends on* relation is the transitive closure of these notions. We assume that no variable depends on itself, i.e. that the dataflow graph is noncircular (this assumption is not quite the same as the traditional noncircularity assumption; see section 5.4.2). We may therefore insist that the calls in each clause be numbered so that if the process producing the variable v is Ri, then consumers of v should be Rj for $j > i$ (we will relax this assumption in section 5.3.3).

We say that variables $v1$ and $v2$ are *connected* if a call Rk depends on both of them. With each variable $v1$ produced by some Ri, we associate a set $inner(v1)$, so that $v2 \in inner(v1)$ if $v1$ and $v2$ are connected, $v2$ does not depend on $v1$, and the producer of $v2$ is Rj for some $j > i$. *inner* sets are related to the candidate sets of [Cone87], but they are defined for variables instead of processes, and they take no part in backtrack literal selection. They are used only for determining which processes to reset, in accordance with the nested loops model [Cone85].

Example Consider the clause $a(Z) :- p(\mathbf{X}), q(\mathbf{Y}), r(Y, \mathbf{Z}), s(X, Z)$, where a's argument is output and the producers of X, Y and Z are p, q and r respectively. In this case, $PV(R0)$ is $\emptyset$ and $CV(Rn+1)$ is $\{Z\}$; $CV(p)$ is $\emptyset$ and $PV(p)$ is $\{X\}$; $CV(q)$ is $\emptyset$ and $PV(q)$ is $\{Y\}$; $CV(r)$ is $\{Y\}$ and $PV(r)$ is $\{Z\}$; and $CV(s)$ is $\{X, Z\}$ and $PV(s)$ is $\emptyset$. Among the variables, Z depends on Y. Among the calls, r depends on q; s depends on p, on r, and through r on q; and $Rn+1$ depends on s, and through s on p, q and r. X and Z are connected because s depends on both; X and Y are also connected because s also depends on Y through Z. The *inner* set of X is $\{Y, Z\}$, and the inner sets of Y and Z are $\emptyset$.

Procedure p is *collapsible* by a set of its input variables if for all possible values of those variables, all the bindings exported by p are produced by a single clause. Consider the following procedure:

```
p(A, B, C, D, E) :- A = a1 &> p11(B, D), p12(C, E).
p(A, B, C, D, E) :- A = a2 &> p21(B, D), p22(C, E).
```

Example 5.3: collapsibility.

At most one of the two unifications involving *A* can possibly succeed. Since each clause has one of these unifications, and since they must succeed before any other part of the clause may execute, the variable *A* collapses *p* to a single clause. We denote this by saying that the set of collapsing variables of *p* (written as *coll_vars*(*p*)) is equal to *{A}*. In general, the *coll_vars* set contains the inputs of the mutually exclusive tests on the left of the sequential AND operator, or the inputs of the (not necessarily mutually exclusive) tests on the left of the guard operator. In both cases we insist that the tests should not export any bindings directly to $Rn+1$ (the usual guard safety condition). Note that locally determinate procedures [Mell85, Nais85b], are a special case of collapsible procedures; they occur when the tests involve only head unifications pushed into the body. See section 4.6 for ways of proving properties related to collapsibility at compile-time; see [Yang86] for ways of testing such properties at run-time.

5.2.2. Dynamic data structures

Backtracking algorithms for independent AND-parallelism have to take an active hand in scheduling: they must ensure that calls are not started until other calls that produce their inputs have been completed. Our algorithm, being based on stream AND-parallelism, is completely independent of scheduling decisions. We therefore assume a separate scheduler that decides independently which processes to start up. The scheduler communicates with our algorithm only through arrays of booleans called *started*, of which there is one in every active clause. *started*(*Ri*) is true if a process has been created for *Ri*. The rest of the forward execution algorithm maintains a parallel set of arrays called *finished*: *finished*(*Ri*) is true if a solution of *Ri* has been completely and successfully evaluated (the *finished* bit is not set on failure). The two arrays have to be updated atomically; their bits are set during forward execution and reset during backward execution. The three possible states of the *started*/*finished* bits of *Ri* are reminiscent of the gather/execution/finished states of [Lin86b] and of the blocked/pending/solved states of [Cone87], but they play completely different and much reduced roles.

To keep track of the history of failures, we associate a set of variables called the $R-set$ (for variables “related” to failure) with each child process. When backtracking to *Rj* after the failure of *Ri*, we record in $R-set(Rj)$ those variables involved in the failure of *Ri* that were not produced by *Rj*. When *Rj* itself fails, the variables in its $R-set$ are indirectly involved in its failure; their producers must also be considered to be potential backtrack points. $R-sets$ resemble the B-lists of [Lin86b] and are related to the redo cause sets of [Woo86] and the candidate sets of [Cone87]. However, being based on variables rather than processes, they can take advantage of the extra precision of our model. $R-sets$ are initialised to empty on clause entry and restart.

When we backtrack out of a clause, we know which input variables contributed to the failure of the clause (see example 5.1). The union over all clauses of these sets of variables is the set of variables responsible for the failure of the process *Q* itself. This information is useful for guiding further backtracking, being a limited form of *match failure analysis* [Toh86], so we save it in a data structure we call $S-set$ (for suspect set). These $S-sets$ have the same function as the inputsets of [Bruy80], but can be implemented with much less overhead. $S-sets$ are initialised to empty on procedure entry and restart.

5.2.3. The algorithm

Our algorithm is invoked on the failure of some call. This causes a *get_next* message to be sent upward, and from then on the work is done by message-handling code. Messages travelling upward are never directly replied to, but they cause other messages to be sent (some of which may find their way back to the originator of the upward message). Messages travelling downward are always replied to, and in one case such replies are handled together with previous messages (see *here_is* algorithm below). Note that the messages a child receives from its parent specify subsets of the child's outputs for *get_next* and *get_first* messages and subsets of the child's inputs for *here_is* messages, while the situation is reversed for messages the child sends to its

Remember, we assume that the current process is Q, whose parent is the process P and whose children are the processes Ri for $i \in 1..n$.

5.2.3.1. Algorithm for primitives

The basis of our recursive algorithm is the behaviour of the primitive processes; the principal one is unification. Its behaviour is very simple: start over on receipt of a *get_first* or *here_is* message, sending to parent a *here_is* message for all outputs; and send a *get_next* message to parent for all inputs on failure and on the receipt of a *get_next* message.

5.2.3.2. Algorithm for negation

We handle negated calls by interposing a negation process between the process of the caller and the process of the callee. If this negation process receives a *get_next* message from its child (i.e. the child has failed), it reports completion by setting its *finished* bit to true. On the other hand, if the child reports completion, it sends a *get_next* message to its parent, specifying all its input variables as suspects. A *here_is* message from the parent is passed on to the child. The negation process will never receive *get_next* or *get_first* messages from its parent, because a negated call may have no (non-void) outputs; we rely on the compiler to enforce this restriction.

5.2.3.3. Algorithm for get_next Vs from child Ri

type	direction	arguments	reply
get_next	upward	inputs	here_is or kill
get_next	downward	outputs	here_is or get_next
get_first	downward	outputs	here_is
here_is	upward	outputs	no reply
here_is	downward	inputs	here_is

Table 5.1: characteristics of message types.

```
set Ws := Vs ∪ R−set(Ri)
set j := i−1
while j > 0 and (PV(Rj) ∩ Ws = ∅ or not started(Rj))
        set j := j−1
end while
if j > 0
        set R−set(Rj) := R−set(Rj) ∪ (Ws − PV(Rj))
        send get_next PV(Rj) ∩ Ws message to child Rj
        wait for reply from child Rj
        execute corresponding algorithm
else
        set S−set(Q) := S−set(Q) ∪ Ws
        if Q has alternative clauses
                kill all started children
                start up the next alternative clause
                send here_is CV(Rn+1) to parent P
        else
                send get_next PV(R0) ∩ S−set(Q) to parent P
        end if
end if
```

Ws contains the set of variables involved in the failure of *Ri*, either directly or indirectly. While the variables in *Ws* have their current values, no solution is possible, so we must change at least one of them. The loop finds the rightmost node in the dataflow graph that produces any such variables ("hence" our algorithm is complete with respect to the nested loops model [Cone85]). If this node represents a child process, we tell this process to produce new values for those of its outputs that were involved in the failure. It may reply with a *here_is* message, to tell us which of its outputs will actually change. (If so, the arguments of the two messages must have a common element, but neither has to be a subset of the other. To remove the failure, a change in one variable may be enough; and it may not be possible to generate a new value for one variable without doing the same for another.) Otherwise, if the child has no way to generate solutions that can remove the failure, it will reply with a *get_next* message,

If the node found by the loop is the head in its role as generator, we know that the clause has failed. If *Q* has other clauses, we backtrack to one of them, signaling *P* that all of *Q*'s outputs will get new values ($CV(Rn+1)$ is the same in all clauses). If there are no such clauses, the whole process *Q* has failed. The set of variables involved in this failure is the suspect set ($S-set$); it is the union of the sets of variables involved in each clause's failure. (Note that we never check whether *Ws* intersects with $PV(R0)$. If a clause contains a call like $a = b$, it will fail with an empty *Ws*. In that case, *Ws* does not intersect with anything, but backtracking out of the clause is still the correct thing to do.)

5.2.3.4. Algorithm for get_next Vs from parent P

execute the algorithm for *get_next Vs* from $Rn+1$.

Like [Cone87], we handle a *get_next* message by considering it to represent the failure of the head in its role as a consumer. We consider the $R-set$ of $Rn+1$ to be always empty.

5.2.3.5. Algorithm for get_first Vs from parent P

```
if collapsible(Q)
        set Ws := Vs ∪ {u: v depends on u, v ∈ Vs}
        set j := 1
        while j <= n and PV(Rj) ∩ Ws = ∅ or not started(Rj)
                set j := j+1
        end while
        if j <= n
                send get_first PV(Rj) ∩ Ws to child Rj
                wait for reply from child Rj
                execute corresponding algorithm
        else
                send here_is ∅ to parent P
        end if
else
        kill Q
        reset the started bit of Q in the calling clause
        send here_is CV(Rn+1) to parent P
end if
```

We handle the reset of a non-collapsible procedure the same way as most other algorithms, by starting over from scratch, telling *P* that all the outputs will probably change. Starting *Q* over from scratch means that we kill it and all its subcomputations and reset its *started* bit in the calling clause; the scheduler will do the rest.

If the procedure *is* collapsible, we can do better than this. We search for the leftmost started process that contributes to the value of a to-be-reset variable. If there is no such process, either because the test has not finished yet or because the scheduler just has not started any, then *Q* has produced no outputs so far and hence there is no need to reset any children. If we do find such a process, we reset it; it will reply with a *here_is* message. All further work will be done in response to this message. All output variables are connected by $Rn + 1$, so any to-be-reset variables that are not produced by *Rj* will be in the *inner* sets of the variables that are. They will therefore be also reset by the *here_is* algorithm. Note that these are the actions that the parent would have taken had the body of the chosen clause been used to *unfold* its call to *q* [Tama84]. However, this unfolding cannot be done statically, as it depends on the values of the variables in *coll_vars*(*q*) (and, for non-exclusive guards, on chance).

5.2.3.6. Algorithm for here_is Vs from child Ri

```
reset the finished bit of Ri
set NVs := Vs
set RVs := ∪ inner(nv) for each nv ∈ NVs
for j := i+1 to n
        if NVs ∩ CV(Rj) ≠ ∅ and started(Rj)
                send here_is NVs ∩ CV(Rj) to child Rj
                wait for here_is UVs message from Rj in reply
                set NVs := NVs ∪ UVs
                set RVs := RVs ∪ (∪ inner(uv) for each uv ∈ UVs)
        end if
        if RVs ∩ PV(Rj) ≠ ∅ and started(Rj)
                send get_first RVs ∩ PV(Rj) to child Rj
                wait for here_is UVs message from Rj in reply
                set NVs := NVs ∪ UVs
                set RVs := RVs ∪ (∪ inner(uv) for each uv ∈ UVs)
        end if
end for
if NVs ∩ CV(Rn+1) ≠ ∅ or parent expects a reply
        send here_is NVs ∩ CV(Rn+1) to parent P
end if
```

The first thing we do when we are informed that a child is working on a new solution is reset its *finished* bit (which may not have actually been set before). Then we must send *here_is* messages to some processes and *get_first* messages to others. We maintain two sets of variables to help us do this: *NVs*, the set of new variables (variables whose value has been retracted and which will get new values soon), and *RVs*, the set of to-be-reset variables (variables whose values we need to start computing again from the start). We sweep forward from the source of the message, sending messages as necessary, waiting for replies, and updating *NVs* and *RVs* accordingly.

A call may get two messages if it consumes changed variables and produces to-be-reset ones. Most of the time, though, the *here_is* message kills its recipient; in that case the *started* bit that was set for the first test will be clear for the second. We do not need to send either *here_is* or *get_first* messages processes that have not started yet, since the effects of process startup duplicate those of both messages.

We may be executing this algorithm either as a result of internal backtracking or because *P* has sent us a message. In the first case, we only need to signal *P* if some of the variables it consumes are in *NVs*. In the second case, *P* expects a reply, so we must send one, even if none of the variables *P* consumes have changed.

5.2.3.7. Algorithm for here_is Vs from parent P

```
if collapsible(Q) and Vs ∩ coll_vars(Q) = ∅
        execute the algorithm for here_is Vs from R0.
else
        kill Q
        reset the started bit of Q in the calling clause
        send here_is CV(Rn+1) to parent P
end if
```

We handle a *here_is* message sent to a non-collapsible procedure the same way as most other published algorithms, by starting over the execution of Q from scratch. We also do this if the procedure is collapsible but the collapsing variables have changed. If they have not changed, we may still be executing a test, in which case we have nothing to reset, or we may have committed to this clause, in which case again we do what the parent would have done had the call to q been unfolded. The *here_is* algorithm handles both of these cases.

5.2.4. Examples

The examples in section 5.1.3 show that our algorithm is free of the clause boundary limitation, both going backwards and (to a lesser extent) going forwards. Our algorithm also incorporates a form of *match failure analysis* [Toh86]: it tries to make sure that the backtrack literal it selects not only produces a variable consumed by one of the failed literals but also that the variables it produces had actually contributed to one of the failures. When r fails in the program

```
?- p(X), q(Y), r(X, Y).
p(X) :- X = x1.
p(X) :- p1(X).
q(Y) :- Y = y1.
q(Y) :- q1(Y).
r(X, Y) :- X = x2, r1(Y).
r(X, Y) :- X = x3, r2(Y).
```

Example 5.4: match failure analysis.

the suspect set of the process for r contains only X if the unification is scheduled first, and hence we backtrack directly to p, skipping q.

Our algorithm does not insist on procedures retracting all their output bindings on receipt of a *here_is* message. If the procedure is collapsible and if the collapsing input variables have not changed, then at least some of those bindings may remain in force. Which ones these are is decided at run time.

In example 5.5, procedure r is collapsible by X. When the process for q generates a *here_is {Y}* message, we do not kill r out of hand, as all other backtracking algorithms would do; we merely forward the message to r. If the current value of X is $x1$, then r does not actually need the value of Y, and the reply of r would be empty; we can therefore let s run unhindered. If the value of X is $x2$, on the other hand, then the *here_is* reply from r would specify Z and we would need to send a *here_is* message to s.

To illustrate the effect of procedures that have not been started yet, consider the following example when p and r have been started but q has not. (Independent AND-parallel systems do not allow this; the fact the ours does is a consequence of the independence of our scheme from any

```
?- p(X), q(Y), r(X, Y, Z), s(Z).
r(X, Y, Z) :- X = x1 &> Z = z1.
r(X, Y, Z) :- X = x2 &> Z = Y.
```

Example 5.5: exploitation of collapsibility.

particular forward execution algorithm.)

```
?- p(X, Y), q(Y, Z), r(X, Z).
```

Example 5.6: exploitation of non-started calls.

Suppose p sends us a *here_is {Y}* message caused by its internal backtracking, as described in section 5.2.3.3. The other algorithms would cancel r, as it depends on Y through Z. In this case, though, the *here_is* algorithm bypasses q; therefore Z is not added to the new variables list, and r can continue execution. When q is started, it will compute Z from the new value of Y; process startup thus duplicates the effect of the *here_is* message.

The *started* bits also help us to avoid propagating the effects too far. Consider the following example when q has been started but r has not:

```
p(X, Y) :- q(X, Z), r(Z, Y).
```

Example 5.7: limited propagation of messages.

Suppose q sends p a *here_is {Z}* message caused by internal backtracking. Because r is not yet started, we would not send a *here_is {Y}* message to p, even though Y depends on Z. The siblings of p can therefore continue undisturbed.

5.3. Optimisations

We now consider ways in which our algorithm can be speeded up. We will concentrate on high-level optimisations; we do not go into low-level "tweaking" of the implementation other than to note that our algorithm has a fair amount of room for it. For example, in the algorithm for *here_is* from Ri we can quickly check with a shift-and-mask operation whether any of the started bits of $Ri+1$ through Rn are set. If they are all clear, we can avoid the main loop of that algorithm; this will save a lot of time in environments dominated by sequential computation.

5.3.1. What are we optimising ?

The time taken by a backtracking algorithm is the product of two factors: the number of backtrack steps executed and the time taken per step (this is a variant of the RISC principle [Patt85]). Ideally, one would like to reduce both these measures at the same time, but experience has shown us that they are mutually antagonistic. The chronological backtracking algorithm of Prolog has very low overhead, but frequently gets lost in blind alleys, exploring large parts of the search space that have nothing to do with the failure. As a reaction to this, the early unification-based intelligent backtracking algorithms [Pere80, Cox81] attempted to minimise the number of backtrack steps executed. They succeeded in this by gathering a lot of information about the flow of data, but since they did this at run-time, their overhead was unacceptably high. The same is true for Conery's original execution algorithm, which relied on a dataflow graph constructed mostly at run-time to guide backward execution [Cone83].

The obvious way to reduce overhead while keeping precision is to collect the necessary information just once. Accordingly, the algorithms of [DeGr84] and [Chan85a] derive their dataflow graphs from a compile-time analysis of the program. These analyses are necessarily approximate, since they are based on Prolog, a language that does not enforce or even encourage regular patterns of information flow. When the analyzer cannot determine whether two calls can safely be run in parallel, the system must execute those calls in sequence and must always backtrack to the rightmost. To exploit more of the available parallelism and to enable more precise backtracking, DeGroot's algorithm supplements the compile-time analysis with run-time checks of the form $ground(X)$ and $indep(X, Y)$. However, such checks can take a long time on large inputs, thus increasing overhead again.

At this point the simple, low-overhead chronological algorithm starts to look attractive. Unfortunately, chronological backtracking is not a viable alternative for us. Though it can be adapted for parallel implementations by substituting virtual time for real time [Clea88], the substitution itself inevitably involves overhead. The real reason though is that some behaviours that are acceptable in a (possibly simulated) sequential environment are not acceptable in a stream AND-parallel environment. For example, we do not want the internal backtracking of p in $p(X)$, $q(X)$ to affect q, unless of course this backtracking changes the value of X. Such independence is crucial when p and q interact with the real world: these interactions usually cannot be "rolled back" as required by the notion of virtual time, because rollback may require the erasure of characters already printed (for printers), the reacquisition of money already dispensed (for ATMs), and in the extreme case the reassembly of a city and a nuclear warhead from a mushroom cloud (for launch control systems).

The fact is that both overhead and precision are important. Overhead is the dominant consideration for deterministic and nearly deterministic programs; these constitute the majority of code executed by many logic programming systems. On the other hand, precision is the more important factor for strongly nondeterministic programs. It greatly reduces the time wasted in exploring dead ends. Just as important, it tends to smooth out any differences in execution time due to the ordering of producers; the programmer therefore does not have to worry about this ordering. Also, if the backtracking algorithm stops at a goal whose solution cannot remove the cause of the failure, we risk getting stuck in an unnecessary infinite loop. With a fair computation rule (such as bounded depth-first), we may be able to avoid this "Turing hole", but of course in this case our point about smoothing differences in execution time applies even more strongly.

None of the algorithms we mentioned in this section has both relatively low overhead and high precision. They make different choices for the same tradeoffs: how much information to collect and when to collect it. Nevertheless, being all based on Prolog, they cannot make certain choices. They cannot collect precise data-flow information at compile time because such information is simply not available then. Our algorithm breaks this pattern: it requires the programmer to supply extra information in the form of mode declarations. Based on these declarations, the compiler designates, for each variable, exactly one call in each clause as its producer. The dataflow graph is therefore static; we do not need to discover it at run-time. (Note: the papers [Lin86b, Woo86, Cone87], on whose algorithms our algorithm is based, do not touch on the derivation of the dataflow graph, although some of the authors of these papers have attacked this problem in other papers, e.g. [Lin86a].)

5.3.2. Optimisations not included

We have decided not to incorporate several previously-known optimisations into our algorithm. The optimisations are result caches, selective reset, special handling of multiple failure, and total match failure analysis. None of these optimisations is fundamentally incompatible with our algorithm, and each is undoubtedly useful in certain situations. However, we believe that in our intended application area (general purpose computing) these situations are not frequent enough to justify the overhead required to exploit them.

A result cache is a list of the previous solutions of a literal. An algorithm that keeps such a cache can use it to avoid recomputing those solutions when the literal is reset. Result caches were first described in [Cone83], and they are retained by most intelligent backtracking methods based on the AND/OR process model [Lin86b, Woo86, Cone87, Wins87]. Now the only time that we can be sure that a process has computed a solution is when its evaluation is complete. Such events require action from algorithms based on independent AND-parallelism, but not from our stream AND-parallel algorithm. To incorporate result caches, we would therefore need a new run-time mechanism, whose overhead is likely to be fairly high. For example, calls that can produce results without all of their inputs being instantiated may be forced to wait for all their inputs.

However, as long as we are contemplating a new mechanism, there is no reason why we cannot make it a general one (although we would of course need to be able to apply it to only a *subset* of all procedures). Goal caches [Fagi86] and their cousins extension tables [Diet87] reuse solutions whether the goal is invoked from the same environment or not. Their only drawback is that they may require more time to index into, but compared to the time saved by not recomputing goals, this overhead should be insignificant.

Selective reset [Wins87] is an optimisation technique for result caches; as we have decided against result caches, we must also decide against selective reset. We also have two other reasons. First, clauses tend to have few calls and simple dataflow graphs, whereas selective reset works best for large clauses with complex graphs. Second, its effect can be approximated by a goal cache that records failures as well as successes.

Another opportunity for optimisation appears when two or more connected calls in a clause fail at the same time [Woo86]. One can then invoke the backward execution algorithm just once; and though this invocation must compute the backtrack literal of each failure, it can backtrack directly to the leftmost, resetting connected goals as usual (the other backtrack points are among the connected goals). The only significant overhead of this optimisation is the test for its applicability; nevertheless we judge this to be too much. The reason is that multiple failures in the same clause at the same time should be very rare. In any event, in the complementary case, when the simultaneous failures are not related, our algorithm has the same effect as Woo & Choe's: we send a *get_next* message to each of the backtrack points. The only difference is that we do this in separate backward execution phases.

```
?- p(Y), q(X), r(X, Y).
p(Y) :- Y = a.
p(Y) :- p1(Y).
q(X) :- X = b.
q(X) :- q1(X).
r(X, Y) :- X = c, Y = d.
```

Example 5.8: total match failure analysis.

Another optimisation is what we call total match failure analysis (MFA). To the best of our knowledge, it first appeared in [Pere80] (where it was called the AND rule; our terminology is based on that of [Toh86]). In example 5.8, our algorithm, with its limited form of MFA, will find that the goal $X = c$ fails, and will therefore backtrack to q as the producer of X. A system with total MFA, on the other hand, will not stop after the failure of $X = c$, but will complete the match of the goal $r(a, b)$ against the “head” $r(c, d)$. It will therefore observe the independent failure of the two unifications, and will send *get_next* messages to both p and q. Total MFA is thus an amalgamation of limited MFA and the multiple failure optimisation. For it to be effective, we need to execute head unification processes (and possibly other primitive goals) even beyond a failure (otherwise we will not get multiple failure; simple head unifications are so small that they should not be spawned off on other processors). This is a form of speculative computation, and we do not think this is a good idea, as the effort invested will be recovered only rarely. Besides, total MFA requires us to keep track of multiple independent suspect sets, which can take up a fair amount of space.

5.3.3. Optimisations for stable procedures

One obvious set of candidates for optimisation is the set of deterministic procedures. Clause indexing can prevent shallow backtracking in such procedures; here we are concerned with deep backtracking.

Suppose the call $p(A, \mathbf{X})$ appears in our program, and that procedure p is functional and monotonic. Then we need never send *get_next {X}* messages to p; we can send a *get_next {A}* message to the producer of A instead. Because p is functional, we know that it has at most one solution; because it is monotonic, we know that the current binding of X is part of that solution. If we reject this value of X, p cannot produce another one until the value of A itself is changed. We say that the *surrogate set* of X is *{A}*.

We can generalise this situation to subsets S of *{R1..Rn}* for which the following conditions hold: each call in S is a call to a functional and monotonic procedure, and if any member of S depends on a variable produced by another member of S, then all intermediate links in the dependency chain must also be in S. We call such sets *shortcut sets*. Given a shortcut set S, we can define the surrogate sets of each variable produced in S. If a variable v is produced by $Ri \in S$, then

$$surrogate(v) = \cup \ \mathtt{CV(Rj)} - \cup \ \mathtt{PV(Rj)}$$

where the unions are over all $Rj \in S$ from which Ri is reachable in the dataflow graph (Ri is *reachable* from Rj if Ri depends on Rj or if $i = j$). The first union consists of the variables on whose values v depends (this set of course includes $CV(Ri)$). However, some of these variables may be produced by members of S. Since their producers must be functional and monotonic, we know that we need never backtrack to them directly. We can delete such variables from the surrogate set; the transitivity of the reachability relation ensures that the other inputs of the producers concerned appear in the first union.

As an example, consider the shortcut set containing $p(A, \mathbf{X})$ and $q(B, X, \mathbf{Y})$, where A does not depend on Y. Since the call to p is not reachable from the call to q, the surrogate set of X is *{A}*, as before. On the other hand, the call to q *is* reachable from the call to p. The union of the $CV(Rj)$ for q is *{A, B, X}* while that of the $PV(Rj)$ is *{X, Y}*; the surrogate set of Y is therefore *{A, B}*.

Where should we backtrack to when we receive a *get_next {X, Y}* message ? Backtracking to q is useless, as q cannot change Y by itself. The obvious candidates are the producers of A and B, and to ensure completeness we should choose the rightmost. We can accomplish this by changing only one line. In the algorithm for *get_next* from child, we replace *Vs* on the first line with

```
∪ surrogate(v) for each v ∈ Vs
```

For this to work, every variable must have a surrogate set. To ensure this, we divide *{R1..Rn}* into m partitions, where m-1 partitions contain shortcut sets, and one partition contains the rest of the calls. The surrogates of the variables produced by members of the first m-1 partitions are given by the formula above; if a variable is produced by a call in the last partition, we define its surrogate set to be a singleton containing the variable itself. (Note that a call may not be a member of two shortcut sets; if it were, it would have two conflicting surrogate sets.)

The amount of work saved by the stability optimisation depends on the size of the process tree of the stable call. If that tree is deep, we gain a lot. However, we gain something even if the tree is small; as we will show in section 6.3, surrogate sets do not actually impose any extra overhead on the implementation.

5.4. Applicability

5.4.1. Partially instantiated data structures

So far we have assumed that variables have no visible structure, i.e. that procedures either leave them unbound or instantiate them to a ground term. In real life, though, a term may be constructed through the cooperation of two or more sibling calls, with all but one of those calls producing a partially instantiated data structure. In fact, if its full value is not required by any call, the term may even be left unfinished. An intelligent backtracking algorithm should be able to handle clauses containing such calls, ideally without loss of precision. Being based on a precise mode system, ours can do so; we need only make a few minor changes in the algorithm as we have presented it so far.

The first change is to replace the word "variable" with the phrase "node of the type tree" everywhere in the description of the algorithm. (As this phrase is a bit clumsy, in practice we still talk about variables, but with the understanding that these variables have a relevant internal structure and different parts may be produced in different calls.) The second change is that when we send a *here_is* message for a node of a type tree, we must include in the argument of the message all the tree nodes reachable from that node. As we will show in section 6.3, we can do so without incurring any new overhead. The third change is that unification may now be a recursive process, each invocation taking care of one functor. As the recursive case can be explicitly compiled, we apply the algorithm of section 5.2.3.1 only to the base cases.

In example 5.9 below, the mode declarations tell the compiler that p is the generator of the main functor of X, and if this main functor is $f/2$, then p is the generator (through $p1$) of its left argument as well, while the right argument is supplied by q. The second half of example 5.9 is a Prolog version of the same program. It has each variable shown in bold where its *main functor* gets its value; this occurrence of the variable may or may not instantiate other functors in it. Such an imperfect notation is the best we can do in Prolog; for precision we need type and mode declarations.

The variables $X1$ and $X2$ do not appear textually in the query; the possibility of their existence at run-time is implicit only in the type tree of X. Nevertheless, because they correspond to different nodes in the type tree, our algorithm can treat them as separate variables. In this example, our algorithm would backtrack to $p1$ on the failure of r, as it can see that the failure of r involves only X and $X1$, not $X2$. Most other algorithms (except unification-based ones) suffer from the *function boundary limitation* [Toh86], and therefore cannot see inside the function symbol f.

5.4.2. Circular dataflow graphs

Previous backtracking algorithms required the dataflow graph to be noncircular when viewed at the granularity level of procedure calls. This test excludes all programs with clauses that pass incomplete messages between clients and servers, as their flow of data follows the pattern of figure 5.1. This is a serious limitation: the paradigm of incomplete messages is a very useful one [Shap83b]. All that we *really* need is that no data item should depend on itself, directly or indirectly; as long as this holds, the call graph may be circular. The diagram of figure 5.1 meets this condition. The traditional limitation is therefore not strictly necessary; we can get around it.

To be able to execute all programs with a non-circular data-level dependency graph, we would need to modify our algorithm to handle an unbounded number of virtual procedure calls in each clause. Such a modification is quite achievable. We can attach a depth number to all node names

```
... p(X), q(X), r(X) ...

procedure p(X: top)
% X and X1 are output, _X2 is created as a free variable
mode => X.top, X.left ! X.right;
def
        X = f(X1, _X2), p1(X1)
end p

procedure q(X: top)
% X and X1 are input, X2 is output
mode X.top, X.left => X.right;
def
        X = f(a, X2), q1(X2)
end q

procedure r(X: top)
% X (and implicitly X1 and X2) are input
mode X;
def
        X = f(b, X2), r1(X2)
end r

procedure p1(X1: left)
% X1 is output
mode => X1;
def
        X1 = a
or
        p11(X1)
end r


?- p(X), q(X), r(X).
p(X) :- X = f(X1, _X2), p1(X1).  % _X2 is void
q(X) :- X = f(a, X2), q1(X2).
r(X) :- X = f(b, X2), r1(X2).
p1(X1) :- X1 = a.
p1(X1) :- p11(X1).
```

Example 5.9: partially instantiated data structures.

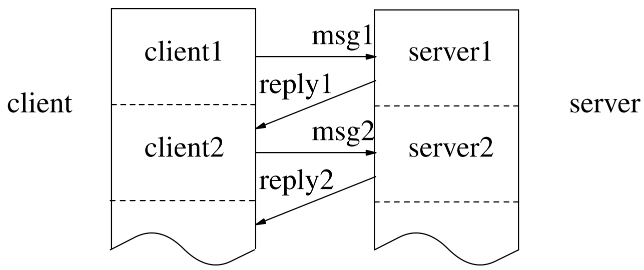

Figure 5.1: client-server type of circularity.

appearing in messages,
specifying all nodes of that name at that and deeper levels.
We would also need to search the whole clause
for potential backtrack points,
since these may now lie to the right of the failed call.
Similarly the calls effected by a *here_is* message
may now lie to the left of the message originator,
so the *here_is* algorithm would also have to search the whole clause
(in fact, a call could even send messages to itself).
The modified algorithms of section 5.2.3 would still terminate,
as the set of depth numbers is well-founded.

The practical problem with this scheme is that the pairing of depth numbers with variables prevents us from using bit vectors to represent sets of variables. This slows down the entire implementation to a degree which we consider unacceptable; we would prefer to allow call-level circularities only when they do not rule out the employment of bit vectors. As the simplest means to this end, we prohibit backtracking between procedures that depend on each other, and to enforce this restriction, we require that all procedures involved in circularities be functional and monotonic. While this may seem a drastic step, it is in fact quite mild when compared with the decisions of the designers of Concurrent Prolog, Parlog, and GHC not to allow any don't-know nondeterminism, and with the later decisions to restrict Concurrent Prolog and GHC to their respective flat versions. In fact, our requirement is not an onerous one, as both clients and servers naturally tend to be stable. (Many systems-type procedures are naturally stable; it is precisely these procedures that also tend to have circular dataflow graphs.)

We purposely defined the surrogate set optimisation technique of section 5.3.3 to be applicable in this situation. The dependency graph of a shortcut set may originally contain loops. However, as the surrogate sets remove any dependence by members of the shortcut set on each other, these loops do not appear in the surrogate dependency graph. Take for example the following query, where *client* and *server* are functional and monotonic:

```
... p(P, D), client(P, S, O), server(S, D), q(O) ...
```

Example 5.10: code with circular dependency graph.

The original dependency graph of this clause appears in figure 5.2. The type tree nodes *S*, *S.element*, *S.readdata*, *S.writedata* and *O* all have the same surrogate set: *{P, D}*. The modified dependency graph is therefore the one in figure 5.3. Since this graph does not contain any loops, our system can handle it without further modification.

Given our decision not to use depth numbers, our requirement that *all* calls in a loop be functional and monotonic is necessary to ensure the termination of our backtracking algorithm. If even one call is either non-functional or non-monotonic, then a *get_next* message from that call may cause a *get_next* message to be sent back to it, since it depends on its own output through the other calls in the loop. This second message may in turn lead to a repetition of the first and hence an infinite loop.

5.4.3. Side-effects

Most research on intelligent backtracking thus far has ignored side-effects. The reasons are:

(1) they greatly complicate the algorithm,

(2) when they occur they reduce the effectiveness of backtrack literal selection,

(3) and even if they do not occur, they increase the overhead of the algorithm.

Nevertheless, we are aware of two efforts in this direction: one for sequential Prolog [Vagh87] and one for restricted AND-parallel Prolog [DeGr87]. These systems suffer from the three problems stated above, but they *are* undeniably useful, insofar as they are the only intelligent backtracking schemes that can execute existing Prolog programs.

We have chosen not to modify our algorithm to handle Prolog-style side-effects. We have three reasons for this decision. First, our algorithm is intended for strongly-moded languages, of which none exist as yet; backward compatibility is therefore not an issue. Second, a side-effect in a stream AND-parallel system manifests itself as the binding of a variable, not as the execution of a goal. Emulation of the chronological backtracking of Prolog does not help in the handling of such side-effects. Third, we believe that such a modification is not really necessary; we can build a practical system without it by separating the parts of the computation with side-effects from the parts

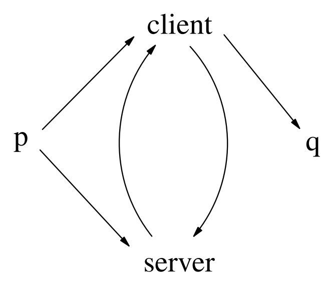

Figure 5.2: original dependency graph.

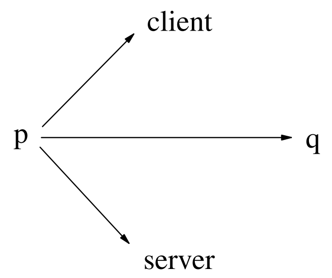

Figure 5.3: modified dependency graph.

that require backtracking.

The mechanism we have presented in chapter 4 can ensure that side-effects will only be executed in the topmost, globally stable layer of the computation. A call in this layer will never be backtracked over: it will simply never get any *get_next*, *get_first* or *here_is* messages. As only a call in this layer can generate values that are monitored by device drivers, this solves our problem. The mechanism operates entirely at compile time, and is quite invisible to the backtracking algorithm: the layers exist only in the data structures of the compiler.

Actually, our mechanism applies only to soft side-effects, i.e. side-effects that do not change the program itself. It cannot handle hard side-effects such as *assert* and *retract*. However, as we have argued in section 1.2, we do not believe this to be a significant restriction: we provide other facilities that programmers can use to duplicate the effects of legitimate uses of *assert* and *retract*: perpetual processes and higher-order variables. The latter may seem to require special support from the backtracking algorithm, but in fact they do not. When a procedure is compiled, the compiler has total access to the interface definitions of all the called procedures, and this includes the values of procedure-valued variables.

Chapter 6

Implementation issues

In this chapter we turn our attention to implementation issues. First we discuss the forward execution algorithm, and consider how existing technology in this area can be adapted to our needs. Then we cover two aspects of the backtracking algorithm: how to enforce the atomicity of backtracking phases and how to implement sets of variables and bit vectors. The last part of this chapter presents a technique whereby fast processors can tolerate the relatively long latency of large, distributed memory systems.

Throughout this chapter, we will assume that our parallel machine has only one physical address space, i.e. every processor can read every memory location. This does not necessarily imply a shared memory architecture: for scalability, memory should be physically distributed among processors. This arrangement is a hybrid between the shared memory and the distributed memory approaches to parallelism; it is the defining characteristic of the DMSV class of architectures in the MIMD taxonomy of [John88] (it stands for "distributed memory, shared variables"). Such architectures can be implemented at various granularities. One can distribute memory as pages, through a modified virtual memory system [Abra85], as words, as in the BBN Butterfly [Cowt85], or as cache lines (which may be words), as in the data diffusion architecture [Warr88a]. Just which of these architectures is the most suitable depends on the granularity of typical remote accesses.

6.1. Forward execution algorithm

The two main thrusts of research in AND-parallel logic programming are independent AND-parallel systems that support don't-know nondeterminism and stream AND-parallel systems that do not support it. Our approach has features of both: it has stream AND-parallelism and it supports don't-know nondeterminism. Accordingly, its implementation calls for a mixture of the implementation techniques of the independent and of the stream AND-parallel systems. (The implementation techniques of attribute grammars [Dera85] may also be applicable.)

By now there are a fair number of papers that deal with the implementation of independent AND-parallel systems [Borg84, Herm86b, Herm87a, Lin87, Lin88c, Lin88b, Lin88a, Herm86a, Herm87b, Baro88, Chas88][1]. Since we have solved the problem of backtracking, the main problem that has so far prevented these systems from supporting stream AND-parallelism, many of the techniques proposed in these papers are applicable to our system (the main exceptions are mechanisms for the detection of goal independence and techniques for preserving compatibility with sequential implementations of Prolog).

The most important lessons we can learn from independent AND-parallel systems lie in the area of memory management. The Warren Abstract Machine [Warr83] is successful in large measure because it does a good job in this area, reducing peak memory requirements and encouraging locality of reference. One can use this locality to optimise performance in virtual memory sys-

[1] The paper [Borg84] also dealt with a form of stream AND-parallelism. However, this feature was quite limited: e.g. it did not allow clauses with circular dataflow graphs.

tems [Ross86]. However, the WAM is not suitable as the basis of a parallel implementation because it implicitly assumes that only one processor is manipulating the stacks at any one time. The independent AND-parallel systems can teach us how to modify the WAM and divide the stacks into segments, with one segment per processor, or how to coordinate multiple processors, each with their own stacks.

The special properties of strongly moded code allow us to simplify the stack organisation: we can omit the pushdown list (PDL). The WAM uses the PDL to store temporary data that its interpreter needs when it unifies two variables. We can dispense with the PDL because strong modes allow us to compile test unification; in an implementation of our system the same temporaries should reside on the local stack.

Another possible change is the deletion of the trail. When a goal fails, its outputs still need to be reset to unbound, but in our system the mode of a call tells us its set of outputs. Using this information instead of a trail is a good tradeoff when goals rarely fail, as it saves the time that would otherwise be spent on pushing addresses onto the stack and it saves the space that would be required to contain those addresses. This last point applies with imperative force for globally stable code: such code never backtracks, and hence it never resets variables. On the other hand, it can run for an unbounded length of time, conceptually even for ever. If we trailed bindings made by globally stable code, even the biggest conceivable trail would overflow sooner or later with useless information. (The WAM can optimise away the trailing of determinate bindings, but it may not always be able to recognise that bindings made in globally stable regions are determinate.)

Despite these points, we believe that implementations of our system should provide a trail stack. The reason is that trails are a very convenient source of information. Modes can only tell us what the outputs of a call are and therefore what locations *might have been* instantiated, but trails tell us what locations actually *were* instantiated. Also, this information is represented in trails in a concentrated form, whereas from modes it can be extracted only with significant overhead. With good arguments on both sides, arguments that apply in disjoint sets of situations, implementations should provide trails but should not make its use compulsory. The compiler should be able to choose to trail or not to trail the bindings made by the primitives invoked by a procedure; this choice should be variable on a procedure-by-procedure basis. The compiler's choice should be stored in the stack frame associated with the call, so that the backtracking algorithm has access to it, and it should also be reflected in the instructions that implement those primitives, to save them from having to make extra memory references.

In the WAM, trail entries only contain the address of a bound variable. In some implementations that allow goals to suspend on variables, trail entries (or at least some of them) also contain the initial contents of the memory location, since this may be a pointer to a list of suspended goals [Boiz86b, Boiz86a]. Our system does not need this refinement: when a goal fails and its bindings are undone, the backward execution sees to it that all potential consumers of those variables are notified, and all actual consumers are killed. When these goals are restarted, they may then again suspend themselves on the same variable, but on backtracking we do not need to reset its value-cell to the old list of suspended goals (since those goals do not exist anymore, it would constitute an error if we did).

Independent AND-parallel systems disallow by definition any direct interaction between user processes. Consequently, they have limited facilities for synchronisation; typically just those needed to establish mutual exclusion on access to data structures that are shared between the various copies of the interpreter. Our system does not make the same restriction, and hence needs much more extensive synchronisation facilities. For ways to implement these facilities, we turn to the implementations of other stream AND-parallel languages [Greg85, Clar85, Cram86, Cram84, Fost87b, Tayl86, Alka88, Klin88a, Mier85, Ichi87, Cram88, Sato87, Fost87c]. As we shall see, many of the techniques described in these papers attack problems that do not arise in strongly moded systems, but their insights can still be useful. For example, they tell us that when instantiating a variable to a structure, we must make sure that the arguments of the structure contain valid data (even if this data is only unbound variables), otherwise readers may race ahead of the writer and read garbage in place of the arguments.

The main issue in stream AND-parallel systems is how to handle concurrent accesses to the same variable. Such accesses pose a synchronisation problem because some of the actions we want to perform on variables are not naturally atomic. When we read the value of a variable, and the variable is already instantiated, there is no problem. If it is not yet instantiated, stream AND-parallel systems generally do one of two things: either they suspend the reading process on the variable, or they allow the reading process to produce a value for it. Both of these alternatives involve writing to the value-cell of the variable. If a second process is allowed to read the contents of that cell between the reading and the writing actions of the first process, then any later writes by the second process will override the write by the first process; this leads to inconsistency. To prevent such problems, most stream AND-parallel systems acquire write locks on memory locations before modifying their contents. This solution has the serious disadvantage that it imposes a relatively large overhead on a very common operation; we would therefore prefer another alternative.

The solution we prefer requires hardware assistance: the assistance of the memory controller to be precise. Its basis is an idea we borrow from research on functional programming: the concept of *I-structure memory* [Arvi83, Arvi80]. In our version of the concept, an I-structure memory controller maintains two status bits per word. These bits state whether the cell has been written into yet, and if not, whether any processes had attempted to read its contents. We denote the three possible states or tags of a word *absent*, *waiting* and *present*. If a word is *absent*, its contents are undefined as yet; if it is *present*, it contains the value that was written into it; if it is *waiting*, then it contains a pointer to the list of processes that had attempted to read the value of the cell. A process should probably be represented in this list by the address of its stack frame.

When a process tries to read a cell whose status is *absent*, the process is suspended, the status of the cell is changed to *waiting* and its value set to point to a list of processes whose only element so far is the reading process[2]. When a process tries to read a cell whose status is *waiting*, it is suspended and its address is put on the list of suspended processes (this list can be implemented by reserving a link field in the stack frame of every process). When a process tries to read a cell whose status is *present*, it gets back the value of stored in the cell as usual. When a process writes to a cell, the memory controller must check first if there are any processes waiting on the value; if yes, it must wake them all up by sending interrupts to the processors holding their stack frames. It should do this after performing the write as usual and changing the status of the location to *present*. The cell will keep this status during the lifetime of the variable that is stored in it. When this lifetime ends, or when the lifetime of its successor begins, the processor will signal its intention to reuse the memory cell to the memory controller, which should then reset the status of the cell to *absent*. Each incarnation of an I-structure memory cell thus serves as a one-shot semaphore.

---

[2] This requires that on every read reference the processor should send to the memory not only the address of the memory cell to be read but also its own address in case the reference suspends. As read and write requests require the same number of bits to be sent, we should be able to handle them with similar bus protocols.

This design is sufficient for our purposes because strong modes eliminate all write-write conflicts (I-structure memories cannot handle such conflicts). Even in a conjunction such as $p(X)$, $p(X)$, where p produces its argument, one of the calls will be assigned the implied mode of p. Modes do not eliminate read-read or read-write conflicts, which manifest themselves as two memory accesses in quick succession. In such cases, the memory controller should delay responding to the second request until it has finished processing the first. When the first request is read access, this is always reasonable, since each read request takes a small and bounded amount of time. If the first request is a write access, the second request may be delayed an unbounded length of time, because the list of processes suspended on a variable may be arbitrarily long. An implementation may choose to allow this: most such lists should be very short. Alternatively, the memory controller may choose to accept read requests for a location before finishing the processing of the write to that location, but after it has set the status of the location to *present*. To allow it to resume its processing of the write request, the memory controller must keep an internal variable that points to the part of the list that points to as-yet-unwoken processes. This means that the number of unfinished write requests at any one time must be less than the number of such internal variables, unless the memory controller goes to the trouble of chaining their suspension lists together. We consider this to require excessive hardware for very limited gains; our preferred option is to disallow any memory requests until previous requests are completely serviced.

The memory controller should implement this policy when necessary by declaring itself to be temporarily unavailable. This declaration should take the form of a temporary negative acknowledgement (TNAK) signal in reply to processors' requests when the memory is busy (or some equivalent mechanism in non-synchronous systems). We believe that processors' bus controllers will need to support such a mechanism anyway, for at least three reasons. One is that in a multiprocessor environment, the refresh cycles of dynamic random access memories (DRAMs) cannot reasonably be implemented by stalling every processor, and memories must be able to respond in some manner to requests from processors even while they are refreshing their capacitors. The second reason is that as the capacity of memory chips continues to grow, the capacitors in dynamic RAM cells are becoming smaller. We can therefore expect them to be damaged by cosmic rays with increasing frequency, increasing the probability that any one reference will be delayed by the need to apply the error-correction algorithm. The third reason is that when a memory is connected (possibly indirectly) to more than one processor, we must be able to delay accepting requests from one processor while servicing a request from another.

The memory controller must be able to support operations other than the operations of I-structure memory. We need such a capability because unlike user data, housekeeping information does not have the single-assignment property; e.g. the word containing the *started* bits of a clause will probably be modified more than once during the lifetime of the clause. To allow us to avoid locking memory locations during such updates, the memory controller should support bit set and bit clear operations, as well as traditional read and write operations, which would still be needed in some contexts (e.g. the extra-logical implementations of device drivers).

The above discussion of course applies only to new designs for logic engines: the memory controllers of conventional multiprocessors do not support the I-structure protocols. On such machines, one has to implement synchronization via more conventional primitives, e.g. test-and-set. Implementations of Ptah should be able to exploit these primitives more efficiently than implementations of other stream AND-parallel languages because strong modes eliminate all write-write conflicts.

Another area in which we can borrow ideas from the papers listed above lies in the implementation of guards. The evaluation of guards in a (pseudo) OR-parallel manner requires some manipulation of environments; we can lift the techniques for doing this from previous proposals. As we can guarantee that the guards of our programs are always safe, this should be a relatively simple matter. The point where we depart from some of these papers and where our implementation can be significantly simpler that the systems proposed in those papers is that our system has no need for atomic commit unification (sometimes also called *tell* unification [Sara88]). This consists of acquiring write locks on the variables that the guard would like to instantiate as an integral part of the process of committing a clause. Atomic commit unification is necessary in some systems to prevent unnecessary failure that might otherwise result from premature commitment. However, our mode system ensures that every incarnation of every memory cell has just one producer, so our system does not need write locks to moderate competition between producers. If programmers wish to allow multiple producers to compete, they have to program the arbitration of the competition; this takes the form of the putative producers each producing a different variable and an arbiter process choosing one of the produced values [Burt88, Nich88].

Our system is capable of expressing deadlock. For example, the expression $p(X, Y)$ & $q(X, Y)$, where p and q each produce one of the variables, is well-moded by the rules of section 2.3, but it will lead to deadlock unless one of the calls can generate its output without waiting for its input. However, we can say that in our system the flow of data will never cause programs to deadlock without the programmer being aware of the possibility. Most procedures should have no call-level circularities, and our mode analysis algorithm can prove many of the rest to be free of data-level circularities. Such code cannot deadlock, as a necessary condition for a deadlock is the existence of a data-level circularity, and a necessary condition of a data-level circularity is a call-level circularity. Only code that has not been proven free of data-level circularities can ever go into deadlock, and our mode analysis will refuse to compile such code in the absence of a directive from the programmer to do so. In fact, the mode analysis algorithm protects programs even against deadlocks caused by faulty control, as it refuses to schedule code from the right side of sequential constructs until it can show that the left side can run to completion.

The above reasoning also tells us that we need not worry about the classical problem of distributed termination, which is the problem of detecting a situation where each member of a set of processes has completed its assignment but is waiting in case it gets messages from other processes [Fran80]. The paper [Shap86] apparently uses the phrase “distributed termination” in a different sense, to denote the detection of the condition that all children of a goal have been completely evaluated. Shapiro argues that even this simplified problem is too hard to solve, and that since sequential AND operators require its solution, such operators should be excluded from parallel logic languages. From our perspective, with our reliance on a global address space, we do not believe that this argument holds water. Shapiro himself presents a solution to the problem in the same paper, a solution that is quite similar to our solution, which is to maintain vectors of *finished* bits in each clause and to set the *finished* bit of a call in its parent when all the *finished* bits in a call are set. With the assistance of the memory controller, using a “set bit and return result” bus operation, this operation can be made quite efficient. That it does have an overhead we do not dispute; but it is a price we are willing to pay, as *finished* bits allow us to implement negation in a natural manner (see section 5.2.3.2). As for Shapiro’s other argument against sequential ANDs, that they tempt programmers into bad habits carried over from von Neumann languages, we believe that no such temptation exists in Ptah. In Concurrent Prolog, programmers may find it easier to put a sequencing construct after a producer than to put read-only annotations on all the consumer instances of variables, which may be in the definitions of other procedures, but this problem does not exist in strongly moded systems. In any case, we intend that sequential ANDs should replace not innocuous parallel AND operators, but guard operators. The implementation of guards is definitely more complex than the implementation of a sequential AND; and the same is true for its semantics.

Our mode system operates at a fine granularity. It can direct very precisely what computations should suspend and what computations may proceed when a given set of data is available.

As we have shown in sections 3.3 and 3.7, this fine granularity allows us to avoid some composition paradoxes that can occur in other systems. However, it may be a disadvantage in that it sometimes allows computations to proceed when there is really nothing for them to do. Consider the call $length(List, N)$ where *List* is supposedly input, and suppose that the order of the clauses were reversed in the definition of *length* (which can be found as example 2.1). When *List* has not yet been instantiated, the modes suspend the unification $List = cons(Head, List1)$, but they allow the execution of the recursive call $length(List1, N1)$, which cannot possibly compute anything useful yet and may lead to an infinite recursion (we are grateful to Jeff Schultz for pointing this out to us). To deal with this problem, we observe that the compiler is free to turn ordinary parallel AND operators into sequential ones whenever the mode analysis algorithm can still find a valid mode for the resulting code. In the case of procedure *length*, this allows us to emit code that waits for *List* to be instantiated before dispatching on its value to the code of one of *length*'s clauses, preventing the waste of computing power. One can view this as compiler assistance to the scheduler.

6.2. Atomic phases

We now turn our attention to the implementation of our backtracking algorithm. The first problem we tackle is the requirement of this algorithm that backward execution phases to be atomic. They must be protected against interference by forward execution phases because both phases modify shared data structures (the *started* and *finished* bits and the $R-$ and $S-sets$). Each backward execution phase must also be protected from interference from other backward execution phases (created for other failures) that may be running at the same time, as our algorithms can handle only one message directed towards any given process at any given time. The implementation of these atomic phases poses no problems for uniprocessors, but it requires special attention on multiprocessors. We do not wish to stop all the processors just because a call failed. Our aim is to stop only the affected processes (this includes sleeping processes woken by messages).

So how do we find out which processes are affected ? We start by locking the process that failed; we then process the *get_next* message according to the algorithms of section 5.2.3, locking all the processes we send messages to. The scheduler may not start children of locked processes, as that would require an update of the *started* bits. The update of the *finished* bits should be also be delayed. As far as interference from other failures is concerned, locked processes will not accept unsolicited messages from other processes; the delivery of such messages should be delayed until the process releases its lock. Unfortunately, two locked processes may try to send messages to each other at the same time; we call such an event a *collision*.

6.2.1. Collision handling

In a naive system a collision would cause the two processes involved to be deadlocked forever. The simplest way to avoid this deadlock is to preempt one of the processes, the parent or the child. (Remember that messages follow the branches of the process tree; two processes sending messages to each other therefore *must* be parent and child.) For simplicity's sake, we choose always to preempt the child: it has only one parent, while the parent may have many children. The problem now is that the child's lock is broken when it is in the middle of a supposedly atomic sequence of updates. The standard transaction processing solution to this problem is the rollback technique. This would require that the backtracking algorithm save the original values of all the memory locations it modifies, and restore these values if and when the lock is broken. It would then handle the message from the parent, and reinvoke the backtracking algorithm for the failure that initiated the child's backward execution phase (if its call has not been killed). We do not consider this to be a viable alternative because it imposes significant overhead on all failures, not just the tiny minority involved in collisions.

A better solution is to restart the child from scratch, and throw away the message it was about to send to the parent. This is a legitimate response to all three types of messages. For two of them, *here_is* and *get_first*, it is what we would do anyway for non-collapsible procedures. For the third (*get_next*) it will probably result in some solutions being returned twice, but this has no effect on the semantics of side-effect free code (and as we discussed in section 5.4.3, our backtracking algorithm is isolated from all side-effects). This technique avoids any slowdown in the common case, at the price of having to do more work when messages collide.

Implementors have two main ways of reducing this penalty. One is to use a (general purpose) goal cache; this is probably worthwhile, though it has its problems, as we discussed in section 5.3.2. The other is to exploit special cases, types of collisions that do not in fact require the child computation to be restarted. We consider three of these special cases to be worth exploiting. (We know of others, but believe that the code needed to make use of them is too complex.) In none of these cases does the child's lock need to be broken; instead it may be merged with the lock of the parent, being released when the parent's lock is released.

The first special case is the collision of a *get_next* message travelling downward and a *here_is* message travelling upward where the arguments of the two messages intersect (the *here_is* message must have been generated by the internal backtracking of the child). In this case the message from the child is exactly what the parent needs; we can send the child's *here_is* message to the parent as the reply to the parent's own message.

The second special case is the dual of the first: the collision of a *here_is* message travelling downward and a *get_next* message travelling upward where the arguments of the two messages intersect. This time the child should not ignore the parent's message; it should process it as usual, as if it were the reply to the child's own message. Note that if the child is not collapsible or if the argument of the *here_is* message contains some of its collapsing variables, then we have to restart the child anyway, and the optimisation gains us nothing (though it does not cost us anything either).

The third special case is the collision of two *here_is* messages. Again we have to restart the child if it is not collapsible or if the argument of the parent's message contains some of the child's collapsing variables. However, if that is not the case, we can process the parent's message as usual, but include in the reply the argument of the child's original message. As an example, consider the procedure below:

```
p(A, B, C, D) :- q(A, C), r(B, C, D).
```

Example 6.1: here_is-here_is collision.

If *q* sent *p* a *here_is {C}* message, *p* would send it along to *r*, and after its *here_is {D}* reply send a *here_is {C, D}* message to its parent. If, at the same time, the parent was trying to send a *here_is {B}* message to *p*, we would process the parent's message as normal. This means that we send another *here_is* message to *r*, this time with the argument being *{B}*. The reply to the parent's message would then be *here_is {D}*, but instead of sending this along as it is, we modify the argument to read *{C, D}*. This is the union of the arguments of the child's original message and of the reply we just computed. (In fact, the union will always be equal to one of its operands. The reason is that a *here_is* message that includes a variable in its argument must also include all variables that depend on it or appear in its *inner* set. One of these relations always holds between any pair of output variables.)

6.2.2. Lock management

So far we have put locks on processes. While this is the logical thing to do, it implies an overhead of several instructions per message. This overhead consumes a significant fraction of the time needed to respond to a message. We would therefore prefer to increase the locks' grain size; the obvious way to do this is to put locks on processors instead.

The problem with this alternative is that when two locked processors try to send messages to each other, we do not know enough to decide which to preempt, or even if we need to preempt one: the originators of the two messages may be completely unrelated. To avoid this quandary, we would like to arrange things so that the two sender processes are not only related but are parent and child sending messages to each other; this way we can apply the techniques of the previous section. To ensure that this condition will always hold, we require the process-to-processor map of each unstable region of the process tree to be non-reentrant and non-recursive (in other words, the process tree should be mapped to a processor tree). Consider figures 6.1 and 6.2, where the boxes represent processors, the circles represent processes, and the arrows represent the structure of the process tree. The configuration of 6.1 is acceptable while the configuration of figure 6.2 is not: the dashed arrow labelled "1" makes the map reentrant while the dashed arrow labelled "2" makes it recursive.

Note that we do not require the *entire* process-to-processor map to be non-reentrant and non-circular. That would be quite an inconvenient constraint: much of the time it would forbid an idle processor's acceptance of work as such action would usually make the map reentrant or recursive. Applying the constraint to each unstable region is sufficient. Processes in stable regions will never send or receive messages, and hence a process in an unstable region will never send messages to a process in another unstable region.

To ensure that the constraint always holds we maintain two bits per unstable region on each processor. One is the lock bit, set when a process in the region is executing its backtracking algorithm.

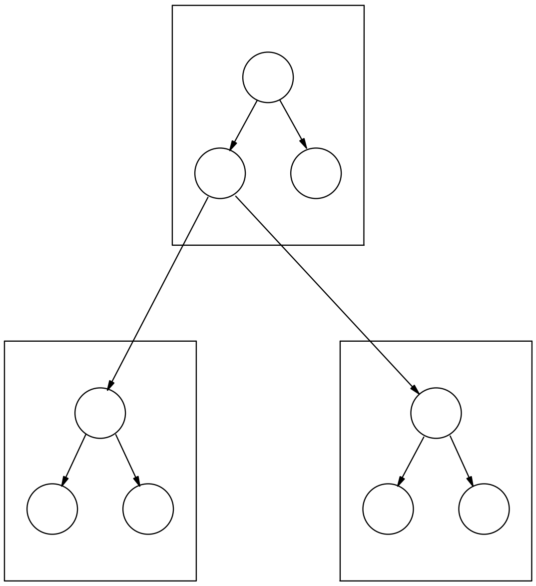

Figure 6.1: a good process map.

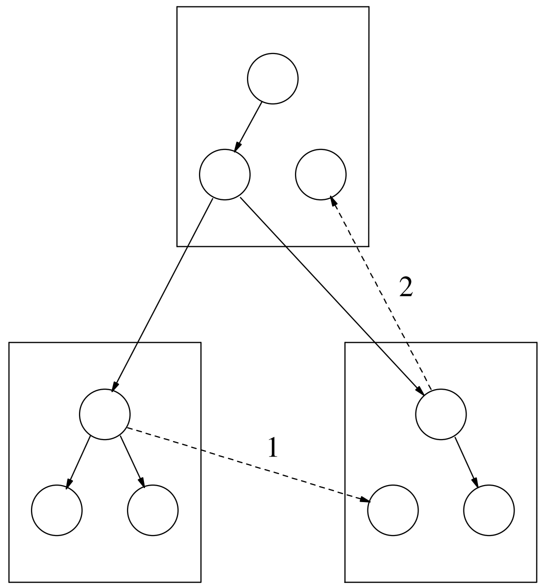

Figure 6.2: a bad process map.

The other (the present bit) is set when the processor steals
its first process from the region[3]
and is reset when that process is killed
(we know that such a process is being killed
when the kill order comes from another processor,
presumably via a hardware interrupt).
While this bit is set, the processor may not steal another process
from the same unstable region, or from other regions inside that region;
that would make the map of the region reentrant or recursive.
It may steal processes belonging to unstable regions
for which the lock is not set
and it may steal processes belonging to stable regions.

This raises another question: how do we know what region a call belongs to ? The answer is: we search upward in the process tree to find a stack frame that is marked as belonging to the root of a region. The important point is that we need to pay the price of this lookup only rarely: in the event of a collision and when stealing a process.

We regard the address of a region's stack frame as its identifying number. The size of an address is probably 32 bits; our data structure therefore needs 2^{32} slots. Fortunately we do not need to store it in its entirety anywhere. It is enough if each processor has space for say 1024 slots. We then map region numbers onto the range 0-1023, possibly based on their last 10 bits: each slot hence represents a potentially large number of regions. A processor may steal processes from on-

[3] We say that a processor *steals* a process when it starts a process whose parent is on another processor. The implementation of such process stealing is much simpler if the stolen process had not been started yet on the other processor (i.e. if there is no process migration). To avoid wasting bus cycles, a processor should steal processes only when it has nothing else to do.

ly one of these regions at a time, as these regions share the same present and lock bits. This in effect constitutes a direct mapped cache with an artificial miss rate of 0% and hence no need for tags.

6.2.3. Other issues related to atomic phases

Our algorithm shares with some other parallel systems the possibility of races between the creation and killing of processes. A process may create another process just before receiving a kill order; to carry out the order, it has to kill its just created child. In the meantime the child may have created yet another process ... and so on ad infinitum.

The probability of such an infinite loop is quite low in systems with fair schedulers (e.g. one based on a bounded depth-first strategy). It can be made even smaller by giving backward execution priority over forward execution: a processor should run in forward mode only when none of its processes are failed or when all its failed processes are waiting for a reply from another processor. However, this is not enough to eliminate infinite loops completely; to be certain, we must add another requirement. This is that a processor should neither steal processes from other processors nor spawn processes on other processors while any of its backward execution phases is waiting for a reply. The first condition guarantees that there will be no infinite loops within processors; the second guarantees that there will be none between processors.

An unrelated point concerns the interaction of the backward execution phases and memory management. When a call fails, we must not deallocate the space occupied by the structures it created until we have killed all processes that consume those structures. If we do, a consumer running on another processor may see inconsistent bindings; it may even change the value-cell of a variable by suspending itself on it [Boiz86a]. On the other hand, we must not allow other processors to restart consumers before we have reset the output variables to the unbound state. If we do, the consumer may see the old value and think it is the new one, leading to inconsistency. The backward execution phase should therefore deallocate and reset the memory claimed by the killed processes after their consumers have been killed (this prevents the first problem) but before the lock is released (this prevents the second problem, as the scheduler will not start their children until then).

6.3. Transposition tables

In our framework, the natural representation of sets of variables is as bit vectors. (Here we are concerned with sets of variables in messages, but we use the same representation in $R-sets$, $S-sets$, *inner* sets, etc.) Each variable is represented by one bit: the bit is 1 if the variable is in the set, and 0 otherwise. Each procedure and each clause has its own map of variables (nodes of type trees) to bit positions. We assume that the map of a clause is an extension of the map of the corresponding procedure; it should assign the same bit positions for the parameters.

This arrangement has the obvious problem that the caller and the callee will probably map the same parameter onto different positions. For example, the procedure p may put its parameter in bit position 0 while the clause in which the call $p(X)$ appears may put X in bit position 1. Altering the bit position of X in the caller does not work in general: there may be other calls to p in the same clause with a different variable as parameter, and we must assign a different position for that variable. We must therefore translate bit vectors across procedure boundaries.

We call our translation mechanism a *transposition table*. It is a bit matrix with as many rows as the number of bits in the vector of the source environment and as many columns as bits in the vector of the destination environment (though it is easiest to represent each row with a machine word). Each bit in the source bit vector specifies a row; to translate the bit vector, we bitwise OR together the rows whose bits are 1 in the bit vector. Most of the time each row has only one bit set: we just move the bit corresponding to a parameter from one position to another (hence the name "transposition"). In some cases a row has no set bits; this happens when a parameter in a call is a constant, as the caller environment then has no variable corresponding to the parameter. In other cases, a row has more than one set bit; this occurs when a parameter is syntactically a structure, containing two or more variables corresponding to the same type tree node.

Conceptually, we need four transposition tables per call: We need to convert bit vectors of both input and output arguments in two directions: upward (to translate the callee's bit vectors to the caller's bit vectors) and downward (to do the reverse translation). The algorithms for *here_is* use the upward table for outputs and the downward table for inputs; the algorithms for *get_next* use the upward table for inputs and the downward table for outputs (this last table is used by the *get_first* algorithm as well).

Transposition tables also help us take care of some other details. As we discussed in section 5.4.1, we must make sure that no *here_is* message refers to a node of a type tree without referring to all nodes reachable from that node. (Reachability here refers to the type tree, not to the dataflow graph.) To get this effect, we need only modify the upward table for outputs so that a row that has a bit set for a node also has bits set for all nodes reachable from it. We could also modify the other table consulted by *here_is*, the one for downward translation of inputs, but this is not strictly necessary, as this modification will merely duplicate the work done by the first table. On the other hand, one of the tables used by *get_next* (the one for upward translation of inputs) can implement the surrogate sets of section 5.3.3. Instead of mapping each formal parameter of the called procedure onto the variables (if any) occurring in the actual parameter, we can just map it onto the union of the corresponding surrogate sets. On a related note, a table parallel to the upward table for outputs is just what we need to compute the union of *inner* sets in the algorithm for *here_is* from child *Ri*.

As an example, consider the call $p(cons(A, cons(B, Cs)), nil, D)$ on the procedure $p(Xs, Ys, Z)$, where the first two arguments of p are lists and the third argument and the elements of the lists are primitive types such as integers. The type tree of *Xs* appears in figure 6.3 while the type tree of the corresponding actual parameter appears in figure 6.4 (it is derived directly from the structure of the actual parameter). The two type trees are compatible, though not identical. The type tree of the actual parameter has two anonymous nodes corresponding to the two occurrences of the constant functor *cons*. The two anonymous nodes and the *Cs* node of the actual parameter correspond to the *Xs* node of the formal parameter; the A, B and $Cs.\,element$ nodes correspond to the $Xs.\,element$ node (in the figures and tables the word "element" is abbreviated to "elt" to fit within small boxes).

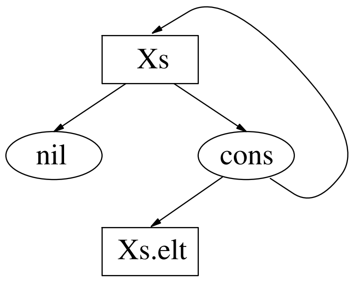

Figure 6.3: type tree of formal parameter.

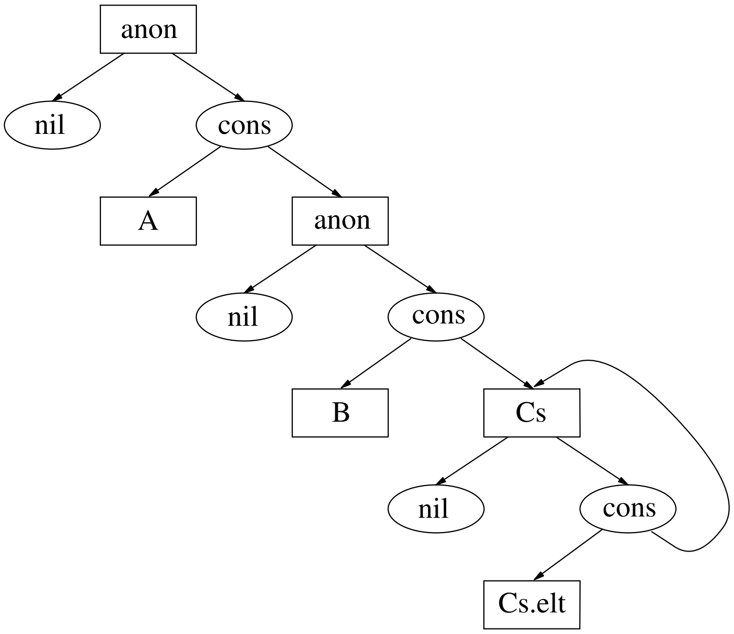

Figure 6.4: type tree of actual parameter.

We show possible node-name-to-bit-position maps of the clause and of the procedure head in table 6.1[4]. The four transposition tables based on these maps are in tables 6.2-6.5. The table for upward translation of inputs (table 6.2) assumes that A is the output of a stable procedure whose only input is E; otherwise that table would simply be the matrix transpose of the table for the downward translation of inputs, table 6.4.

maps for clause and head		
bit no.	clause node	head node
0	A	Xs
1	B	Xs.elt
2	Cs	Ys
3	Cs.elt	Ys.elt
4	D	Z
5	E	-
...	...	-

Table 6.1: maps of clause and head.

[4] In practice there would be some other nodes before A in the map of the clause; we chose this mapping for simplicity. Note also that A is in no way paired with Xs; they simply happen to be assigned the same bit positions.

upward inputs		clause: name and posn						
head		A	B	Cs	Cs.elt	D	E	...
name	posn	0	1	2	3	4	5	...
Xs	0	0	0	1	0	0	0	...
Xs.elt	1	0	1	0	1	0	1	...
Ys	2	0	0	0	0	0	0	...
Ys.elt	3	0	0	0	0	0	0	...

upward outputs		clause: name and posn						
head		A	B	Cs	Cs.elt	D	E	...
name	posn	0	1	2	3	4	5	...
Z	4	0	0	0	0	1	0	0

downward inputs		head: name and posn			
clause		Xs	Xs.elt	Ys	Ys.elt
name	posn	0	1	2	3
A	0	0	1	0	0
B	1	0	1	0	0
Cs	2	1	0	0	0
Cs.elt	3	0	1	0	0
D	4	0	0	0	0
E	5	0	0	0	0
...	...	0	0	0	0

downward outputs		head: name and posn
clause		Z
name	posn	4
A	0	0
B	1	0
Cs	2	0
Cs.elt	3	0
D	4	1
E	5	0
...	...	0

Tables 6.2-6.5: transposition tables.

Upward translation tables need as many rows as the call has inputs/outputs, and as many columns as the clause has variables; in downward translation the roles of rows and columns are reversed. Based on the not-altogether-comparable data of [Mats85], we estimate that the average procedure has two or three arguments about evenly divided between inputs and outputs, while the average clause has about five variables. The average number of type tree nodes per variable is most likely between one and two. From this we can calculate that the average size of an upward transposition table is one to three words while that of a downward table is five to ten words. The average call would therefore need ten to twenty-five words for transposition tables. This is clearly excessive. We have therefore developed techniques to reduce the size of the tables and for some calls to eliminate them altogether.

6.3.1. Reducing the size of transposition tables

The simplest way to reduce the amount of memory needed is to put several tables into a chunk of memory side by side; this is possible since the number of columns in an average table is substantially less than 32, the typical wordsize. This technique can reduce storage requirements by a factor of two to four at the expense of necessitating a shift and a mask after the rows had been ORed together. Note that the neighbouring tables do not have to belong to the same call or even to the same clause.

Our second technique exploits the fact that our message-handling algorithms work correctly even when extraneous variables are included in the arguments of the messages. We can therefore map two or more input variables (type tree nodes) or two or more output variables (type tree nodes) of a procedure onto the same bit position; when this position contains a 1, we consider all those variables to be included in the argument. For inputs, this merging takes place in the called procedure; for outputs, it takes place in the calling clause.

```
?- p(A), q(B), r(A, B, C, D), s(C), t(D).
```

Example 6.2: merging bits.

For this example, assume that we want to merge both the inputs and the outputs of *r*. Then procedure *r* would map *A* and *B* onto bit position 0, *C* onto bit position 1, and *D* onto bit position 2, while the above goal clause would map *A* onto bit position 0, *B* onto bit position 1, and *C* and *D* onto bit position 2. The effect of the merging of the inputs is that *r* now has one less argument, and each clause of *r* has one less variable, as far as the sizes of transposition tables are concerned. The effect of the merging of the outputs is that the goal clause has one less variable, again reducing the size of the tables. Other clauses calling *r* may or may not merge the bits of the outputs, as this is a call level decision; the question of whether to merge the bits of input arguments is a procedure level decision.

This technique saves space and run-time overhead (it takes time to OR those rows together). The price it pays for this is loss of precision. The *get_next* algorithm may not skip as many choice points, and the *get_first* and *here_is* algorithms may kill more innocent computations than they would otherwise. In example 6.2, we would always backtrack to *q* following the failure of *r*, even if only *A* was involved in the failure; similarly we would restart both *s* and *t* even if the *here_is* message from *r* specifies only one of *C* and *D*.

The simplest use of this technique is to allocate one bit to all the type tree nodes of a variable. Most of the time this can be done without loss of precision[5]; the number of type tree nodes per variable thus ceases to have any effect on the size of the tables. To go one step further, we may merge some (but not all) of the input and/or output variables of a procedure: we may be forced to do this if a clause has more variables than there are bits in a machine word[6]. The last

[5] As an example of an exception, consider code that first constructs a list of unbound variables and then instantiates those variables in a separate call. If the code then tests the length of the list and finds it incorrect, our algorithm will try to give different values to the elements of the list, when in fact it should go back and construct another list skeleton.

[6] Merging can reduce the number of bits needed to what is available only if the number of calls in the clause is itself less than or equal to the number of bits in a machine word. If it is greater, we need to divide the clause into manageable portions using the *definition* and *folding* program transformation operations [Tama84], but this should be necessary only rarely.

step is the merging together of all inputs and all outputs. If carried out consistently, this step has the further advantage that the relevant transposition table will always have just one row; the underlying implementation (whether in software or firmware) can therefore be simplified somewhat. Note that in this limiting case, where every procedure merges all its inputs and all its outputs, our overhead is virtually identical to the overhead of [Lin86b, Woo86, Cone87]. The difference is that unlike the algorithms in those papers, our algorithm allows a wide range of overhead/precision tradeoffs, and these tradeoffs can be controlled on a procedure-by-procedure and call-by-call basis.

6.3.2. Eliminating transposition tables

We now consider the circumstances under which transposition tables can be eliminated altogether. The most obvious one occurs when a procedure has no inputs or no outputs; in such cases we need neither of the two corresponding tables (upward and downward).

The second opportunity involves stable procedures. They do not need any downward transposition tables for outputs; the surrogate set optimisation will redirect any *get_next* messages sent to them to the producers of their inputs. We can also exploit the fact that stable procedures never send *get_next* messages to their parents. They will not do so on their own initiative (type I backtracking) because they are total; they will not do so on behalf of their consumers (types II and III backtracking) because of surrogate sets. (The terms "type I/II/III backtracking" were introduced by [Chan85a].) Calls to stable procedures therefore do not need to provide upward transposition tables for inputs, either.

The most complex situation arises for non-collapsible procedures (including procedures that are naturally collapsible but which we call non-collapsible because we do not need the collapsibility optimisation for them). Any *here_is* message that these procedures receive from their parents causes them to start again from scratch. We can therefore safely give them a garbage vector as the argument of *here_is*, as they will ignore the argument. Hence such procedures need no downward transposition table for inputs. A somewhat related case is that of a call in a non-collapsible procedure whose effective inputs (union of the surrogate sets of the inputs) are all produced by the head of the clause. If this call is to a stable procedure, we can dispense with its upward table for outputs; it would be needed only when the parameters of the call change, and in that case the procedure is restarted from scratch.

These optimisations together should eliminate a large percentage of all transposition tables. The first applies mainly to tests, including test unification. The second also applies to a substantial number of calls: assignment unification, most uses of append, etc. The third applies to most calls, as many procedures are naturally not collapsible and we do not need the optimisation for most of the rest. When put together these optimisations can eliminate *all* transposition tables in the special case of globally stable code (see section 4.5.1). Such codes behave like programs written in compulsory committed-choice languages: they never backtrack. They do not need upward translation of inputs or downward translation of outputs because their procedures are all stable; they do not need downward translation of inputs or upward translation of outputs because none of their procedures need the collapsibility optimisation.

6.4. Tolerance of latency

One of the driving factors of computer architecture at the present level of technology is the fact that processors are faster than memories. We expect the ratio between memory and processor cycle times to increase, especially in parallel logic programming machines, for the following reasons. First, off-chip delays (driven by packaging technology) will stay relatively constant, while on-chip delays (driven by process technology) will fall. Second, a global address space distributed among multiple local memories, which we expect to be an increasingly popular design, implies occasional reference to remote memories; this again is inherently slow. Third, parallel logic programming systems tend to require large amounts of main memory, and for the system to be economically viable these memories must be cheap and therefore slow. Fourth, the complexity of memory controllers will probably increase. Not only do they need to do error-detection and correction, they will also need to support the I-structure protocols, and in some systems the data diffusion protocols as well. Fifth, translations from virtual to physical addresses take time. The last three reasons combine to make memory access relatively slow even for a local access; this is important because logic programs tend to be very memory access intensive.

The traditional way to handle the imbalance between processor and memory speeds is to interpose a cache between the processor and memory. We feel that this is not a full solution for some applications, including ours, because the first and third arguments above apply to caches as well as main memories, although with reduced force. Machines intended for number crunching (e.g. Cray [Russ78], Multiflow [Colw87]) are usually built without caches, because this application area, like logic programming, tends toward programs with poor locality. Another reasons why we believe caches to be less than fully effective for parallel logic programming systems is that such systems tend to have a low ratio between reads and writes.

We believe that another approach, based on interleaved memory, will be more effective. We allow the processor to tolerate memory latency by switching to other tasks while waiting for outstanding memory references. We call this a micro context switch. It has been tried before: the idea is at least 25 years old (e.g. CDC 6600s used it for their peripheral processors [Thor70]). It has not caught on even for cacheless designs because when applied to programs written in conventional languages it cannot reduce the execution times of single programs; it can only increase throughput for multiprogrammed systems. We believe our contribution to be the recognition that programs written in parallel logic programming languages should have enough low-level parallelism among their lightweight processes to make reductions possible even in the run-time of single programs.

To be able to support a micro context switch, the processor must have immediate access to the state of more than one process, i.e. it must have multiple registers sets, including program counters, status words etc. We can take advantage of this arrangement to speed up parameter passing between processes: we can copy data from one register set to another without going through memory. Thus the set of process states resembles the RISC concept of overlapped windows [Patt85], as well as the goal cache of the Concurrent Prolog machine described in [Alka88].

The minimum number of states a processor should have is the ratio between the average memory access time and the processor cycle time. However we favor designs with a few more states, for two reasons. The first is that it helps the processor tolerate strings of above-average-latency memory accesses. The second is that some states may be in the middle of a macro-level context switch, caused e.g. by page faults or by processes suspending themselves waiting for variables to become bound. We wish to ensure that the remaining active states can exploit the full memory bandwidth. However, “can exploit” does not mean “does exploit”, and not all microcycles will access memory. In many cases we can use these idle bus cycles to do the swapouts and swapins required by macro context switches, hiding their overhead via a dribble-back/dribble-forward mechanism.

Another benefit of having a larger number of states is that it enables us to do a micro context switch on every microcycle, rotating the processor among the active states. This has the advantage that we can pipeline instruction fetch, decode, operand read, execute, and operand write without interference between the stages, because the instructions in the pipeline will belong to different processes. The macro- and micro-instructions of each process therefore execute sequentially; this eliminates pipeline breaks due to branches. This is particularly important given the fact that logic programming systems have a very high branching rate; consider the tight loop required to implement dereferencing.

In a processor designed along these lines a cache may not increase execution speed as long as the program contains sufficient parallelism, because caches exist to reduce latency, and our processors can tolerate latency, supported as they are by a fast local bus and interleaved memory. Another reason why we may wish to leave out caches is that logic programming systems typically have less locality than conventional systems; the introduction of multiple states of course aggravates the situation. The main advantage of caches is that by reducing effective memory latency they reduce the micro-multiprogramming level required to keep the processor busy. They therefore allow us to exploit the full power of the processor even for programs that have a lower level of parallelism. Another advantage is that they reduce contention for the local memory banks (in data diffusion systems the caches must contend with the bus-watching memory coherence protocols for access to the local memory banks). The upshot of all these factors is that the decision of whether to include caches in a design is much more finely balanced than usual.

We believe that logic programming machines need microprogramming because non-microcoded implementations lead to unacceptably large code size (the papers [Borr87] and [Mill86] report the expansion factor to be between 7 to 14). As a consequence, a single processor with N states can be implemented in much less silicon area than N similar processors each with one state, mainly because the latter configuration requires a separate microstore for each processor (and a cache if we want to support caches). We expect that a single N state processor will get more work done than M one state processors that together have the same total area, because the one state processors will have to be idle much of the time through inability to tolerate memory latency. If one attempts to reduce the effective memory latency by providing each processor with a general data cache, or by giving each processor the ability to execute instructions out of sequence, as is done in Aquarius [Desp87], then the performance of each processor will increase but a smaller number of them will fit on the same silicon area, so that the aggregate performance will remain about the same.

As another line of argument supporting our claim, consider that in logic programming systems one can usually do only a very small amount of work between memory references. A single N-state processor should be able to make profitable use of almost all of the available memory bandwidth. The number of single-state processors that would be required to obtain the same bus utilisation would occupy much greater area, because each single-state processor contains many components that will be idle most of the time. By contrast, the only parts of a multi-state processor that are idle most of the time are the states themselves, and their area is small compared to the area of a microstore or a cache.

Chapter 7

Conclusions

Our goal, the design of a parallel logic programming system for general purpose computing on the Unix model, is shared by a number of other research projects. The main feature that distinguishes our approach from the approaches of these other projects is our unwillingness to compromise clean semantics in the name of efficiency. We of course know that non-logical features can in fact gain efficiency (although often they do not), but we believe that the gains are not worth it. The price we have to pay for these gains is a substantial increase in conceptual complexity, and this price is one that the typical programmer cannot afford to pay.

The most important consequence of this attitude is our decision to support both stream AND-parallelism and don't-know nondeterminism in a single unified framework. We have to support stream AND-parallelism, as it is essential to our chosen application area: it is the natural representation of multiple cooperating processes and as such it is indispensable in the construction of parallel logic operating systems. Don't-know nondeterminism is required by the standard declarative semantics of logic programs. However, in the absence of compile-time dataflow information it is impossible to integrate the two in a regular and reasonably efficient manner without losing the operational semantics that makes stream AND-parallelism desirable in the first place.

The main contribution of this thesis is a strong mode system that gives us exactly the information we need to allow us to perform this integration. This mode system is desirable in its own right because it enforces good programming practices and allows us to detect many errors at compile time. The unique feature of our approach is that by relying on type information, we can look inside the possible structures of terms and apply our modes to the smallest possible unit. This precision makes our mode system flexible enough to express the paradigms required for the construction of logic operating systems, e.g. the paradigm of incomplete messages.

The implementation of a logic operating system in a don't-know nondeterministic language poses a unique problem: we must ensure that the operating system will never confuse the hardware it controls by giving it too many orders, vacillating orders, or no orders at all. We have shown how to use stability declarations to guarantee that an operating system written in a don't-know nondeterministic language will do none of these things. Not only does this allow programmers to write user programs in a language with clean semantics, but it also allows them to write much of the operating system in such a language.

The introduction of don't-know nondeterminism into a stream AND-parallel language also poses an implementation problem. Although previous backtracking algorithms applied only to AND-sequential and independent AND-parallel systems, strong modes and the stability mechanism allow us to adapt them to a stream AND-parallel environment, including programs with circular dataflow graphs. Our backtracking algorithm represents a considerable improvement over its predecessors in other respects as well. It does not lose efficiency when procedures generate partially instantiated data structures, and it can be more precise in its selection of the backtrack literal, as it overcomes the clause and function boundary limitations and does a limited form of match failure analysis.

Compared to existing concurrent logic languages, implementations of our system will incur some extra overheads, these overheads going to the maintenance of choice points and other housekeeping information that goes with don't-know nondeterminism and the handling of negation. However, these overheads can be avoided completely for programs that do not require don't-know nondeterminism or negation (i.e. programs that can be expressed in the committed-choice subset of Ptah), and we have shown that even in code that does need backtracking, the part of this overhead can be traded off against the increased precision of our backtracking algorithm. On the other hand, the simpler semantics of our language allows us to implement synchronisation more efficiently, especially with hardware support, and the mode system allows us to exploit indexing more extensively. Also, we expect that implementations of our system, by virtue of borrowing the stack organisation of independent AND-parallel systems, will have better memory referencing behaviour than the present heap-based implementations of other concurrent logic languages; this can have a significant effect in a virtual memory environment. Taken together, these factors lead us to believe that an implementation of our system can be efficient enough to serve as the basis of a practical parallel logic engine.

Appendix A

Glossary

break_cocoon	An operation to convert a universal object back to its original type, checking that the type expected by the code in question is compatible with the type stored in the cocoon.
cocoon	A device that wraps up an object and its type, manipulated by the primitive operations *make_cocoon*, *break_cocoon* and *probe_cocoon*.
collapsible	A procedure is *collapsible* by a set of its input variables if for each set of values of those variables, all the solutions of the procedure are produced by a single clause.
connected	Two variables are *connected* if a call depends on both of them.
CV(Rk)	The set of variables whose values are consumed by the call Rk.
depends on	A call Rk *directly depends on* variable $v1$ if $v1 \in CV(Rk)$. A variable $v2$ directly depends on variable $v1$ if $v2 \in PV(Rk)$ and Rk directly depends on $v1$. The *depends on* relation is the transitive closure of these notions.
extended type forest	The set of type trees of the variables that occur in an expression, including the arguments of the procedure in question.
finished	There is a *finished* bit for every call in every active clause. It is set if the call has been completely and successfully evaluated (the *finished* bit is not set on failure).
functional	A procedure in a given mode is *functional* if it never (i.e. for no set of input bindings) has more than one solution.
get_next	A *get_next Vs* message tells the recipient process to generate new bindings for at least one variable in the set *Vs*.
get_first	A *get_first Vs* message tells its recipient to start generating the bindings of the variables in *Vs* all over again.
hard side-effect	A side-effect that changes the program, e.g. *assert* and *retract* in Prolog.
here_is	A *here_is Vs* message tells its recipient that all the variables in the set *Vs* have had their values retracted, and that they should soon get new bindings.
implied mode	A mode created from some other mode by changing some tags to INBOUND.
inner set	With each variable $v1$ (produced by say Ri) we associate a set $inner(v1)$ so that $v2 \in inner(v1)$ if $v1$ and $v2$ are connected, $v2$ does not depend on $v1$, and the producer of $v2$ is Rj for some $j > i$.
make_cocoon	An operation to convert an object of any given type into another object of universal type by wrapping it up in a cocoon together with its type.
monotonic	A procedure in a given mode is *monotonic* if it never retracts a binding it makes to any part of any of its output variables unless a consumer compels it to do so by sending it a *get_next* message.

principal mode	A mode not implied by any other mode.
probe_cocoon	An operation to return the type stored in the cocoon of an object of universal type.
PV(Rk)	The set of variables whose values are produced by the call *Rk*.
R-set	There is an $R-set$ (failure related set) for every call C in every active clause. This set contains the variables (except those produced by C) that were involved in the failure that caused a backtrack to C.
S-set	There is an $S-set$ (suspect set) for every process. It contains the input variables whose values have contributed to the failures of the clauses of the process, and hence to the possible failure of the process itself.
shortcut set	A *shortcut set* is a subset S of the calls in a clause for which the following conditions hold: each call in S calls a functional and monotonic procedure, and if any member of S depends on a variable produced by another member of S, then all intermediate links in the dependency chain must also be in S.
soft side-effect	A side-effect that does not change the program.
stable	A procedure in a given mode is *stable* if it is functional, monotonic and total.
started	There is a *started* bit for every call in every active clause. It is set if the execution of the call has started.
surrogate set	The *surrogate set* of a variable v is the set of variables whose values determine the value of v. Surrogate sets are defined with respect to a given shortcut set.
suspect set	See S-set.
total	A procedure in a given mode is *total* if it never fails without first producing a solution.
type forest	The set of type trees of the arguments of a procedure.
univ	The universal type; the type to which objects of all types can be converted via the *make_cocoon* operation.

References

[1%Q83] 'The programming language Ada reference manual', Springer-Verlag, New York, Lecture Notes in Computer Science, 1983, 331 pages.

[1%Q86] 'Quintus Prolog user's guide', Report, Quintus Inc., Sunnyvale, California, April 1986.

[Abra85] D.A. Abramson and J.L. Keedy, 'Implementing a large virtual memory in a distributed computing system', *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences, Volume 2*, Honolulu, Hawaii, January 1985, pages 515-522.

[Alka88] Leon Alkalaj and Ehud Shapiro, 'An architectural model for a Flat Concurrent Prolog processor', *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, pages 1277-1297.

[Arvi83] Arvind and R.A. Iannucci, 'A critique of multiprocessing von Neumann style', *Proceedings of the Tenth Annual International Symposium on Computer Architecture*, Stockholm, Sweden, June 1983, pages 426-436, published as SIGARCH Computer Architecture News 11:3.

[Arvi80] Arvind and Robert E. Thomas, 'I-structures: an efficient data type for functional programming', Report MIT/LCS/TM-178, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1980, 17 pages.

[Av-R84] E. Av-Ron, 'Top-down diagnosis of Prolog programs', M.Sc. thesis, Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel, 1984.

[Azzo88] H. Azzoune, 'Type inference in Prolog', *Proceedings of the Ninth Conference on Automated Deduction*, Argonne, Illinois, May 1988, pages 258-277, published as Lecture Notes in Computer Science 310 by Springer-Verlag.

[Bahg88a] Reem Bahgat, 'Towards an integrated don't-know stream-and-parallel logic language', Research Report DOC 88/1, Department of Computing, Imperial College of Science and Technology, University of London, England, February 1988, 59 pages.

[Bahg88b] Reem Bahgat and Steve Gregory, 'Pandora: non-deterministic parallel logic programming', Research Report, Department of Computing, Imperial College of Science and Technology, University of London, England, November 1988.

[Bark87] Jonas Barklund and Hakan Millroth, 'Hash tables in logic programming', *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pages 411-427.

[Baro88] Uri Baron, Jacques Chassin and Jean-Claude Syre, 'The parallel ECRC Prolog system PEPSys: an overview and evaluation results', *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988, pages 841-850.

[Beer87] Catriel Beeri and Raghu Ramakrishnan, ‘On the power of magic’, *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, San Diego, California, March 1987, pages 269-283.

[Bens72] A. Bensoussan, C.T. Clingen and R.C. Daley, ‘The Multics virtual memory: concepts and design’, *Communications of the ACM*, 15:5, May 1972, pages 308-318.

[Boiz86a] P. Boizumault, ‘General model for implementing DIF and FREEZE’, *Proceedings of the Third International Conference on Logic Programming*, London, England, July 1986, pages 585-592, published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Boiz86b] Patrice Boizumault, ‘A classical implementation for Prolog-II’, *Proceedings of the First European Symposium on Programming*, Saarbrucken, Germany, March 1986, pages 262-273, published as Lecture Notes in Computer Science 213 by Springer-Verlag.

[Borg84] P. Borgwardt, ‘Parallel Prolog using stack segments on shared-memory multiprocessors’, *Proceedings of the First IEEE Symposium on Logic Programming*, Atlantic City, New Jersey, February 1984, pages 2-11.

[Borg86] Peter Borgwardt and Doris Rea, ‘Distributed semi-intelligent backtracking for a stack-based AND-parallel Prolog’, *Proceedings of the Third IEEE Symposium on Logic Programming*, Salt Lake City, Utah, September 1986, pages 211-222.

[Borr87] Gaetano Borriello, Andrew R. Cherenson, Peter B. Danzig and Michael N. Nelson, ‘RISCs vs. CISCs for Prolog: a case study’, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, October 1987, pages 136-145.

[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, Scotland, November 1982.

[Bruy80] Maurice Bruynooghe, ‘Analysis of dependencies to improve the behaviour of logic programs’, *Proceedings of the Fifth Conference on Automated Deduction*, Les Arcs, France, 1980, pages 293-305, published as Lecture Notes in Computer Science 87 by Springer-Verlag.

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

[Bruy88] Maurice Bruynooghe and Gerda Jenssens, ‘An instance of abstract interpretation integrating type and mode inferencing (extended abstract)’, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, pages 669-683.

[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.

[Burt88] A. Burt and G.A. Ringwood, 'The binding conflict problem in concurrent logic languages', manuscript, Imperial College of Science and Technology, London, February 1988.

[Butl88] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek and Rick Stevens, 'Scheduling OR-parallelism: an Argonne perspective', *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, pages 1590-1605.

[Byrd80] Lawrence Byrd, 'Understanding the control flow of Prolog programs', *Proceedings of the 1980 Logic Programming Workshop*, Debrecen, Hungary, July 1980, pages 127-138.

[Card83] Luca Cardelli, 'Stream input/output', *Polymorphism*, 1:3, December 1983.

[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.

[Card85] L. Cardelli, 'Amber', *Proceedings of the Thirteenth Spring School of the LITP: Combinators and functional programming languages*, Val d'Ajol, France, May 1985, pages 21-47, published as Lecture Notes in Computer Science 242 by Springer-Verlag.

[Chan85a] Jung-Herng Chang, Alvin M. Despain and Doug DeGroot, 'AND-parallelism of logic programs based on a static data dependency analysis', *Digest of Papers of the Thirtieth IEEE Computer Society International Conference (CompCon 85 Spring): Technological Leverage: a Competitive Necessity*, San Francisco, California, February 1985, pages 218-226.

[Chan85b] J.-H. Chang, 'High performance execution of Prolog programs based on a static data dependency analysis', Ph.D. thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, October 1985.

[Chas88] J. Chassin and P. Robert, 'An abstract machine to implement efficiently OR-AND parallel Prolog', *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, page appendix.

[Chik84] Takashi Chikayama, 'Unique features of ESP', *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1984, pages 292-298.

[Chik87] Takashi Chikayama and Yasunori Kimura, 'Multiple reference management in flat GHC', *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pages 276-293.

[Chik88] Takashi Chikayama, Hiroyuli Sato and Toshihiko Miyazaki, 'Overview of the parallel inference machine operating system (PIMOS)', *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988, pages 230-251.

[Chow86] F. Chow, M. Himelstein, E. Killian and L. Weber, 'Engineering a RISC compiler system', *Digest of Papers of the Thirtyfirst IEEE Computer Society International Conference (CompCon 86 Spring): Intellectual Leverage*, San Francisco, California, February 1986, pages 132-137.

[Clar87a] Keith L. Clark, ‘Parlog: the language and its applications’, *Proceedings of the Conference on Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, Eindhoven, Netherlands, June 1987, pages 30-53, published as Lecture Notes in Computer Science 259 by Springer-Verlag.

[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.

[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.

[Clar87b] Keith L. Clark and Steve Gregory, ‘Parlog and Prolog united’, *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pages 927-961.

[Clar79] K.L. Clark and F.G. McCabe, ‘The control facilities of IC-Prolog’, in *Expert systems in the microelectronic age*, Donald Michie (ed.), Edinburgh University Press, 1979, pages 122-149.

[Clar77] Keith L. Clark and Sten-Ake Tarnlund, ‘A first order theory of data and programs’, *Proceedings of the Seventh IFIP Congress (Information Processing 77)*, Toronto, Ontario, August 1977, pages 939-944.

[Clar83] Keith Clark and Steve Gregory, ‘Parlog: a parallel logic programming language’, Research Report DOC 83/5, Department of Computing, Imperial College of Science and Technology, University of London, England, May 1983, 35 pages.

[Clar84] Keith Clark and Steve Gregory, ‘Notes on systems programming in Parlog’, *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1984, pages 299-306.

[Clar85] Keith Clark and Steve Gregory, ‘Notes on the implementation of parlog’, *Journal of Logic Programming*, 2:1, April 1985, pages 17-42.

[Clea88] John Cleary, Brian Unger and Xining Li, ‘A distributed AND-parallel backtracking algorithm using virtual time’, *Proceedings of the 1988 Conference on Distributed Simulation (part of Multi ’88)*, San Diego, California, February 1988.

[Codi86] Michael Codish and Ehud Shapiro, ‘Compiling OR-parallelism into AND-parallelism’, *Proceedings of the Third International Conference on Logic Programming*, London, England, July 1986, pages 283-297, published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Cohe84] Shimon Cohen, ‘Multi-version structures in Prolog’, *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1984, pages 265-274.

[Colm82] A. Colmerauer, ‘Prolog II: reference manual and theoretical model’, Internal report, Groupe Intelligence Artificielle, Universite Aix-Marseille, France, 1982.

[Colw87] Robert P. Colwell, Robert Nix, John J. O’Donnell, David B. Papworth and Paul K. Rodman, ‘A VLIW architecture for a trace scheduling compiler’, *Proceedings of the Second International Conference on Architectural Support* *for Programming Languages and Operating Systems*, Palo Alto, California, October 1987, pages 180-192.

[Cone83] John S. Conery, 'The AND/OR process model for parallel interpretation of logic programs', Ph.D. thesis, Department of Information and Computer Science, University of California, Irvine, June 1983.

[Cone87] John S. Conery, 'Implementing backward execution in non-deterministic AND-parallel systems', *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pages 633-653.

[Cone85] John S. Conery and Dennis F. Kibler, 'AND parallelism and nondeterminism in logic programs', *New Generation Computing*, 3:1, 1985, pages 43-70.

[Cowt85] W. Cowther, J. Goodhue, E. Starr, E. Thomas, W. Williken and T. Blackadar, 'Performance measurements on a 128-node Butterfly parallel processor', *Proceedings of the 1985 International Conference on Parallel Processing*, University Park, Pennsylvania, August 1985, pages 531-540.

[Cox81] Philip T. Cox and Tomasz Pietrzykowski, 'Deduction plans: a basis for intelligent backtracking', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3:1, January 1981, pages 52-65.

[Cram84] James A. Crammond and C. D. F. Miller, 'An architecture of parallel logic languages', *Proceedings of the Second International Logic Programming Conference*, Uppsala, Sweden, July 1984, pages 183-194.

[Cram86] Jim Crammond, 'An execution model for committed-choice non-deterministic languages', *Proceedings of the Third IEEE Symposium on Logic Programming*, Salt Lake City, Utah, September 1986, pages 148-158.

[Cram88] Jim Crammond, 'Implementation of committed choice logic languages on shared memory multiprocessors', Research Report PAR 88/4, Department of Computing, Imperial College of Science and Technology, University of London, England, May 1988, Ph.D. thesis from Heriot-Watt University.

[Dart87] Philip W. Dart, 'Using propositional formulae to derive deductive database dependencies', Technical Report 87/13, Department of Computer Science, University of Melbourne, Melbourne, Australia, September 1987, 24 pages.

[Dart88] Philip W. Dart and Justin Zobel, 'Transforming typed logic programs into well-typed logic programs', Technical Report 88/11, Department of Computer Science, University of Melbourne, Melbourne, Australia, May 1988.

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

[DeGr87] Doug DeGroot, 'Restricted AND-parallelism and side effects', *Proceedings of the Fourth IEEE Symposium on Logic Programming*, San Francisco, California, August 1987, pages 80-89.

[Debr88a] Saumya K. Debray, 'Efficient dataflow analysis of logic programs', *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988, pages 260-273.

[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*, 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 Third IEEE Symposium on Logic Programming*, Salt Lake City, Utah, September 1986, pages 78-88.

[Debr87] Saumya Debray, ‘Flow analysis of a simple class of dynamic logic programs’, *Proceedings of the Fourth IEEE Symposium on Logic Programming*, San Francisco, California, August 1987, pages 307-316.

[Debr88b] Saumya Debray, ‘Static analysis of parallel logic programs’, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, pages 711-732.

[Demb85] P. Dembinski and J. Maluszynski, ‘AND-parallelism with intelligent backtracking for annotated logic programs’, *Proceedings of the Second IEEE 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.

[Dera85] Pierre Deransart and Jan Maluszynski, ‘Relating logic programs and attribute grammars’, *Journal of Logic Programming*, 2:2, July 1985, pages 119-155.

[Desp87] A. Despain, Y. Patt, V. Srini, P. Bitar, W. Bush, C. Chien, B. Fagin, W. Hwu, S. Melvin, R. McGeer, A. Singhal, M. Shebanow and P. van Roy, ‘Aquarius’, *SIGARCH Computer Architecture News*, 15:1, March 1987, pages 22-34.

[Devi88] Philippe Devienne, ‘Weighted graphs - a tool for expressing the behaviour of recursive rules in logic programming’, *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988, pages 397-404.

[Diet88] Roland Dietrich and Frank Hagl, ‘A polymorphic type system with subtypes for Prolog’, *Proceedings of the Second European Symposium on Programming*, Nancy, France, March 1988, pages 79-93, published as Lecture Notes in Computer Science 300 by Springer-Verlag.

[Diet87] Suzanne Wagner Dietrich, ‘Extension tables: memo relations in logic programming’, *Proceedings of the Fourth IEEE Symposium on Logic Programming*, San Francisco, California, August 1987, pages 264-272.

[Dijk79] Edsger W. Dijkstra, ‘An essay on the notion: the scope of variables’, in *A discipline of programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1979, pages 79-93.

[Disz87] Terry Disz, Ewing Lusk and Ross Overbeek, ‘Experiments with OR-parallel logic programs’, *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pages 576-600.

[Dona85] James Donahue and Alan Demers, ‘Data types are values’, *ACM Transactions on Programming Languages and Systems*, 7:3, July 1985, pages 426-445.

[Drab87] Wlodzimierz Drabent, ‘Do logic programs resemble programs in conventional languages ?’, *Proceedings of the Fourth IEEE Symposium on Logic Programming*, San Francisco, California, August 1987, pages 389-396.

[Drab88] Wlodek Drabent, Simin Nadjim-Tehrani and Jan Maluszynski, ‘Algorithmic debugging with assertions’, *Proceedings of the Workshop on Meta-programming in Logic Programming*, Bristol, England, June 1988, pages 365-378.

[Emde79] M.H. van Emden and D.J. de Lucena, ‘Predicate logic as a language for parallel programming’, Research Report CS-79-15, Department of Computer Science, University of Waterloo, Ontario, Canada, 1979.

[Erik84] Lars-Henrik Eriksson and Mannr Rayner, ‘Incorporating mutable arrays into logic programming’, *Proceedings of the Second International Logic Programming Conference*, Uppsala, Sweden, July 1984, pages 101-114.

[Fagi86] Barry S. Fagin and Alvin M. Despain, ‘Goal caching in Prolog’, *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, Volume 1*, Honolulu, Hawaii, January 1986, pages 277-281.

[Ferr87] Gerard Ferrand, ‘Error diagnosis in logic programming, an adaptation of E.Y. Shapiro’s method’, *Journal of Logic Programming*, 4:3, September 1987, pages 177-198.

[Fokk81] Maarten M. Fokkinga, ‘On the notion of strong typing’, *Proceedings of the International Symposium on Algorithmic Languages*, Amsterdam, The Netherlands, 1981, pages 305-320, published as “Algorithmic languages” by North-Holland.

[Fost87a] Ian T. Foster, ‘Logic operating systems: design issues’, *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pages 910-926.

[Fost86] Ian T. Foster and Anthony J. Kusalik, ‘A logical treatment of secondary storage’, *Proceedings of the Third IEEE Symposium on Logic Programming*, Salt Lake City, Utah, September 1986, pages 58-67.

[Fost87b] Ian Foster, ‘Parallel implementation of Parlog’, Research Report PAR 87/19, Department of Computing, Imperial College of Science and Technology, University of London, England, July 1987, 20 pages.

[Fost87c] Ian Foster and Stephen Taylor, ‘Flat Parlog: a basis for comparison’, *International Journal of Parallel Programming*, 16:2, April 1987, pages 87-125.

[Fran80] Nissim Francez, ‘Distributed termination’, *ACM Transactions on Programming Languages and Systems*, 2:1, January 1980, pages 42-55.

[Fruh88] Thom W. Fruhwirth, ‘Type inference by program transformation and partial evaluation’, *Proceedings of the Workshop on Meta-programming in Logic Programming*, Bristol, England, June 1988, pages 199-212.

[Gall87] H. Gallaire, ‘Boosting logic programming’, *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pages 962-988.

[Gogu84] Joseph A. Goguen and Jose Meseguer, ‘Equality, types, modules and (why not ?) generics for logic programming’, *Journal of Logic Programming*, 1:2, August 1984, pages 179-210.

[Goto88] A. Goto, Y. Kimura, T. Nakagawa and T. Chikayama, 'Lazy reference counting - an incremental garbage collection method for parallel inference machines', *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, pages 1241-1256.

[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, England, September 1985, 263 pages.

[Hank87] W. Hankley, 'Feature analysis of Turbo Prolog', *SIGPLAN Notices*, 22:3, March 1987, pages 111-118.

[Hari88] Seif Haridi and Per Brand, 'Andorra Prolog - an integration of Prolog and committed choice languages', *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988, pages 745-754.

[Hauc68] E.A. Hauck and B.A. Dent, 'Burroughs' B6500/B7500 stack mechanism', *Proceedings of the AFIPS Spring Joint Computer Conference*, Atlantic City, New Jersey, April 1968, pages 245-251.

[Haus88] Bogumil Hausman, Andrej Ciepielewski and Alan Calderwood, 'Cut and side-effects in OR-parallel Prolog', *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988, pages 831-840.

[Hend82] P. Henderson, 'Purely functional operating systems', *Proceedings of the Workshop on Functional Programming and its Applications*, 1982, pages 177-192.

[Herm86a] Manuel V. Hermenegildo, 'An abstract machine based execution model for computer architecture design and efficient implementation of logic programs in parallel', Ph.D. thesis, University of Texas at Austin, August 1986.

[Herm86b] Manuel V. Hermenegildo, 'An abstract machine for restricted AND-parallel execution of logic programs', *Proceedings of the Third International Conference on Logic Programming*, London, England, July 1986, pages 25-39, published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Herm87a] M.V. Hermenegildo, 'Relating goal-scheduling, precedence and memory management in AND-parallel execution of logic programs', *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pages 556-575.

[Herm86c] Manuel V. Hermenegildo and Roger I. Nasr, 'Efficient management of backtracking in AND-parallelism', *Proceedings of the Third International Conference on Logic Programming*, London, England, July 1986, pages 40-54, published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Herm87b] M.V. Hermenegildo and R.A. Warren, 'Designing a high performance parallel logic programming system', *SIGARCH Computer Architecture News*, 15:1, March 1987, pages 43-52.

[Hoar78] C.A.R. Hoare, ‘Communicating sequential processes’, *Communications of the ACM*, 21:8, August 1978, pages 666-677.

[Hoar81] Charles Anthony Richard Hoare, ‘The emperor’s old clothes’, *Communications of the ACM*, 24:2, February 1981, pages 75-83.

[Hunt87] Matthew M. Huntbach, ‘Algorithmic Parlog debugging’, *Proceedings of the Fourth IEEE Symposium on Logic Programming*, San Francisco, California, August 1987, pages 288-297.

[Ichi87] N. Ichiyoshi, T. Miyazaki and K. Taki, ‘A distributed implementation of flat GHC on the multi-PSI’, *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pages 257-275.

[Jaff84] Joxan Jaffar, ‘Efficient unification over infinite terms’, *New Generation Computing*, 2:3, 1984, pages 207-219.

[John88] Eric E. Johnson, ‘Completing an MIMD multiprocessor taxonomy’, *SIGARCH Computer Architecture News*, 16:3, June 1988, pages 44-47.

[Jr75] Frederick P. Brooks, Jr., ‘The mythical man-month - essays on software engineering’, Addison-Wesley Publishers, Reading, Massachusetts, 1975, 195 pages.

[Kana85] Tadashi Kanamori and Kenji Horiuchi, ‘Type inference in Prolog and its applications’, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, California, September 1985, pages 704-707.

[Kasi85] Simon Kasif and Jack Minker, ‘The intelligent channel: a scheme for result sharing in logic programs’, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, California, September 1985, pages 29-30.

[Keef87] Richard A. O’Keefe, ‘Findall/3’, , 8710120717.AA18779 @ ucbvax.Berkeley.EDU, 1 October 1987.

[Keef88] Richard A. O’Keefe, ‘Practical Prolog for real programmers’, Notes for a tutorial, Fifth International Conference/Symposium on Logic Programming, Seattle, Washington, August 1988, 164 pages.

[Klin88a] Shmuel Klinger and Ehud Shapiro, ‘A decision tree compilation algorithm for FCP(!,:,?)’, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, pages 1315-1336.

[Klin88b] Shmuel Klinger, Eyal Yardeni, Kenneth Kahn and Ehud Shapiro, ‘The language FCP(:,?)’, *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988, pages 763-773.

[Kond88] Sei-ichi Kondoh and Takashi Chikayama, ‘Macro processing in Prolog’, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, pages 466-480.

[Kore88] Bogdan Korel, ‘PELAS - program error-locating assistant system’, *IEEE Transactions on Software Engineering*, SE-14:9, September 1988, pages 1253-1260.

[Kowa80] Robert A. Kowalski, ‘Logic for problem solving’, North Holland, New York, 1980.

[Kuma88] Vipin Kumar and Yow-Jian Lin, ‘A data-dependency-based intelligent backtracking scheme for Prolog’, *Journal of Logic Programming*, 5:2, June 1988, pages 165-181.

[Kusa86] Anthony J. Kusalik, ‘Specification and initialization of a logic computer system’, *New Generation Computing*, 4:2, 1986, pages 189-209.

[Kusa88] Anthony Joseph Kusalik, ‘Logic programming as a formalism for specification and implementation of computer systems’, Ph.D. thesis, University of British Columbia, Vancouver, Canada, June 1988, 128 pages.

[Levy84] Jacob Levy, ‘A unification algorithm for Concurrent Prolog’, *Proceedings of the Second International Logic Programming Conference*, Uppsala, Sweden, July 1984, pages 333-341.

[Lin88a] Yow-Jian Lin, ‘A parallel implementation of logic programs’, Technical Report AI88-84, Artificial Intelligence Laboratory, University of Texas at Austin, August 1988, Ph.D. dissertation.

[Lin86a] Yow-Jian Lin and Vipin Kumar, ‘A parallel execution scheme for exploiting AND-parallelism of logic programs’, *Proceedings of the 1986 International Conference on Parallel Processing*, University Park, Pennsylvania, August 1986, pages 972-975.

[Lin87] Yow-Jian Lin and Vipin Kumar, ‘An execution model for exploiting AND-parallelism in logic programs’, *New Generation Computing*, 5:4, 1987, pages 393-425.

[Lin88b] Yow-Jian Lin and Vipin Kumar, ‘AND-parallel execution of logic programs on a shared memory multiprocessor: a summary of results’, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, pages 1123-1141.

[Lin88c] Yow-Jian Lin and Vipin Kumar, ‘Performance of AND-parallel execution of logic programs on a shared memory multiprocessor’, *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988, pages 851-860.

[Lin86b] Yow-Jian Lin, Vipin Kumar and Clement Leung, ‘An intelligent backtracking algorithm for parallel execution of logic programs’, *Proceedings of the Third International Conference on Logic Programming*, London, England, July 1986, pages 55-68, published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Lind87] Timothy G. Lindholm and Richard A. O’Keefe, ‘Efficient implementation of a defensible semantics for dynamic Prolog code’, *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pages 21-39.

[Lloy86a] John W. Lloyd, ‘Declarative error diagnosis’, Technical Report 86/3, Department of Computer Science, University of Melbourne, Melbourne, Australia, February 1986, 20 pages.

[Lloy87] John W. Lloyd, ‘Foundations of logic programming (second, extended edition)’, Springer-Verlag, New York, Springer series in symbolic computation, 1987, 212 pages.

[Lloy84] J.W. Lloyd and R.W. Topor, ‘Making Prolog more expressive’, *Journal of Logic Programming*, 1:3, October 1984, pages 225-240.

[Lloy86b] J. Lloyd and A. Takeuchi, ‘A framework for debugging GHC’, ICOT Technical Report TR-186, Institute for New Generation Computer Technology, Tokyo, Japan, 1986.

[Marr88a] Kim Marriott and Harald Sondergaard, ‘Bottom-up abstract interpretation of logic programs’, Technical Report 88/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, May 1988, 17 pages.

[Marr88b] K. Marriott and H. Sondergaard, ‘Semantics-based dataflow analysis of logic programs’, Technical Report 88/27, Department of Computer Science, University of Melbourne, Melbourne, Australia, November 1988.

[Mats85] Hidekazu Matsumoto, ‘A static analysis of Prolog programs’, *SIGPLAN Notices*, 20:10, October 1985, pages 48-59.

[Maye82] Alastair J.W. Mayer, ‘The architecture of the Burroughs B5000 - 20 years later and still ahead of the times ?’, *SIGARCH Computer Architecture News*, 10:4, June 1982, pages 3-10.

[Mell81] C.S. Mellish, ‘The automatic generation of mode declarations for Prolog programs’, Research Paper 163, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1981.

[Mell85] C.S. Mellish, ‘Some global optimisations for a Prolog compiler’, *Journal of Logic Programming*, 2:1, April 1985, pages 43-66.

[Mich68] Donald Michie, ‘Memo functions and machine learning’, *Nature*, vol. 218, April 6, 1968, pages 19-22.

[Mier85] C. Mierowsky, S. Taylor, E. Shapiro, J. Levy and S. Safra, ‘The design and implementation of Flat Concurrent Prolog’, Research Report CS85-09, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, July 1985.

[Mill86] J. Mills, ‘A high performance LOW RISC machine for logic programming’, Technical Report TR-86-008, Computer Science Department, Arizona State University, 1986.

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

[Mish84] Prakeesh Mishra, ‘Towards a theory of types in Prolog’, *Proceedings of the First IEEE Symposium on Logic Programming*, Atlantic City, New Jersey, February 1984, pages 289-298.

[Mitc88] John C. Mitchell and Robert Harper, ‘The essence of ML’, *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988, pages 28-46.

[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, ‘All solutions predicates in Prolog’, *Proceedings of the Second IEEE Symposium on Logic Programming*, Boston, Massachusetts, July 1985, pages 73-77.

[Nais85b] Lee Naish, ‘Automating control of logic programs’, *Journal of Logic Programming*, 2:3, October 1985, pages 167-183.

[Nais85c] 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.

[Nais87] Lee Naish, ‘Specification = program + types’, *Proceedings of the Seventh Conference on the Foundations of Software Technology and Theoretical Computer Science*, Pune, India, December 1987, pages 326-339, published as Lecture Notes in Computer Science 287 by Springer-Verlag.

[Nais88a] Lee Naish, ‘Parallelizing NU-Prolog’, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, pages 1546-1564.

[Nais88b] Lee Naish, ‘Declarative diagnosis of missing answers’, Technical Report 88/9, Department of Computer Science, University of Melbourne, Melbourne, Australia, May 1988, 18 pages.

[Nais88c] Lee Naish, Philip W. Dart and Justin Zobel, ‘The NU-Prolog debugging environment’, Technical Report 88/31, Department of Computer Science, University of Melbourne, Melbourne, Australia, November 1988, 15 pages.

[Naka86] Katsuhiko Nakamura, ‘Control of logic program execution based on the functional relations’, *Proceedings of the Third International Conference on Logic Programming*, London, England, July 1986, pages 505-512, published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Nich88] A.E. Nicholson, ‘Declarative debugging of the parallel logic programming language GHC’, *Proceedings of the Eleventh Australian Computer Science Conference*, Brisbane, Australia, February 1988, pages 225-236.

[Nils83] Jorgen Fischer Nilsson, ‘On the compilation of a domain-based Prolog’, *Proceedings of the Ninth IFIP Congress (Information Processing 83)*, Paris, France, September 1983, pages 293-298.

[Patt85] David A. Patterson, ‘Reduced instruction set computers’, *Communications of the ACM*, 28:1, January 1985, pages 8-21.

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

[Pere80] Luis Moniz Pereira and Antonio Porto, ‘Selective backtracking for logic programs’, *Proceedings of the Fifth Conference on Automated Deduction*, Les Arcs, France, 1980, pages 306-317, published as Lecture Notes in Computer Science 87 by Springer-Verlag.

[Poll81] George H. Pollard, ‘Parallel execution of Horn clause programs’, Ph.D. thesis, Department of Computing, Imperial College of Science and Technology, University of London, England, 1981, 226 pages.

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

[Port82] Antonio Porto, ‘Epilog: a language for extended programming in logic’, *Proceedings of the First International Logic Programming Conference*, Marseille, France, September 1982, pages 31-37.

[Rama88] Raghu Ramakrishnan, ‘Magic templates: a spellbinding approach to logic programming’, *Proceedings of the Fifth International Conference/Symposium* *on Logic Programming*, Seattle, Washington, August 1988, pages 140-159.

[Rama86] Raghu Ramakrishnan and Avi Silberschatz, ‘Annotations for distributed programming in logic’, *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1986, pages 255-262.

[Redd88] Uday S. Reddy, ‘Notions of polymorphism for predicate logic programs’, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, page appendix.

[Ross86] M.L. Ross and K. Ramamohanarao, ‘Paging strategy for Prolog based dynamic virtual memory’, *Proceedings of the Third IEEE Symposium on Logic Programming*, Salt Lake City, Utah, September 1986, pages 46-57.

[Russ78] Richard M. Russell, ‘The Cray-1 computer system’, *Communications of the ACM*, 21:1, January 1978, pages 63-72.

[Safr86] Shmuel Safra and Ehud Shapiro, ‘Meta interpreters for real’, *Proceedings of the Tenth IFIP Congress (Information Processing 86)*, Dublin, Ireland, September 1986, pages 271-278.

[Sand78] Erik Sandewall, ‘Programming in the interactive environment: the LISP experience’, *ACM Computing Surveys*, 10:1, March 1978, pages 35-71.

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

[Sara88] Vijay A. Saraswat, ‘A somewhat logical formulation of CLP synchronization primitives’, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, pages 1298-1314.

[Sato87] Masatoshi Sato, Hajime Shimizu, Akira Matsumoto, Kazuaki Rokusawa and Atsuhiro Goto, ‘KL1 execution model for PIM cluster with shared memory’, *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pages 338-355.

[Sawa85] Hajime Sawamura and Taku Takeshima, ‘Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to Prolog optimization’, *Proceedings of the Second IEEE Symposium on Logic Programming*, Boston, Massachusetts, July 1985, pages 200-207.

[Shap82] Ehud Y. Shapiro, ‘Algorithmic program debugging’, Ph.D. thesis, Yale University, New Haven, Connecticut, May 1982.

[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.

[Shap84a] Ehud Y. Shapiro and Colin Mierowsky, ‘Fair, biased and self-balancing merge operators: their specification and implementation in Concurrent Prolog’, *New Generation Computing*, 2:3, 1984, pages 221-240.

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

[Shap84b] Ehud Shapiro, ‘Systems programming in Concurrent Prolog’, *Conference Record of the Eleventh ACM Symposium on Principles of Programming Languages*, Salt Lake City, Utah, January 1984, pages 93-105.

[Shap86] Ehud Shapiro, ‘Concurrent Prolog: a progress report’, *IEEE Computer*, 19:8, August 1986, pages 44-59.

[Shap85] Ehud Shapiro and Muli Safra, ‘Fast multiway merge using destructive operation’, *Proceedings of the 1985 International Conference on Parallel Processing*, University Park, Pennsylvania, August 1985, pages 118-122.

[Shoh84] Yoav Shoham and Drew V. McDermott, ‘Directed relations and inversion of Prolog programs’, *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1984, pages 307-316.

[Sick79] Sharon Sickel, ‘Invertibility of logic programs’, *Proceedings of the Fourth Workshop on Automated Deduction*, Austin, Texas, February 1979, pages 103-109.

[Silv86] W. Silverman and et al, ‘The Logix system user manual, version 1.2’, Research Report CS86-21, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, 1986.

[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.

[Ster88] Leon Sterling and Arun Lakhotia, ‘Composing Prolog meta-interpreters’, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, pages 386-403.

[Stro86] R.E. Strom and S. Yemini, ‘Typestate: a programming language concept for enhancing software reliability’, *IEEE Transactions on Software Engineering*, SE-12:1, January 1986, pages 157-171.

[Taka84] Shigeyuki Takagi, Toshio Yokoi, Shunichi Uchida, Toshiaki Kuorkawa, Takashi Hattori, Takashi Chikayama, Ko Sakai and Junichuro Tsuji, ‘Overall design of SIMPOS’, *Proceedings of the Second International Logic Programming Conference*, Uppsala, Sweden, July 1984, pages 1-12.

[Take86] A. Takeuchi, ‘Algorithmic debugging of GHC programs and its implementation in GHC’, ICOT Technical Report TR-185, Institute for New Generation Computer Technology, Tokyo, Japan, 1986.

[Tama84] Hisao Tamaki and Taisuke Sato, ‘Unfold/fold transformation of logic programs’, *Proceedings of the Second International Logic Programming Conference*, Uppsala, Sweden, July 1984, pages 127-138.

[Tayl86] Stephen Taylor, Shmuel Safra and Ehud Shapiro, ‘A parallel implementation of Flat Concurrent Prolog’, *International Journal of Parallel Programming*, 15:3, June 1986, pages 245-275.

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

[Thor70] J.E. Thornton, ‘The design of a computer: the Control Data 6600’, Scott, Foresman & Company, 1970, 181 pages.

[Toh86] Jin Ai Toh and Kotagiri Ramamohanarao, ‘Failure directed backtracking’, Technical Report 86/9, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986, 30 pages.

[Ueda86a] Kazunori Ueda, ‘Making exhaustive search programs deterministic’, *Proceedings of the Third International Conference on Logic Programming*, London, England, July 1986, pages 270-282, published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Ueda86b] Kazunori Ueda, ‘Guarded Horn clauses’, D.Eng. Thesis, University of Tokyo, Tokyo, Japan, March 1986.

[Vagh87] Jayen Vaghani, ‘Towards a practical intelligent backtracking scheme for full Prolog’, Honours report, Department of Computer Science, University of Melbourne, Melbourne, Australia, November 1987, 25 pages.

[Voda88] Paul J. Voda, ‘Types of Trilogy’, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, pages 580-589.

[Voda84] Paul J. Voda and Benjamin Yu, ‘RF-Maple: A logic programming language with functions, types and concurrency’, *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1984, pages 341-347.

[Wadl84] Philip Wadler, ‘Listlessness is better than laziness: lazy evaluation and garbage collection at compile time’, *Conference Record of the ACM Symposium on LISP and Functional Programming*, Austin, Texas, July 1984, pages 45-52.

[Wadl88] Philip Wadler, ‘Deforestation: transforming programs to eliminate trees’, *Proceedings of the Second European Symposium on Programming*, Nancy, France, March 1988, pages 344-358, published as Lecture Notes in Computer Science 300 by Springer-Verlag.

[Warr77] David H.D. Warren, ‘Implementing Prolog - compiling predicate logic programs’, Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, Scotland, May 1977.

[Warr80] David H.D. Warren, ‘Logic programming and compiler writing’, *Software - Practice and Experience*, 10:2, February 1980, pages 97-125.

[Warr82] David H.D. Warren, ‘Perpetual processes: an unexploited Prolog technique’, *Logic Programming Newsletter*, Summer 1982, page 2.

[Warr83] David H.D. Warren, ‘An abstract Prolog instruction set’, Technical Note 309, SRI International, Menlo Park, California, October 1983.

[Warr88a] David H.D. Warren and Seif Haridi, ‘The data diffusion machine - a shared virtual memory architecture for parallel execution of logic programs’, *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988, pages 943-952.

[Warr84] David Scott Warren, ‘Database updates in pure Prolog’, *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1984, pages 244-253.

[Warr88b] Richard Warren, Manuel Hermenegildo and Saumya K. Debray, ‘On the practicality of global flow analysis of logic programs’, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, pages 684-699.

[Wegb71] B. Wegbreit, ‘The ECL programming system’, *Proceedings of the AFIPS Fall Joint Computer Conference*, Las Vegas, Nevada, November 1971, pages 253-262.

[Wegb74] Ben Wegbreit, ‘The treatment of data types in EL1’, *Communications of the ACM*, 17:5, May 1974, pages 251-264.

[Wein71] Gerald M. Weinberg, ‘The psychology of computer programming’, Van Nostrand Reinhold, New York, 1971, 288 pages.

[Wein84] P.J. Weinberger, ‘Cheap dynamic instruction counting’, *Bell Laboratories Technical Journal*, 63:8 (part 2), October 1984, pages 1815-1826.

[West87] Harald Westphal, Philippe Robert, Jacques Chassin and Jean-Claude Syre, ‘The PEPSys model: combining backtracking, AND- and OR-parallelism’, *Proceedings of the Fourth IEEE Symposium on Logic Programming*, San Francisco, California, August 1987, pages 436-448.

[Wilk81] A.L. Wilkinson, D.H. Anderson, D.P. Chang, L.H. Hin, A.J. Mayo, I.T. Viney, R. Williams and W. Wright, ‘A penetration analysis of a Burroughs large system’, *SIGOPS Operating Systems Review*, 15:1, January 1981, pages 14-25.

[Wins87] Will Winsborough, ‘Semantically transparent selective reset for AND parallel interpreters based on the origin of failures’, *Proceedings of the Fourth IEEE Symposium on Logic Programming*, San Francisco, California, August 1987, pages 134-152.

[Wise86] Michael Wise, ‘Prolog multiprocessors’, Prentice-Hall, Englewood Cliffs, New Jersey, December 1986, 168 pages.

[Woo86] Nam Sung Woo and Kwang-Moo Choe, ‘Selecting the backtrack literal in the AND/OR process model’, *Proceedings of the Third IEEE Symposium on Logic Programming*, Salt Lake City, Utah, September 1986, pages 200-210.

[Xu88] Jiyang Xu and David S. Warren, ‘A type inference system for Prolog’, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Washington, August 1988, pages 604-619.

[Yan88] Song Yuan Yan, ‘Foundations of declarative debugging in arbitrary logic programming’, Technical Report 88/30, Department of Computer Science, University of Melbourne, Melbourne, Australia, November 1988.

[Yang86] Rong Yang and Hideo Aiso, ‘P-Prolog: a parallel language based on exclusive relation’, *Proceedings of the Third International Conference on Logic Programming*, London, England, July 1986, pages 255-269, published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Yard87] E. Yardeni and E. Shapiro, ‘A type system for logic programs’, Research Report CS87-05, Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel, March 1987.

[Zobe87] J. Zobel, ‘Derivation of polymorphic types for Prolog programs’, *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pages 816-838.

Abstract

The aim of this thesis is the design of a parallel logic programming system that is a suitable vehicle for general purpose computing. We intend its area of applicability to be as wide as that of the language C: it should be a convenient tool in the hands of the next generation of professional programmers for the construction of operating systems, utilities and other application programs.

The centerpiece of our design is a new mode system that ensures that two calls will never disagree on whose responsibility it is to instantiate a given variable. If a variable appears in more than one call in a clause, the compiler will pick one of those calls as its producer based on declarations supplied by the programmer. The other calls will treat the variable as input; if they need its value they will delay until it is instantiated by the producer. However, producers do not need to generate whole values for variables. Our mode system is flexible enough to accommodate situations where two or more calls cooperate in the construction of a term, as long as the calls agree on which call should instantiate which part of the term. To allow us to talk precisely about such matters, we rely on a rather conventional type system to tell us about the possible structures of terms. Based on this information, we analyze binding patterns in terms of the smallest possible unit (individual functors); this gives us maximal flexibility among strong mode systems in the handling of partially instantiated data structures.

After defining our concept of modes and presenting our mode analysis algorithm, we discuss the implications of our type and mode systems and give comparisons with existing schemes, concentrating on software engineering issues. We argue that a strong mode system is desirable for the same reason a strong type system is: it detects an important class of programmer errors at compile time (errors involving incorrect management of logical variables). As such, it contributes to reliability by ensuring that these errors will be detected even if they do not lead to incorrect output in any test case. It should help improve programmer productivity, as it allows the compiler to generate messages that guide programmers more or less directly to the erroneous piece of code. Finally, we argue that mode declarations improve readability and thus make software maintenance easier.

The main form of parallelism in our system is stream AND-parallelism, with the usual dataflow synchronisation (which can be implemented much more cleanly in our strongly moded context). We believe that stream AND-parallelism is a necessity in logic programming languages whose intended scope includes operating systems, as it is the natural expression of multiple cooperating processes. The main problem of such languages is that they must ensure that side-effect-causing variable bindings must never be retracted, as real world devices do not like being told to do more than one thing in any one situation. To this end, the concurrent logic programming languages Parlog, Concurrent Prolog and GHC outlaw don't-know nondeterminism altogether, abandoning the conventional semantics of logic programs. Armed with the information provided by modes, we can afford to be more selective. We allow programs to have don't-know nondeterminism as long as the compiler can prove that this will never cause ambiguous commands to be sent to a real-world device; in practice this rule affects only the topmost layers of user programs.

The other side of the coin is that our execution algorithm must be able to support both stream AND-parallelism and don't-know nondeterminism in a reasonably efficient manner. Again the mode system is the key to our solution; without it the forward execution algorithm would have to spend too much time collecting dataflow information. The modes supply us with compile-time

knowledge of the dataflow graph of each clause; this lets us figure out efficiently which goals to kill or restart when a goal fails. It also allows us to localise the effects of failures; the failure of a goal in one user program will never affect goals in any other program, and will affect only directly related goals in its own program. We also show that modes increase the precision with which we select the backtrack literal, making our system as good as (and sometimes better than) most intelligent backtracking algorithms. Of course our backtracking algorithm can handle programs that manipulate partially instantiated data structures; it can also handle operating system-type programs with circular dependency graphs (e.g. those based on the client-server model).

Towards the end of the thesis, we investigate implementation issues. After some discussion of the forward execution algorithm, we show how to take advantage of some properties of our backward execution algorithm to simplify its implementation on multiprocessors. Next we demonstrate a technique that can trade off the precision of the backtracking algorithm against its overhead on a procedure-by-procedure and call-by-call basis. We then show how a fast processor executing our language (or in fact any parallel logic language) can tolerate the comparatively large latency of the memory system.

Preface

This thesis contains no material that has been accepted or is under consideration for the award of any other degree or diploma in any other university. To the best of my knowledge it contains no material previously published or written by any other person except where the text of the thesis makes explicit reference.

Most parts of this thesis have been published before, although they have been extensively modified and updated for inclusion in this thesis. Chapter 2 and a rudimentary version of chapter 3 had first appeared in the proceedings of the Fourth International Conference on Logic Programming [Somo87a]. Much of chapter 4 has appeared as a Melbourne University technical report [Somo87b]; this technical report is currently being refereed by the Journal of Logic Programming. Chapter 5 was published in the proceedings of the Fifth International Conference/Symposium on Logic Programming [Somo88]. An extended version of that paper, incorporating material from chapter 6, will soon appear in the International Journal of Parallel Programming. Other parts of chapter 6 appeared in a handout that was part of the informal “proceedings” of the Workshop on Concurrent Programming held at ICOT after the FGCS conference in December of 1988.

This thesis is less than 100,000 words in length.

Acknowledgements

I would like to thank my father, who introduced me to computers and programming (in the guise of a PDP-8 clone and Fortran respectively); Victor Veres, who told me more than anybody would ever want to know about how to get more performance out of cars; Ben Golding, who taught me never to take any wooden nickels; Paul Dunn, who first fired up my enthusiasm for computer science by his after-the-lecture chats; JL, who first convinced me that logic programming really is a better way; Graeme Port, who first introduced me to the notion of modes; Rao, who is the best wall in existence for bouncing ideas off; John Shepherd, who first got me involved in underground design and human and creature movement studies; Isaac Balbin, who taught me never to say "And how are you ?"; Ann Nicholson, who has proofread parts of this thesis despite being half the world away; Jayen Vaghani, who in his honours report proved to my satisfaction (but not to his) that side-effects and intelligent backtracking do not mix; Robert Elz, without whom our machines wouldn't have stayed up long enough for me to write this thesis; and all the other members of the Computer Science Department, who created an environment that can be called many things but cannot possibly be called "boring".

I am especially grateful to Rao, John, Jayen, Graeme, Ann, Will Winsborough, Reem Bahgat and the referees of the papers on which this thesis is based for their careful proofreading.

This research was supported by a Commonwealth Postgraduate Research Award and the Australian Computer Research Board through scholarships, and the Commonwealth Department of Science and the Australian Research Grants Scheme through the Machine Intelligence Project.

Table of contents