Parallel Execution of Logic Programs

John S. Conery

University of Oregon

For Leslie ♡

Contents

List of Figures

Preface

This book is an updated version of my Ph.D. dissertation, *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. The three years since that paper was finished (or so I thought then) have seen quite a bit of work in the area of parallel execution models and programming languages for logic programs. A quick glance at the bibliography here shows roughly 50 papers on these topics, 40 of which were published after 1983. The main difference between the book and the dissertation is the updated survey of related work.

One of the appendices in the dissertation was an overview of a Prolog implementation of an interpreter based on the AND/OR Process Model, a simulator I used to get some preliminary measurements of parallelism in logic programs. In the last three years I have been involved with three other implementations. One was written in C and is now being installed on a small multiprocessor at the University of Oregon. Most of the programming of this interpreter was done by Nitin More under my direction for his M.S. project. The other two, one written in Multilisp and the other in Modula-2, are more limited, intended to test ideas about implementing specific aspects of the model. Instead of an appendix describing one interpreter, this book has more detail about implementation included in Chapters 5 through 7, based on a combination of ideas from the four interpreters.

One of the implementation methods is an algorithm for generating multiple results in nondeterministic programs during parallel execution. The algorithm in the dissertation had a flaw, in that it fails to generate all possible results in certain situations. The flaw was first pointed out to me by Jung-Herng Chang. Also, N. S. Woo and K-M Choe were kind enough to send me a preprint of their improved algorithm. The algorithm presented here in Chapter 6 does not have the same flaw. It was developed in the context of the Modula-2 implementation of parallel AND processes, and differs from the other two in several respects.

Other than the updated survey and the incorporation of recent implementation methods into the main text, and some minor changes in presentation that will hopefully make some topics more clear, this work is basically the

same as the dissertation.

I would again like to thank my Ph.D. advisor, Dennis Kibler, for his outstanding support and encouragement during my career as a graduate student. Also, I still remember the thoughtful comments of Bruce Porter, Paul Morris, Jim Neighbors, and Steve Fickas. More recently, Paul Bloch, Dave Meyer, Nitin More, Don Pate, and Tsyuoshi Shinogi have all made contributions to the continuing project and the book. I have also benefited from my association with Al Davis and his group at Schlumberger Palo Alto Research, Gary Lindstrom, and Doug DeGroot. In fact, Doug is mostly responsible for the existence of this book. The Department of Computer and Information Science at the University of Oregon provided the resources to format and print the laserscript and to carry out the research described in the last chapter.

Finally, I would like to express my appreciation to my wife Leslie; her love, patience, support, and interest have made this task much easier than it otherwise would have been.

JC

Eugene, Oregon
August, 1986

Parallel Execution of Logic Programs

Chapter 1

Introduction

The past few years have seen an explosion of interest in the field of logic programming. An indication of the interest is attendance at meetings of researchers in the field. Initially, small annual workshops, such as those held at Debrecen, Hungary in 1980 and Syracuse, New York in 1981, were sufficient to disseminate the latest results. Five years later, in the summer of 1986, two major international conferences are being held, with more than 100 different papers submitted to each. In addition, two journals are devoted exclusively to logic programming, and journals for artificial intelligence, programming languages, and computer architecture regularly feature articles related to logic programming.

Much of the current research involves techniques for implementing logic programming languages, such as Prolog. One of the attractions of logic programming is the clean separation of semantics and control. It is easy to separate specification of what a program should compute from how an implementation can efficiently compute it. Thus, even though early implementations were quite inefficient compared to conventional programming languages, there has been the promise of more efficient implementation, since it is possible to experiment with different implementation techniques without violating the semantics. This experimentation has taken the form of investigating the effect of representations such as "structure sharing" on the use of memory, more efficient unification algorithms, virtual machine instruction sets, and alternative control strategies.

The major advantage of the separation of semantics from control, however, is the potential for parallelism. When it is clear what the final result of a computation has to be, and that any of a number of different sequences of operations will lead to the result, it is reasonable to expect to do some of the operations in parallel. Such is the case with logic programs.

The subject of this book is the AND/OR Process Model, an abstract

Theory	A model of computation and semantics of programs in the model.
Operation	Abstract interpreter; rules for performing steps of a computation and ordering the steps.
Implementation	Concrete interpreter, specifying representation of programs and data.
Machine	Low level implementation; hardware components, interconnection, process distribution.

The AND/OR Process Model is a specification at the Operation layer in this hierarchy. It is an operational semantics for interpreting logic programs, defining a framework for implementation of a parallel interpreter.

Figure 1.1: Layers of Abstraction

model for parallel execution of logic programs. The goal has been the definition of a framework for implementing parallel interpreters. The long range goal is the design of a large scale parallel processor. The research presented here provides an intermediate level of abstraction between hardware and semantics, a set of requirements for a parallel interpreter running on a multiprocessor architecture (Figure 1.1).

The AND/OR Process Model is not a model for parallel execution of Prolog. Prolog is a high level programming language based on logic, but a Prolog program is not a logic program, in the same sense in which a Common Lisp program is not a "pure Lisp" program. Prolog is one example of a system at the Implementation level of the hierarchy of Figure 1.1. It has many extensions to the formalism of logic programming that make programming more convenient. Many of the data representations and rules for executing Prolog programs are defined in terms of sequential machines. In the top-down design of a parallel machine starting from the most abstract levels, it would be a mistake to include all of the extensions of Prolog in the parallel model. Alternatively, one should define mechanisms for parallel control of logic, and then implement the practical extensions to the formalism in terms of those mechanisms. As a concrete example, the formalism of logic programming does not provide for conditional expressions. Conditional expressions are implemented in Prolog, but the definition is in terms of "cut," a control operator defined in terms of sequential search. Conditional expressions will be defined for the AND/OR Process Model in Chapter 7, using the mechanisms

of the parallel control instead Prolog's cut operation.

Important principles for an abstract parallel interpreter are *accuracy*, *scalability*, and *modularity*. Accuracy means the abstract interpreter should be as faithful as possible to the formal semantics of the model of computation. As always in a hierarchy of abstractions, compromises are made as lower levels implement concepts of the higher levels. A good example in the context of logic programming is a compromise made by the incompleteness of Prolog. There are cases where a Prolog interpreter will fail to compute a result for an input expression that has a defined meaning according to the semantics of logic programming. Accuracy in this context means the abstract interpreter should provide as many of the defined results as possible.

Scalability means the abstract interpreter can be implemented on a wide range of multiprocessors, able to exploit as much parallelism as possible given the number of processors and the size of an input problem. A classic example of non-scalable parallelism is a pipelined floating point unit in a vector processor. The number of independent stages of a floating point operation determines the amount of parallelism. The same factor of speedup is obtained in each execution, independent of the length of input vectors. If operations on vectors of length 100 are five times as fast, operations on vectors of length 10,000 will also be five times as fast. In a dataflow system, however, it is possible to exploit an amount of parallelism proportional to the size of the input problem. When multiplying two $n \times n$ matrices, $O(n^2)$ parallel tasks are created. At the level of the abstract interpreter, the speedup for 10×10 matrices is a factor of 100, and for 20×20 matrices the speedup is a factor of 400.

Modularity at the level of the abstract interpreter means parallel activities share a minimum of data, preferably communicating by messages instead of shared variables. Modularity will be important when the abstract interpreter is implemented on a multiprocessor. One of the major technological barriers for multiprocessors is the problem of memory access. When more than a few processors access the same memory, contention introduces delays. If all operations require access to a common memory, performance degrades as more processors are built into the system. However, if processors primarily access independent local memories, this performance bottleneck is avoided in large systems. By building modularity into the abstract interpreter, we are confident it can later be implemented on an independent memory multiprocessor.

The AND/OR Process Model is an abstract parallel interpreter based on the principles of accuracy, scalability, and modularity. It can be viewed as an Actors system, where numerous objects communicate via messages and update local state information in asynchronous operations. The local states of objects replace the centralized binding environment usually created in the execution of Prolog programs, and the asynchronous operation of the objects

replaces the sequential steps of Prolog interpreters.

This book starts with a complete introduction to logic programming. The purpose of this chapter is to describe the Theory level of the hierarchy of Figure 1.1, defining the model of computation and the formal semantics of logic programs. It provides a reference point for understanding aspects of the AND/OR Process Model. The chapter also contains a description of Prolog as an example of an interpreter based on the formalism.

The following chapter is a survey of different models of parallel interpreters. Some of the projects described here are, like the AND/OR Process Model, abstract models, systems useful for analyzing potential sources of parallelism in logic programming languages. Others are implementations of parallel logic programming languages at the lower two levels of the hierarchy.

The next three chapters describe the AND/OR Process Model in detail. An abstract interpreter, written in Prolog, was first used to test the accuracy of the model. More recently other interpreters have been written in order to test the model on a small scale multiprocessor. Aspects of these interpreters are described in Chapter 7.

Logic programming languages, like languages based on applicative models, are often inefficient in comparison to traditional languages when implemented on von Neumann architectures. One hope for more efficient implementation lies in parallel architectures. In fact, some argue the opposite side of the coin: languages based on non von Neumann models provide the key to the acceptance of large-scale parallel machines, due to the inherent difficulties of exploiting parallelism in von Neumann languages.

The philosophy behind the research presented here is that parallel architectures should be designed in a "top-down" fashion, proceeding from the formal model of computation to actual hardware. The abstract interpreter is an important step in this process, providing a set of design principles and specifications for the lower levels. Implementation techniques for an efficient concrete interpreter based on the specifications of the AND/OR Process Model are the subject of the last chapter of this book. Efficient parallel machines based on the interpreters will be the subject of future research.

Chapter 2

Logic Programming

The phrase *logic programming* refers to the use of formulas of first order predicate logic as statements of a programming language. The first logic programming system was developed by Colmerauer and his colleagues at Marseille, growing out of a project to implement an automatic theorem prover. Since then, the semantics of logic as a programming language have been formalized, and there have been a number of implementations of Prolog, a high level language that extends the formalism of logic programming in ways that make it more useful and efficient for solving practical problems.

This chapter is an extensive discussion of logic programming and Prolog. It starts with the definition of the syntax and formal semantics of logic programs. Next is a description of the standard sequential control strategy used by most interpreters, including Prolog, followed by a detailed description of the Prolog language, with its extensions to the formalism, and a discussion of alternative (but still single processor) control strategies used in various Prolog systems.

The discussion of Prolog is included here for three reasons. First, it reinforces the notion that logic programming is a theoretical foundation, the model of computation at the Theory layer of the scale in Figure 1.1. Prolog is one example of a system at the Operation layer, and the AND/OR Process Model is another. The intent here is to clarify what is an essential part of the formalism and what is part of an abstract interpreter based on the formalism.

Second, the discussion is offered in support of a claim made in the introduction, that it is relatively easy to separate semantics from control in systems based on logic programming. The presentation of the standard sequential control followed by numerous examples of alternative but still semantically correct control strategies provides a good introduction to the survey of parallel control in the next chapter.

Finally, Prolog provides an example of the kinds of extensions to the

formalism that are accepted as useful. The AND/OR Process Model has some of the same extensions, such as allowing program clauses to be operated on as data. Other extensions will be defined in terms of the underlying control primitives of the parallel model.

2.1 Syntax

The basic data objects of logic programs are *terms*. The simplest type of term is an *atom*. As in FP [1], atoms stand for themselves, and have no further meaning, *i.e.* there is no notion of binding an atom to a value as there is in Lisp.

Complex terms are built from an atomic *function symbol* and any number of *arguments*, which can be any type of term. Complex terms are usually written using prefix notation; thus the term with function symbol `f` and arguments `x` and `y` is written `f(x,y)`. The function symbol is also known as the *functor* or *principle functor* of the term. A complex term with n arguments is said to be an n-ary term, or to be of *arity* n. For the sake of regularity, atoms are considered to be complex terms of arity 0.

The third type of term is the *variable*. Syntactically, variables are distinguished from atoms by starting the names of variables with upper case letters: `A` and `B` are variables, but `a` and `b` are atoms. During execution, variables will take values. We say the variable is *replaced* by, or becomes *bound* to, or is *instantiated* to another term. The other term becomes the value of the variable.

Logic programming languages are single assignment languages. Once a variable takes a value in a computation, it keeps that value. If a variable occurs in many places, every occurrence takes the same value at the same time. So, for example, if the variable `X` in the term `p(X,Y,a,X)` is bound to the term `f(a)`, the term becomes `p(f(a),Y,a,f(a))`.

The smallest unit of a program is a *literal*. Syntactically a literal is the same as a non-variable term, having a function symbol and zero or more terms as arguments. Literals are combined into *clauses*, which have two components, a *head* and a *body* separated by a left arrow. The head can have at most one literal; the body can have zero or more literals. A period follows the last literal in the body. There are four different types of clauses, depending on the number of literals in the head and body. The types of clauses and examples of each are shown in Figure 2.1.

The text of a program usually consists of implications and assertions only. Goal statements and null clauses are derived by an interpreter as it executes a program. Assertions are also called *unit clauses*, and implications are *nonunit clauses*. The terminology of problem solving is sometimes used when describing the execution of a logic program. A clause may be called a

Clause Type	*Example*	
Implication	`p` ← `q` ∧ `r.`	One head literal, one or more body literals.
Assertion	`p.`	One head literal, zero body literals. The implication sign is omitted in this case.
Goal Statement	← `q` ∧ `r.`	No head literal, one or more body literals.
Null clause	□	No literals in head or body.

The four types of clauses in a logic program. Assertions are also known as unit clauses. *Implications and assertions form the text of a program, goal statements and null clauses are created during execution of the program.*

Figure 2.1: Examples of Clauses

goal, and literals in the right hand side may be *subgoals*.

Finally, the largest syntactic unit in a logic program is a *procedure*. A procedure is a set of clauses, all of which have head literals with the same function symbol and same arity. A literal `p(X)` in a goal statement or the body of a clause is a *call* to procedure `p`.

A simple logic program used for examples throughout the book is given in Figure 2.2. This program has seven procedures. Six are simply sets of assertions. The seventh, `paper`, is defined by two implications and one assertion.

User interaction with a logic programming system is similar to interaction with Lisp. A program is entered into the system, and the user then types expressions to be evaluated. The expressions typed in by the user are goal statements; evaluation consists of proving the goal statement with an automatic proof procedure and printing the bindings made for variables of the goal during the proof. We will sometimes refer to the initial goal statement as a *query*. The rules for evaluating logic expressions – the semantics of the language – will be given in the next section. The remainder of this section is an overview of the applications that can be written in this style.

One application, exemplified by the program in Figure 2.2, is database processing. The natural connection between logic programs and relational databases has been recognized for a long time [24, 35, 93]. In this application, tuples of a relation are represented by assertions in the logic program, where the functor of the assertion is the name of the relation. The database in Figure 2.2 is written in the style advocated by Deliyanni and Kowalski [26]. All information is stored in binary relations, where the first term in an as-

```
paper(P,D,I) ← date(P,D) ∧ author(P,A) ∧ loc(A,I,D).
paper(P,D,I) ← tr(P,I) ∧ date(P,D).
paper(xform,1978,uci).

author(fp,backus).              date(fp,1978).
author(df,arvind).              date(df,1978).
author(eft,kling).              date(eft,1978).
author(pro,pereira).            date(pro,1978).
author(sem,vanemden).           date(sem,1976).
author(db,warren).              date(db,1981).
author(sasl,turner).            date(sasl,1979).
author(xform,standish).

title(db,efficient_processing_of_interactive...).
title(df,an_asynchronous_programming_language...).
title(eft,value_conflicts_and_social_choice...).
title(fp,can_programming_be_liberated...).
title(pro,dec_10_prolog_user_manual).
title(sasl,a_new_implementation_technique...).
title(sem,the_semantics_of_predicate_logic...).
title(xform,irvine_program_transformation_catalog).

loc(arvind,mit,1980).           journal(fp,cacm).
loc(backus,ibm,1978).           journal(sasl,spe).
loc(kling,uci,1978).            journal(kling,cacm).
loc(pereira,lisbon,1978).       journal(sem,jacm).
loc(vanemden,waterloo,1980).
loc(turner,kent,1981).          tr(db,edinburgh).
loc(warren,edinburgh,1977).     tr(df,uci).
loc(warren,sri,1982).
```

In this program, most binary literals `p(X,Y)` *stand for "the* `p` *of* `X` *is* `Y`*,"* e.g. `author(fp,backus)` *stands for "the author of the FP paper is Backus."*

Figure 2.2: An Example of a Logic Program

sertion is a unique key identifying an object of the domain modeled by the database. In the example program, the assertion `author(db,warren)` represents the fact that David H. D. Warren wrote a paper identified as `db`, and `date(db,1981)` says the date of the same paper is 1981.

Implications represent "derived data" – information not explicitly stored in a database, but computable from the data and programs stored in the system. Some examples of queries to this database, using the paradigm of typing a goal statement and then having the system prove the goal and print the bindings of the variables, are:

← `author(X,warren).`

This goal is "is there a value for X such that `author(X,warren)` is true?" In other words, is there a paper written by Warren? The system can prove this, and in doing so it binds X to the term `db` and prints it.

← `author(db,X).`

Similar to the above, but asks "who wrote the paper identified by the key `db`?" The system will print `warren`.

← `author(X,warren)` ∧ `date(X,D).`

"When did Warren write a paper?" This query is a conjunction of two goals; the system must prove both in order to solve the goal statement, and the value of `X` has to be the same in each literal. The proof succeeds, with `X` = `db` and `D` = `1981`.

It is possible to write a set of clauses to implement a function. An n-ary function `f` can be represented in a logic program by an $(n+1)$-ary procedure with functor `f`. In a call to `f`, n of the arguments will be bound, providing the n input values to the function. The remaining argument position in the call is an unbound variable. As a result of the call, the variable will be bound to the value of the function. For example, consider the addition function of integer arithmetic. A procedure call of the form `sum(a,b,Z)` means "add the numbers represented by terms `a` and `b` and bind `Z` to the result." Techniques for representing numbers by terms and performing arithmetic are discussed later, when we will see how it is done in Prolog. When represented as relations, functions are invertible, *i.e.* the relation represents not only the function but also its inverse. For example, given a set of assertions of the form `sqr(X,Y)`, where each `Y` value is the square of the corresponding `X`, the square of `3` can be computed with the call `sqr(3,S)`, and the square root of `9` computed with the call `sqr(S,9)`.

Deterministic goals have exactly one output for each distinct combination of inputs. If there is more than one solution, the goal is *nondeterministic*. The first example above is nondeterministic if the database has many papers

written by Warren. Some systems print all possible answers to a query, while others print one result and wait for the user to ask for additional solutions.

Another natural application is parsing of sentences [70, 92]. The operations required to prove an implication are similar to those used to reduce a string using a grammar. In this application, the head of a clause corresponds to a nonterminal symbol, and the literals in the body correspond to the right hand side of the rule. An example of a top-level goal in this application is

```
← sentence("time flies",[],P).
```

This goal is a request to the system to prove the string is a sentence in the grammar. If the proof is successful, the system will bind the variable `P` to the parse tree for the sentence. The transformation from grammar rule to executable clause is very simple; the DEC-10 Prolog system has a built-in "read macro" to transform grammar rules into clauses. A grammar written as a logic program is effectively a parser for the language it describes.

Another application area is artificial intelligence. Numerous papers have been written on the effectiveness of Prolog for writing expert systems and other AI programs. Some of the advantages of Prolog for this style of programming are due to features it has in common with Lisp, Planner, and other AI languages; other features are unique to logic programming languages. A good overview of the components of an expert system and some opinions on how Prolog and related languages provide support for implementing these components can be found in a paper by Subrahmanyam [80].

2.2 Semantics

Van Emden and Kowalski were the first to assign a formal semantics to logic programs [32]. The denotation $\mathbf{D}(\mathtt{p})$ of an n-ary procedure `p` is a set of n-tuples of terms. This definition is similar to the definition of a relation, and in fact the denotation of a procedure is often called a relation. There are three ways of defining $\mathbf{D}$; all three methods define the same set.

$\mathbf{D}_1(\mathtt{p})$, the *operational* semantics of `p`, is defined to be the set of all tuples $\mathtt{<t}_1 \ldots \mathtt{t}_n\mathtt{>}$ such that the predicate $\mathtt{p(t}_1 \ldots \mathtt{t}_n\mathtt{)}$ is *provable*, given the clauses of the program as axioms. Implementations of logic programming systems use a constructive proof procedure to create the tuples of $\mathbf{D}_1$. An input goal statement, such as

```
← p(X,a).
```

is a request to prove `p(X,a)`. A constructive proof not only satisfies the request, it generates a set of terms T such that $\mathtt{p(x}_i\mathtt{,a)}$ is provable for any $\mathtt{x}_i$ belonging to T. In response to the above query, an interpreter would construct

$$\{<\mathtt{x}_i, \mathtt{a}> \mid \mathtt{x}_i \in T\}$$

This is the subset of $\mathbf{D}_1(\mathtt{p})$ where the atom `a` is the second term in the tuple.

In the *model-theoretic* semantics, the denotation $\mathbf{D}_2(\mathtt{p})$ is the set of all n-tuples $<\mathtt{t}_1 \ldots \mathtt{t}_n>$ such that $\mathtt{p}(\mathtt{t}_1 \ldots \mathtt{t}_n)$ is *true* with respect to a Herbrand model for the program. The *fixed point* semantics $\mathbf{D}_3(\mathtt{p})$ is derived from the program through a transformation that maps clauses into ground clauses, from which tuples of ground terms are formed. The definition of the Herbrand model, the nature of the transformation, and a proof that $\mathbf{D}_1 \equiv \mathbf{D}_2 \equiv \mathbf{D}_3$ can be found in the article by van Emden and Kowalski [32].

The operational semantics of a logic program is determined by the proof procedure embodied in the interpreter. For most logic programming systems, the proof procedure is based on *resolution*, a deductive inference method defined for formulas of first order predicate calculus written as clauses [76]. One step of the interpreter corresponds to one logical inference. The remainder of this section is a discussion of resolution; the next section takes up the subject of controlling the order of the inferences in an interpretation.

The formal definition of a clause is that it is a disjunction of literals. All variables in a clause are universally quantified, and the scope is restricted to the clause itself. Literals as we have defined them so far are *positive* literals. The negation of a literal `p(X)`, written $\neg$`p(X)`, is a *negative* literal. The resolution rule states that from two clauses

$$\mathtt{p}_1 \vee \ldots \vee \mathtt{p}_{n-1} \vee \mathtt{q} \vee \mathtt{p}_{n+1} \cdots$$
$$\mathtt{r}_1 \vee \ldots \vee \mathtt{r}_{m-1} \vee \neg\mathtt{q} \vee \mathtt{r}_{m+1} \ldots$$

each containing a literal `q`, positive in one clause and negative in the other, one can infer the new clause

$$\mathtt{p}_1 \vee \ldots \vee \mathtt{p}_{n-1} \vee \mathtt{p}_{n+1} \vee \ldots \vee \mathtt{r}_1 \vee \ldots \vee \mathtt{r}_{m-1} \vee \mathtt{r}_{m+1} \vee \ldots$$

The derived clause, known as a *resolvent*, contains copies of every literal from the original two clauses except `q` and $\neg$`q`.

As a prerequisite to forming the resolvent, the theorem prover has to attempt to *unify* the literals `q` and $\neg$`q`. Unification is a pattern matching operation. Two literals are unifiable if they are syntactically identical, or if variables in either can be replaced by terms in order to make them identical. The literals are identical if they have the same function symbol, the same arity, and the corresponding argument terms are identical. When unifying terms, a variable can be replaced by any other term, including another variable, as long as the replacement is consistent throughout the two literals. For example, `q(f(a))` and $\neg$`q(X)` can be unified, since when `X` replaces `f(a)` in `q(X)` the literals are both `q(f(a))`. The two input clauses must not have any variables in common. Since the scope of a variable is the clause that contains

it, it is an easy matter to rename the variables of one of the inputs so there are no name conflicts.

Unification is the operation that binds variables during a proof. The set of bindings created during a unification is known as a *substitution*. The notation {X/a} refers to a substitution where the variable X is bound to the term a. Substitutions are often identified by lower case Greek letters, such as θ. $C\theta$ is the clause obtained by *applying* substitution θ to clause C, *i.e.* replacing every variable of C on the left side of a binding in θ with the corresponding right side. After unifying the two literals, the theorem prover applies the substitution generated during unification to the remaining occurrences of the variables in the two input clauses in order to form the resolvent.

Summarizing the steps, in order to resolve two input clauses, find a literal that is positive in one clause and negative in the other; unify the literals; form a new clause by copying all other literals from both inputs, and apply the substitution to the new clause. As an example, given the two clauses

$$\texttt{p(X,1)} \vee \texttt{q(Y,X)} \vee \texttt{r(X,Z)}$$
$$\neg\texttt{p(0,W)} \vee \texttt{s(W)}$$

it is possible to unify p(X,a) and ¬p(0,W) with substitution {W/1, X/0}. The resolvent is

$$\texttt{q(Y,0)} \vee \texttt{r(0,Z)} \vee \texttt{s(1)}$$

Since the scope of a variable is the clause that contains it, the effect of a binding is confined to the resolvent; variables in the input clauses are not modified. Figure 2.3 shows many more examples of successful and unsuccessful resolutions.

A *Horn clause* is a clause containing at most one positive literal. Since

$$P \vee (\neg Q \vee \neg R) \equiv P \leftarrow Q \wedge R$$

a Horn clause such as

$$\neg\texttt{p(a,b)} \vee \texttt{q(X)} \vee \neg\texttt{r(X,f(a))}$$

can be written as

$$\texttt{q(X)} \leftarrow \texttt{p(a,b)} \wedge \texttt{r(X,f(a))}$$

which brings us back to some familiar syntax. The head of a clause written in this syntax is a positive literal, and literals in the body are negative literals. When written this way, it is also easy to see that resolution can be considered a generalization of the *modus ponens* rule of propositional logic: from $A \rightarrow B$ and A it is possible to infer B. Stated another way, if we know $A \rightarrow B$ and we want to prove B, we can do it by proving A.

Successful Resolutions

Input Clauses		Resolvent	Substitution
p(X) ∨ q(X)	¬p(a) ∨ r(Y)	q(a) ∨ r(Y)	{X/a}
p(X,a) ∨ q(X)	¬p(b,Y) ∨ r(Y)	q(b) ∨ r(a)	{X/b, Y/a}
p(X)	¬p(1)	□	{X/1}
p(X) ∨ q(X)	¬p(f(a,B))	q(f(a,B))	{X/f(a,B)}
p(X) ∨ q(X)	¬p(A) ∨ r(A)	q(X) ∨ r(X)	{A/X}
p(X) ∨ q(X)	¬p(A) ∨ r(X)†	q(X) ∨ r(X2)	{A/X}

† *Note:* X *in second clause renamed* X2 *before unification.*

Unsuccessful Resolutions

Input Clauses		Reason for Failure
p(X) ∨ q(X)	p(X) ∨ r(Y)	must unify a positive literal with a negative literal
p(X) ∨ q(Y)	¬r(X) ∨ s(Y)	no predicate symbols match
p(X,a) ∨ q(X)	¬p(b)	predicate symbols match, but literals do not have same arity
p(a)	¬p(b)	arguments not unifiable

Figure 2.3: Examples of Resolution

When all clauses are restricted to Horn clauses, a theorem prover knows that in order to perform a resolution, a literal from the body of one input clause can only be used in a unification with a literal in the head of another clause. This restriction will greatly simplify the control structure used in the proofs underlying the execution of logic programs.

A useful mnemonic for reading clauses is to move the implied universal quantifier of any variables that occur only in the body to the right of the implication, turning them into existential quantifiers. An implication

$$\texttt{p(X,Z)} \leftarrow \texttt{q(X,Y)} \wedge \texttt{r(Y,Z)}$$

is the same as

$$\forall \texttt{X} \forall \texttt{Z} : \texttt{p(X,Z)} \leftarrow \exists \texttt{Y} : \texttt{q(X,Y)} \wedge \texttt{r(Y,Z)}$$

which is read "for all X and Z, p(X,Z) if there exists a Y such that q(X,Y) and r(Y,Z)."

Often a unifying substitution requires the replacement of one variable by another, as in the following example:

$$\texttt{p(1,X)} \vee \texttt{q(1,X)}$$
$$\neg\texttt{q(1,Y)} \vee \texttt{r(2,Y)}$$

When unifying two unbound variables, one is bound to the other. Formally, it does not matter which one is bound. Either {`X/Y`} or {`Y/X`} is a unifying substitution in this case. The resolvent has one variable; one of these substitutions names it `X` and the other names it `Y`. Either is acceptable because the scope of the name is the resolvent, and neither name conflicts with variables from the original clause. As a practical matter, however, the handling of variable-variable bindings is important in both sequential and parallel logic programming systems. The representation of these bindings will have an impact on the efficiency of unification, and thus the entire system.

2.3 Control

A complete resolution proof of a clause C with respect to a set of axioms A is the derivation of the null clause from $\{\neg C \cup A\}$. In other words, it is a proof by contradiction: negate C and show the negation leads to a contradiction. Unification and substitution make the proof a constructive proof. After the null clause has been derived, it is possible to construct terms for the variables of the original clause by using the substitutions performed during the proof.

In a logic programming system, the set of axioms is the program, and the clause to prove is the goal statement typed in by the user. Since program clauses are Horn clauses, the heads of clauses are positive literals, and goal statements and clause bodies consist of all negative literals. An execution step consists of selecting a goal from a goal statement, finding a clause with a matching head, and constructing a new goal statement through resolution. The *control* component of a logic programming system decides which literal in a goal statement will be used in the next resolution and the program clause it will be resolved with. The simplest and most commonly used control strategy is presented in this section.

The basic steps in deriving goal statement G_{i+1} from the current goal G_i are as follows.

1. Select a literal L from G_i.
2. Find a clause C in the program such that the head of C can potentially be unified with L.
3. Rename the variables in C so there are no variables in common with G_i.

4. Do the unification, in the process creating a substitution θ. If unification fails, the proof fails, otherwise continue.

5. Create goal G_{i+1} by replacing L in G_i with the body of C and applying θ.

When the selected clause is a unit clause, the body does not have any literals, so G_{i+1} will have one less literal than G_i. G_{i+1} will be the null clause when G_i contains exactly one literal, and this literal is resolved with a unit clause.

The control decisions in this procedure are in steps 1 and 2. In step 1 we must choose a literal to resolve. The standard control mechanism always selects the first (leftmost) literal in the goal statement. In step 2 we find a candidate for unification. The standard technique is to search the program from top to bottom, using the clauses in the order they occur in the program.

Another part of the standard control mechanism deals with failed unifications. Instead of terminating the entire proof, the interpreter can *backtrack* to a previous choice and try another alternative. The standard control keeps a record of choices made in step 2, and when a failure occurs it backs up and tries another clause if one exists. Note that backing up requires the *unbinding* of variables; all bindings made since the last choice point are undone, resetting the state of the computation back to what it would have been if the new clause was chosen in the first place.

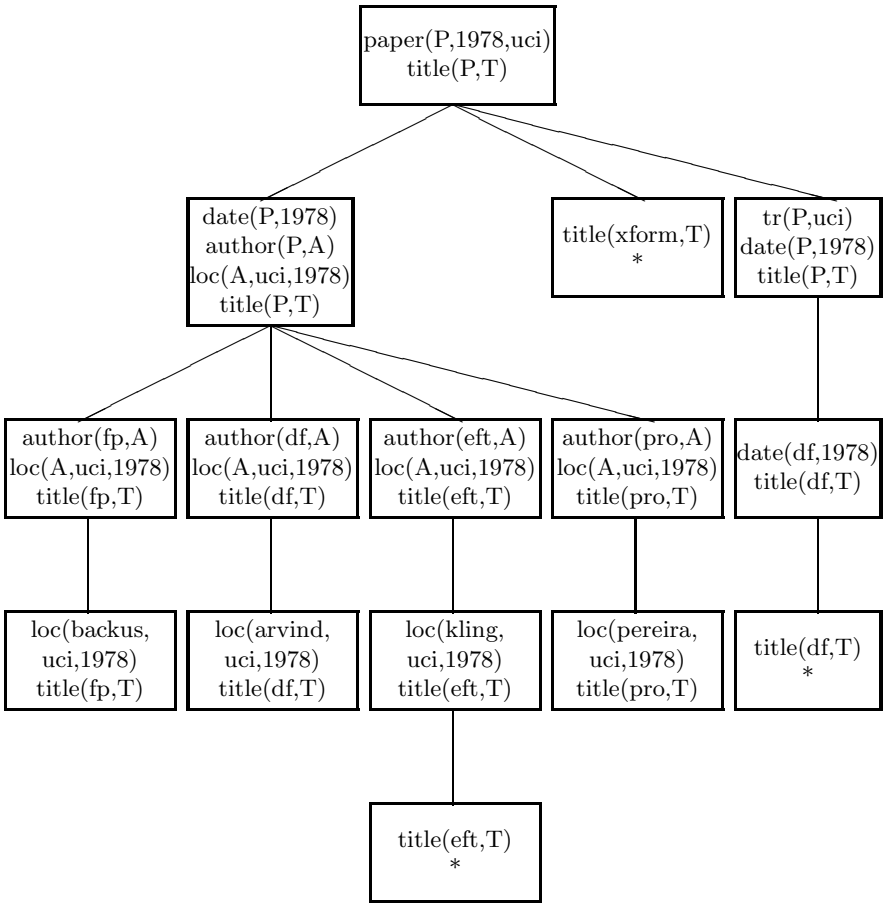
A *goal tree* is a tree where each node is a goal statement. Immediate descendants of a node N are goal statements derivable from N in one inference step. Steps in the execution of a logic program can be represented as a goal tree: the root of the tree is the goal typed in by the user, interior nodes are the intermediate goal statements, and the leaves of the tree are either null clauses or failure nodes. Figure 2.4 shows the goal tree formed from the initial goal

$\leftarrow$ `paper(P,1978,uci)` $\wedge$ `title(P,T).`

and the program of Figure 2.2, using the standard control.

The standard control strategy corresponds to a depth-first search of a goal tree. When the interpreter reaches a fail node, it backtracks to the most recent choice point. Note that if the goal tree contains an infinite branch, the standard control will not find all occurrences of the null clause; it will miss those to the right of the infinite branch.

A goal tree is not the easiest way to visualize execution of a logic program. It is introduced here because it represents explicitly the individual steps of the execution in the order they normally occur. It also presents a useful framework for describing different forms of parallel execution. The execution of a sequential logic program is easily explained in terms of the text of a



Goal tree generated during solution of `paper(P,1978,uci)` $\wedge$ `title(P,T)`. *Only the first literal within a node is used to generate descendant nodes. Leaf nodes marked with an asterisk represent successful branches; the nodes derived from these contain the null clause. Other leaf nodes are failure nodes.*

Figure 2.4: Goal Tree

program. The goals in a goal statement are solved in order, from left to right. A goal is solved by calling a procedure for the literal. Within a procedure, clauses are tried from top to bottom. If the head of a clause unifies with the goal literal, the unifying substitution is applied to the body of the clause and the goals in the body are solved from left to right. When the last goal is solved, the interpreter returns to the calling goal statement to solve the remaining goals. A more appropriate tree structure for visualizing the normal order of solution of goals might be an AND/OR tree, but explaining backtracking via an AND/OR tree is often difficult.

2.4 Prolog

The version of Prolog described here is DEC-10 Prolog [72]. This was one of the first implementations of the language, and certainly the most influential. It introduced the "Edinburgh syntax" and a set of built-in operations used in most other versions of the language.

The syntax of Prolog is similar to the syntax of pure logic programs used so far. The differences are:

- The implication symbol is `:-` instead of $\leftarrow$, and literals in the body of a clause are separated by commas, not the logic symbol $\wedge$.
- Lisp-like lists are allowed as data structures. Lists are enclosed in square brackets. The empty list is `[]`, and the list with `A` as first element and `B` as the tail is written `[A|B]`. Lists are complex terms according to the definitions of Section 2.1: `[a,b,c]` is simply shorthand for the term `.(a,.(b,.(c,[])))`, which has a period for the function symbol.
- It is possible to use infix notation for some terms and literals. For example, the term `+(X,Y)` can be written `X+Y`. There is a predefined set of infix functors and their precedence, and the set can be extended by the programmer.

Throughout the remainder of the book, we will continue to use the implication and conjunction symbols of logic in pure logic programs. If `:-` is used in a clause, it is because there is some reason to emphasize the fact that it is a Prolog clause as opposed to a logic program clause.

2.4.1 Evaluable Predicates and Arithmetic

In an earlier section, `sqr(X,Y)` was given as an example of a binary relation to implement the function $y = x^2$. $\mathbf{D}$(`sqr`) is an infinite set of tuples. Within the formalism of logic programming, there are two methods for doing arithmetic in cases such as this. The relation can be given explicitly, as a set of assertions

`:- X is 2+1.`	The expression is evaluated to `3`, and `X` is unified with `3`, thus binding `X` to `3`.
`:- 3 is 2+1.`	Since the value of the expression unifies with the constant `3`, the goal succeeds.
`:- Z is (2*3)+(4*5).`	`Z` is bound to `26`.
`:- 5 is 2+1.`	Fails, since `5` is not unifiable with `3`.
`:- X is 3*2, Y is X+3.`	Binds `Y` to `9`.
`:- 9 is X+3.`	Fails, since the second argument must be a ground term (a term containing no unbound variables).

Examples of goal statements in DEC-10 Prolog using the evaluable predicate `is`.

Figure 2.5: Examples of `is` in DEC-10 Prolog

of the form `sqr(a,b)`, and arithmetic operations will be essentially table searches. Obviously, the entire infinite relation cannot be stored, and the defined subset would consume a large amount of space. Alternatively, the relation can be computed, using a set of axioms of arithmetic. For example, the symbol `0` could represent the integer zero, the terms `s(0)`, `s(s(0))`, *etc.* could represent the positive integers, and we could write a set of clauses to be used in proving every integer has a square [54].

In Prolog, arithmetic is performed by metalogical *evaluable predicates* analogous to the built-in primitive functions of applicative languages. The evaluable predicates are metalogical because arithmetic is done be escaping from the system of resolution proofs and using another formal system, in this case the underlying machine hardware. We use the machine for arithmetic because it is much faster. In Prolog, numbers are a subset of the atoms, and the evaluable predicates implementing arithmetic operations are restricted to operating on terms representing numbers.

Arithmetic in DEC-10 Prolog is performed by the evaluable predicate `is`. This is a binary predicate that can be used as an infix operator. The second argument must be a legal arithmetic expression, constructed from the usual operators and integer terms, and the first argument can be either an integer or a variable. When `is` is called, the expression is evaluated, and the first argument is unified with the value of the expression. Some examples of Prolog goal statements with calls to `is` are in Figure 2.5.

The use of metalogical features such as evaluable predicates has an effect

on the semantics of programs. Goals with a defined meaning may not be solvable when the machine performs the operation. An example is

```
← 5 is X + Y.
```

In other words, what integers `X` and `Y` sum to `5`? This goal has six solutions in the positive integers, and all can be found if the system proves the statement using axioms of arithmetic or searches an explicit relation for tuples with `5` as the third element. However, the goal fails in DEC-10 Prolog when a machine is asked to add two uninstantiated variables.

The idea that a goal will fail if certain argument positions contain uninstantiated variables is expressed in DEC-10 Prolog by *I/O modes*. Each argument in a goal has one of three modes associated with it. In input-only mode, the argument must be a ground term. In output-only mode, the argument must be an uninstantiated variable; it will be bound in the solution of the goal. In the default, don't-care, mode, arguments can be instantiated or uninstantiated. When a Prolog procedure implements an n-ary function, n argument positions of the predicate symbol will be input-only mode, and the remaining argument positions can be either output-only or don't-care. Evaluable predicates in DEC-10 Prolog have modes for their arguments, and users are allowed to annotate their programs with mode declarations so the compiler can generate more efficient code. The concept of I/O modes will also be used to order goals for parallel solution in the AND/OR Process Model.

An alternative to mode declarations is the concept of *thresholds* [18, 51, 99]. Addition can be performed by a ternary evaluable predicate named `sum` having a threshold of two, meaning the goal is solvable if any two its three arguments are bound to non-variable terms. For example, `sum(X,3,5)` is solvable by binding `X` to `2`. `sum` will fail if fewer than two arguments are bound.

2.4.2 Higher Order Functions

DEC-10 Prolog allows implication and conjunction symbols to be used as function symbols in complex terms. This allows higher order functions: programmers can pass clauses as parameters to procedures and define new procedures dynamically [94]. The goal

```
:- assert(T).
```

adds the term `T` to the program currently in the system. The opposite of `assert` is `retract`:

```
:- retract(T).
```

finds a clause unifiable with `T` and then deletes it from the program.

The evaluable predicate `call` is similar to the `eval` function of Lisp. The goal

```
:- call(P).
```

treats the term `P` as if it were a goal statement, and calls the interpreter recursively to solve it. An evaluable predicate useful in conjunction with `call` is `=..`, which constructs terms from a list of components. The goal

```
:- T =.. [F|A].
```

succeeds if `T` is a term with function symbol `F` and argument list `A`. For example, the goal

```
:- T =.. [author,db,warren].
```

unifies `T` with the term `author(db,warren)`, and

```
:- p(X) =.. L.
```

unifies `L` with `[p,X]`.

An example using these higher order constructs is a Prolog procedure to implement the Lisp function `mapcar`:

```
mapcar(F,[],[]).
mapcar(F,[X1|Xn],[Y1|Yn]) :-
       Goal =..  [F,X1,Y1],
       call(Goal),
       mapcar(F,Xn,Yn).
```

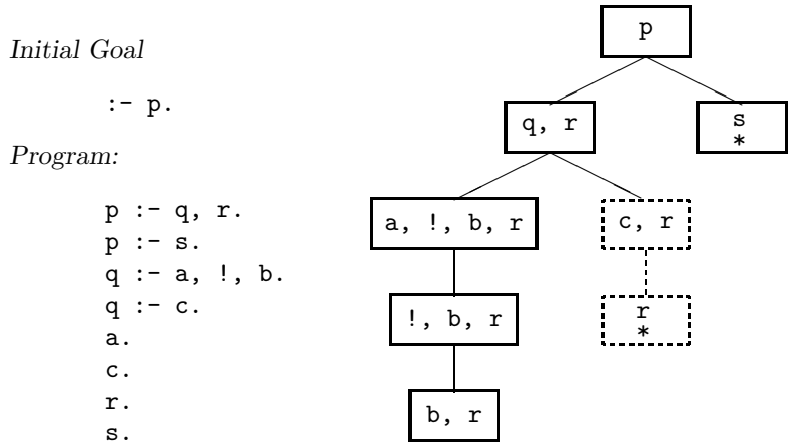
This procedure takes in a function symbol and list of arguments and returns a list built from applying the functions to each element of the input list. Assuming the existence of the procedure named `sqr` used in previous examples, the goal

```
:- mapcar(sqr,[1,2,3,4],L)
```

unifies `L` with the list `[1,4,9,16]`.

2.4.3 The Cut Symbol

The standard Prolog control has been described as a depth first search of a goal tree. The *cut symbol*, `!`, allows the programmer to control the search by pruning unwanted branches from the search tree. Cut is inserted into the body of a clause, along with other goals. When the interpreter encounters cut as a goal, it always succeeds. However, if the interpreter ever backtracks to the point where it has to re-solve the cut, the goal that called the clause



This figure shows the effect of the cut symbol (`!`*) on a depth first search. When the cut is executed, all choice points for the* head *of the clause containing the cut are discarded. In this example, the head of the clause containing the cut is* `q`*, so alternatives for* `q` *are discarded. The part of the tree in dotted lines is not searched.*

Figure 2.6: The Effect of "Cut"

containing the cut fails. The net effect is that all further solutions for literals to the left of the cut in the clause *and* all clauses in the same procedure following the clause with the cut are deleted from the goal tree.

As an example, refer to the small program in Figure 2.6, and observe what happens when a call is made to `p`. The first clause for `p` has a body with literals `q` and `r`, so the system starts to solve `q`. The first clause for `q` has body

```
a, !, b.
```

The call to `a` succeeds, but `b` fails. Now the interpreter backtracks, and encounters `!` while backtracking. The *head* of the clause containing the cut is `q`, so the call to `q` from `p` fails. The call to `q` was made from the first clause for `p`, and the failure of `q` forces the interpreter to move on to the second clause for `p`, where the program finally succeeds.

Common uses of the cut symbol are in finalizing choices from nondeterministic procedures and in the definitions of conditional expressions and negation.

Consider this small program with two different definitions for a procedure `p`:

```
p1(X) :- q(X).
p2(X) :- q(X), !.
q(a).
q(b).
```

When solving the goal

```
:- p1(X), continue.
```

the system eventually selects the first clause for `q` to solve `q(X)`, and `X` is bound to `a`. If `continue` fails, the system backtracks into `p1(X)`, `q(X)` is re-solved, and `X` will be bound to `b`. In solving the goal statement

```
:- p2(X), continue.
```

the first answer, `X = a`, is produced as before. When the system backtracks into `p2(X)` after `continue` fails, it encounters the cut symbol, so it does not retry `q(X)` and causes `p2(X)` to fail. Without the cut symbol, `p` is nondeterministic procedure, since there is more than one solution to a call to `p(X)`. With a cut symbol, this procedure becomes a deterministic procedure, producing one answer and then failing when asked to produce more answers. In general, a cut symbol as the last goal in a clause body indicates the clause and the procedure which it is a part of are deterministic. If the clause succeeds, it succeeds only once.

A conditional expression in a functional language has the general form

```
f(X) = if p(X) then g(X) else h(X)
```

If `p(X)` is true, the value of `f(X)` is given by `g(X)`, otherwise it is defined by `h(X)`. Recall that in Prolog, *n*-ary functions are defined by *(n+1)*-ary predicates. In Prolog, the above expression is written as two clauses:

```
f(X,Y) :- p(X), !, g(X,Y).
f(X,Y) :- h(X,Y).
```

When the function is called, for example by the goal

```
:- f(10,Y).
```

the interpreter selects the first clause for `f`, then calls `p(10)`. If `p(10)` succeeds, the value of `Y` is determined by the call to `g`. If `p(10)` fails, the interpreter backtracks to the second clause for `f`, and the value of `Y` is computed by the call `h(10,Y)`.

The cut symbol is necessary for those occasions when `g(X,Y)` fails after `p(X)` succeeds. The desired control behavior is that when `p(X)` is true, `f(X,Y)` is defined by `g(X,Y)`; this means that if `g(X,Y)` fails, `f(X,Y)` should

also fail. This situation is analogous to the definition of conditional expressions in FP, where `f` is undefined when `p` is true but `g` is undefined. We do not want the interpreter to backtrack to `h(X,Y)` when `p(X)` succeeds but `g(X,Y)` fails. The desired behavior is enforced by the cut symbol.

Cut is commonly used with `fail` (a goal which always fails) to negate the result of a call. The usual definition of `not` is:

```
not(G) :- call(G), !, fail.
not(G).
```

This is *negation as failure*, first defined by Clark [14]. Recall that an implication

```
p :- q, r.
```

is equivalent to the Horn clause

```
p ∨ ¬q ∨ ¬r.
```

and that literals of the body of a clause are actually negative literals. Thus one cannot simply write

```
:- ¬p(X).
```

for the request "prove p(X) is false," since a negated literal in the body of a clause would actually be a positive literal, and by definition a goal statement must contain only negative literals. A resolution theorem prover works for clauses containing more the one positive literal, but the control strategies of Prolog are based on the assumption that the head is the only positive literal in the clause.

Negation as failure means that if one fails to prove a statement, it can be assumed the statement is false. Operationally, an implementation of negation as failure means that if a call to `G` succeeds, `not(G)` should fail, but if `G` fails, `not(G)` should succeed. Referring to the above Prolog definition, when `not(G)` is called, the interpreter first tries to solve `G`, via the goal `call(G)`. If this fails, then the second clause for `not(G)` is tried, and since this is a unit clause, it succeeds. In other words, when `G` fails, `not(G)` succeeds.

In the other case, when `call(G)` succeeds, the interpreter moves on to the cut and `fail` literals. Cut succeeds, and `fail` fails. Because the cut is there, the interpreter does not try to solve `G` again, and in addition causes the failure of `not(G)`, the head of the clause with the cut. Thus when `G` succeeds, `not(G)` fails.

In logic, there is a major difference between the statements "P is false" and "P cannot be proven," especially when higher order functions are introduced into the system. There are also practical problems in Prolog systems that use this definition of negation. These problems arise when the negated goal

contains variables. Consider a procedure with only one clause, the assertion `p(t)`. The call `p(t)` will succeed, and the call `p(f)` will not. `not(p(t))` and `not(p(f))` behave as expected – `not(p(t))` fails and `not(p(f))` succeeds. But notice what happens with the goal `not(p(X))`: the system solves `p(X)` by binding `X` to `t`, then executing cut and fail, so `not(p(X))` fails. But it was just shown that `not(p(X))` succeeds when `X` is bound to `f`, so one can argue the system should actually succeed in solving `not(p(X))` by binding `X` to `f`. Thus it is not clear at all whether `not(p(X))` should succeed or fail, and if it should succeed, how to construct the set of legal values for `X` – there are an infinite number of substitutions for `X` to make `p(X)` fail. The general rule for using negation as failure is to make sure the argument to `not` is a goal with only ground terms as arguments. In spite of these shortcomings, negation as failure is used effectively in many logic programs and logic databases [24]. Negation as failure will be defined for parallel systems, without using the cut symbol, in Chapter 4.

2.5 Alternate Control Strategies

Control in a logic program has been characterized in the previous sections as search of a tree of goal statements. The object of the search is a null clause at the end of a sequence of resolutions. The unifications used on the path from the starting goal statement at the root of the tree to a null clause define an n-tuple of values for the n variables of the starting goal statement. If there is more than one way of solving the original goal, there will be a number of null clauses at the leaves of the tree, with an n-tuple defined by each path.

The denotation of a procedure is a relation, an unordered set of tuples of terms. Ideally, a control strategy helps an interpreter construct every tuple in the relation if necessary. In practice, however, a given control method may not be able to order the required resolutions so that all tuples are constructed. In particular, a depth-first interpreter never terminates when there is an infinite branch in the search tree; this control method will never construct any tuples defined by finite branches to the right of an infinite branch.

The meaning of a procedure is independent of the control mechanism. The meaning is a relation, an unordered set of tuples, and a control mechanism merely defines an order in which those tuples are created. Some alternatives to the standard depth first search optimize the search by decreasing the size of the search space. Others generate a larger set of answers by working around infinite branches or growing branches in parallel. Four techniques used in sequential systems will be discussed in this section. Although the mechanisms explained here are defined in terms of a sequential search of a single goal space, some of the principles illustrated have been used in the definition of parallel control, surveyed in the next chapter. The first three

methods select literals other than the leftmost as the literal to expand. The fourth is intelligent backtracking, a more effective method for backtracking that prunes portions of the search tree that cannot contain solutions.

2.5.1 Selection by Number of Solutions

In general, a tree search is more efficient when the branching factor in the tree is smaller. If a search algorithm can expand the nodes that generate fewer descendants, it might save itself needless work by traversing fewer unsuccessful branches.

Consider a program containing two procedures, `p(X)` and `q(X)`. In the following discussion, let n_p be $|\mathbf{D}_1(\mathtt{p})|$ the (finite) number of solutions for `p`, and n_q be defined the same way for `q`. The n_{pq} terms in solutions of both `p` and `q` are the terms constructed in response to the goal

$$\leftarrow \mathtt{p(X)} \wedge \mathtt{q(X)}.$$

Solving either literal by itself will bind `X` to a term. If the interpreter selects `p(X)` from this goal statement, there will be n_p descendant search trees, each with a root of the form `q(a)`, where `a` is one of the solutions for `p`. The remaining steps consist of a search through the descendants looking for one of the n_{pq} occurrences of the null clause. On the other hand, if the interpreter selects `q(X)` as the literal to resolve from the above goal statement, there will be n_q descendants, with root nodes `p(b)`, where `b` is constructed in the solution of `q`. There are still exactly n_{pq} null clauses. Whether the interpreter must generate all answers (*i.e.* find all null clauses) or just one (*i.e.* find the leftmost null clause), the efficiency of the search is determined by the proportion of the number of null clauses to the number of branches, and this proportion is better when fewer branches are generated. Thus when it is known in advance that n_q is less than n_p, the interpreter should select `q(X)` for resolution first, regardless of whether it is the leftmost literal in the goal statement. The general principle is to select the goal that most constrains the remainder of the search.

Again, it is important to note that the order of selection of literals affects the efficiency of the search, and possibly the order in which the answers are reported, but not the final result. The paths ending with a null clause in either tree lead to the construction of the same set of values for `X`.

This general strategy, of selecting literals known beforehand to have the fewest number of solutions, is used to optimize queries in the CHAT-80 relational database system [93, 96]. In a database, where the arguments of assertions consist of ground literals, search for a clause with a unifiable head is augmented by hashing on the functor of the literal and the value of the first argument term, so decreasing the size of the search space means decreasing the number of unifications required in the next step.

2.5.2 Selection by Number of Uninstantiated Variables

It may be possible to limit the size of the search space even when the interpreter does not have prior information about the number of solutions for each literal, by assuming that literals with fewer uninstantiated variables will generate fewer branches.

Consider a nondeterministic procedure `p(X,Y)` that has n results when `p` is called with two unbound variables for parameters. If one variable is bound when `p` is called, there will generally be fewer results returned, since the solutions are constrained to be the subset of the original n results having a matching term as the corresponding input argument. If both arguments are bound when `p` is called, there are fewer results still – there is just one response, a yes/no answer. In general, but not always, the branching factor at a node decreases with the number of variables bound in the literal used to expand the node. This observation can form the basis of a number of heuristics for selecting the literal from the current goal statement.

Consider a clause based on the procedures of the program in Figure 2.2:

```
query(P,I) ← author(P,X) ∧ loc(X,I,D).
```

Given a depth first interpreter and an initial goal statement

```
← query(eft,I).
```

the derived goal statement will be

```
← author(eft,X) ∧ loc(X,I,D).
```

There is just one way to solve the first subgoal, and the solution binds `X` to a term that leads to only one solution for the second subgoal; the final answer has `X` bound to `kling`, `I` to `uci`, and `D` to `1978`.

On the other hand, if the initial goal statement is

```
← query(P,uci).
```

the derived goal statement is

```
← author(P,X) ∧ loc(X,uci,D).
```

The result of this call is the same as the previous one, with `P` bound to `eft` in the only solution. There are eight ways to solve the first subgoal, since any of the unit clauses in the procedure for `author` are unifiable when no arguments are bound in the call, but only one of those unifications leads to a solution for the second subgoal. An interpreter using the heuristic of selecting the goal with the highest percentage of its variables bound would first solve the rightmost subgoal, `loc(X,uci,D)` in this derived clause. It would find just one solution for that goal, leading immediately to a solution for the leftmost

literal. In other words, the number of misleading branches can be reduced from seven to zero by selecting a literal with one instantiated variable instead of zero instantiated variables.

The relationship between search efficiency and the pattern of bindings on variables was mentioned by Kowalski [54]. The IC-Prolog interpreter allows users to write a number of versions of the same clause, and then annotate these clauses so the interpreter selects the most efficient one at runtime, depending on the pattern of variable instantiation in a goal [18]. A related strategy will also be used to order literals for parallel solution, described in Chapter 6.

2.5.3 Intelligent Backtracking

Consider the set of unit clauses

```
p(a).
p(b).
q(1).
q(2).
r(b,1).
r(b,2).
```

Given the goal statement

$\leftarrow$ `p(X)` $\wedge$ `q(Y)` $\wedge$ `r(X,Y).`

a depth first interpreter first solves `p(X)`, binding `X` to `a`, then solves `q(Y)`, binding `Y` to `1`, and then tries to solve `r(a,1)`. When the latter fails, the interpreter backtracks. The most recent choice point is in the selection of the clause for solving `q(Y)`; when this is redone, another solution is found, binding `Y` to `2`, and the next goal is `r(a,2)`, which also fails.

Both of these calls to `r` fail because the solution of `p(X)` binds `X` to a value that cannot be used to solve `r(X,Y)`. When the interpreter backs up only as far as `q(Y)`, it cannot fix this erroneous choice, and by re-solving `q(Y)` and binding `Y` to a different value it is wasting time.

An interpreter incorporating *intelligent backtracking* analyzes the cause of a failure, and backtracks to the source of values causing the failure [6]. An interpreter designed and implemented by Pereira and Porto performs this kind of analysis [73, 74]. In the example given above, it finds that any goal of the form `r(a,X)` fails because of the presence of the term `a` in the first argument position. Since `X` was bound to `a` in the call to `p(X)`, the interpreter backs up past the call to `q(X)`, all the way to a choice point in the solution of `p(X)`. When `p(X)` is solved again, binding `X` to `b` this time, the entire goal list can be solved, without the wasteful attempt to solve `r(a,2)`.

Other cases where intelligent backtracking can be helpful are in goals such as

```
← p(A) ∧ q(B) ∧ r(A).
```

When `r(A)` fails, `q(B)` can be skipped on backtracking since it does not produce any values that can affect the solution of `r(A)`. This is a case where it is not necessary to analyze the exact cause of the failure; it is only necessary to notice that a new solution of `q(B)` cannot help solve `r(A)`, since `r(A)` and `q(B)` have no variables in common [19]. This has been called semi-intelligent backtracking, since it is not quite as effective as intelligent backtracking. For example, a semi-intelligent backtrack scheme would not make the correct backtracking choice for the first example in this section. This limited form of intelligent backtracking will be seen in the parallel control described in Chapter 6.

2.5.4 Coroutines

Consider these definitions of the procedures `concat` and `sqrs`:

```
concat([],List,List).
concat([Car|Cdr],L1,[Car|L2]) ← concat(Cdr,L1,L2).
sqrs([],[]).
sqrs([X1|Xn],[Y1|Yn]) ← Y1 is X1*X1 ∧ sqrs(Xn,Yn).
```

The goal

```
← concat([1,2],[3,4],L) ∧ sqrs(L,S).
```

is a request to construct a list `L` by concatenating lists `[1,2]` and `[3,4]`, and a list `S` such that every element of `S` is the square of the corresponding element of `L`. The only solution in the goal tree with this goal at the root gives the answers `L = [1,2,3,4]` and `S = [1,4,9,16]`. After the first step in the computation, the derived goal statement is

```
← concat([2],[3,4],L1) ∧ sqrs([1|L1],S).
```

The variable `L` from the original goal has been bound to the term `[1|L1]`, where `L1` is a new variable. Bindings such as these, where a variable is bound to a complex term containing unbound variables, are *partial bindings*. The top level structure of `L` is known, but values of component substructures have not been determined yet.

The normal depth first control completely solves the call to `concat`, binding `L` to `[1,2,3,4]`, before the solution of `sqrs` is started. A *coroutine* control interleaves the steps in the solutions, by having `concat` make a "piece" of the list `L` through a partial binding, and then having `sqrs` use this piece. The

concat([1,2],[3,4],L) ∧ sqrs(L,S).	L = [1|L1]
concat([2],[3,4],L1) ∧ sqrs([1|L1],S).	L = [1,2|L2]
concat([],[3,4],L2) ∧ sqrs([1,2|L2],S).	L = [1,2,3,4]
sqrs([1,2,3,4],S).	S = [X1|S1]
X1 is 1*1 ∧ sqrs([2,3,4],S1).	S = [1|S1]
sqrs([2,3,4],S1)	S = [1,X2|S2]
X2 is 2*2 ∧ sqrs([3,4],S2).	S = [1,4|S2]
sqrs([3,4],S2).	S = [1,4,X3|S3]
X3 is 3*3 ∧ sqrs([4],S3).	S = [1,4,9|S3]
sqrs([4],S3).	S = [1,4,9,X4|S4]
X4 is 4*4 ∧ sqrs([],S4).	S = [1,4,9,16|S4]
sqrs([],S4).	S = [1,4,9,16]
□	

concat([1,2],[3,4],L) ∧ sqrs(L,S).	L = [1|L1]
concat([2],[3,4],L1) ∧ sqrs([1|L1],S).	S = [X1|S1]
concat([2],[3,4],L1) ∧ X1 is 1*1 ∧ sqrs(L1,S1).	S = [1|S1]
concat([2],[3,4],L1) ∧ sqrs(L1,S1).	L = [1,2|L2]
concat([],[3,4],L2) ∧ sqrs([2|L2],S2).	S = [1,X2|S2]
concat([],[3,4],L2) ∧ X2 is 2*2 ∧ sqrs(L2,S2).	S = [1,4|S2]
concat([],[3,4],L2) ∧ sqrs(L2,S2).	L = [1,2,3,4]
sqrs([3,4],S2).	S = [1,4,X3|S3]
X3 is 3*3 ∧ sqrs([4],S3).	S = [1,4,9|S3]
sqrs([4],S3).	S = [1,4,9,X4|S4]
X4 is 4*4 ∧ sqrs([],S4).	S = [1,4,9,16|S4]
sqrs([],S4).	S = [1,4,9,16]
□	

The first column shows a goal list, the second the bindings for `L` *or* `S` *after a derivation based on the goal list.* `L1`*,* `L2`*, etc. are new instances of* `L` *created in recursive calls. The first sequence of derivations is from a depth-first interpreter, where the derivation is based on the leftmost literal in the goal. The second sequence of derivations is made by a coroutine interpreter, using* `concat` *as producer and* `sqrs` *as consumer. The literal selected for reduction at each step is underlined.*

Figure 2.7: Coroutine *vs.* **Depth-First Control**

literal `concat([1,2],[3,4],L)` is called the *producer* of `L`, `sqrs(L,S)` is a *consumer* of `L`, and the variable `L` is a *communication channel* between the two literals.

The series of derivations made for the above example by a coroutine control is shown in Figure 2.7. Successive goal statements are derived until a partial binding is created for `L`. At that point, the consumer literal is selected, and derivations continue until a call to the consumer sees an uninstantiated variable in the argument position for `L`. Then the producer is selected, another piece of `L` is created, and the consumer is resumed. To summarize, a depth first interpreter creates the entire list `L`, and then calls `sqrs` to square every element in `L`, making `S`. The coroutine interpreter interleaves the interpretation of the two calls, creating and squaring the first element, then creating and squaring the second element, until the last element has been squared. Steps in the solution of the consumer are executed until the input channel is empty, *i.e.* the term is an unbound variable. Then steps in the solution of the producer are executed until a partial binding is made for the communication channel, and execution switches back to the consumer.

An interesting use of coroutines is in the definition of infinite data structures. The clause

```
inf(N,[N|L]) ← M is N + 1 ∧ inf(M,L).
```

describes an infinite list of integers. In the goal

```
← inf(1,X) ∧ consume(X,Y).
```

the call to `inf(1,X)` unifies `X` with the infinite list of integers starting with `1`, and `consume(X,Y)` uses the elements of the list to compute `Y`. The call to `inf` results in an infinite loop when an attempt is made to solve it with a depth first interpreter, since the interpreter tries to create the entire list of integers starting from 1. A coroutine interpreter would create the sequence of integers only up to the last integer required by the call to `consume`.

Coroutines were implemented in IC-Prolog [18], where users annotate literals in a clause to indicate producer/consumer relationships. Infinite data structures are used in many elegant programs written in SASL [86] and other applicative programming languages.

2.6 Chapter Summary

This chapter presented logic programming as a formal model of computation. The syntax is a subset of first order predicate calculus. The meaning of a procedure in the denotational semantics is a relation, a set of tuples of terms. The operational semantics is defined by resolution, a constructive proof procedure that creates tuples of the relation in order to provide values for variables of a goal statement. Control was shown to be important for efficiency: it may have an effect on the order in which answers are found, or the number of steps executed in deriving an answer, but any results generated are tuples of the relation. Some control strategies may not be complete, since some computable results are not generated. A number of interesting alternatives to the simplest and most common control method were discussed. These alternatives, all defined for sequential systems, illustrate some of the techniques used in parallel systems.

Resolution and unification were first defined by Robinson for use in automatic theorem proving [76]. An overview of first order predicate calculus, an algorithm for transforming well formed formulae into sets of clauses, and a discussion of resolution can be found in Nilsson's book [68]. Martelli and Montanari presented three efficient unification algorithms, and discussed the *occur check*, a component of unification which is essential for logical correctness but usually omitted from logic programming systems for reasons of efficiency [63].

DEC-10 Prolog and its successors are the most widely used implementations of Prolog [72]. It was the first to use a compiler to generate machine code. Compiled DEC-10 Prolog programs are comparable in speed to compiled Lisp programs [91, 97]. Building on this work, Warren described an abstract machine for the target code of a compiler; the abstract machine can be the basis for a byte-code interpreter or implemented in microprogram [95].

Luis Pereira and his colleagues have written a number of papers on the subject of control in logic programs. In addition to the work on intelligent backtracking, there is Epilog, a language that allows programmers to define special-purpose control constructs for different situations [71].

Two logic programming languages not based on the resolution rule are LPL, described in Haridi's thesis [40], and Relational Programming (MacLennan [61]). Haridi's system is based on a natural deduction proof procedure, and supports the negation and if-and-only-if constructions that are difficult to express with Horn clauses and depth-first search. In the relational programming system, entire relations are computed at the same time, and operations are performed on relations as a whole, instead of on individual tuples within the relation.

A number of large and useful applications have been written in Prolog. Among these applications are the natural language query processor of the

CHAT relational database system [24, 93, 96], Warren's problem solving program [89], and Kibler and Porter's episodic learning program [53]. Expert systems written in Prolog are described by Subrahmanyam [80]. The use of Prolog as a metacompiler was described by Warren [92], and a comparison of definite clause grammars and augmented transition networks for processing natural language was given in the paper by Pereira and Warren [70].

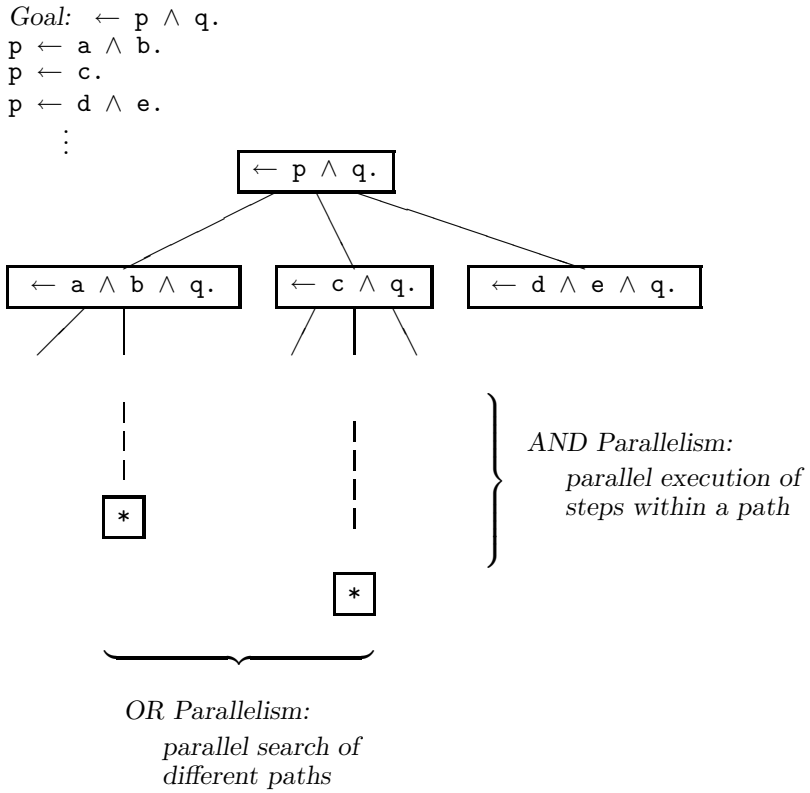
Chapter 3

Parallelism in Logic Programs

Execution of a logic program begins when the user provides an initial goal statement. The interpreter computes values for variables of the goal through a series of resolutions. If the null clause can be derived, the substitutions used in the derivation provide values for the variables of the initial goal. The execution can be represented as a goal tree, where multiple descendants of a node indicate a choice of clauses for resolving the goal at that node.

The two major forms of parallelism in logic programs can be explained in terms of speeding up the search of the goal tree (Figure 3.1). *OR parallelism* refers to a parallel search strategy – when a search process reaches a branch in the tree, it can start parallel processes to search each descendant branch. *AND parallelism* corresponds to parallel construction of a branch – when the interpreter knows a number of steps must be done to complete a branch, it can start parallel processes to perform those steps. The name for OR parallelism comes from the fact that in a nondeterministic program, we are often satisfied with any answer. When any of the processes started at a choice point finds a result, the original goal is solved. The name for AND parallelism is based on the fact that all steps must succeed in order for a result to be produced.

Another way to explain the distinction is in terms of the denotation of a goal statement. An interpreter is expected to produce a set of tuples of terms, where each tuple has a value for each variable of the initial goal. OR parallelism is a technique for parallel construction of the different tuples in the denotation of a nondeterministic goal, while AND parallelism is parallelism in deriving any one tuple. In an OR-parallel interpreter, each element is constructed sequentially, using the same sequence of operations performed



A computation step in a logic program is the derivation of a new goal from an existing goal and a program clause. A solution corresponds to the steps from the initial goal to a null clause at a leaf node, shown here as a node marked with an asterisk.

Figure 3.1: AND Parallelism *vs.* **OR Parallelism**

by a sequential interpreter. A deterministic goal statement has exactly one solution; for deterministic goals an OR-parallel interpreter finds an answer no faster than a sequential interpreter which makes perfect decisions at each choice point. An AND-parallel interpreter has the potential to speed up deterministic computations, since it executes in parallel the steps needed to construct the result.

Explained in terms of the structure of a program, OR parallelism is the parallelism obtained from parallel execution of the different clauses of a procedure. AND parallelism is the parallelism obtained from parallel execution of the goals in the body of a clause. AND and OR parallelism were originally defined this way by Conery and Kibler [21].

A third source of parallelism described in this survey is parallelism in low level operations, such as unification. Systems exploiting parallelism at this level are typically sequential interpreters with respect to the global view of the computation, often executing exactly the same sequence of derivations as a Prolog interpreter.

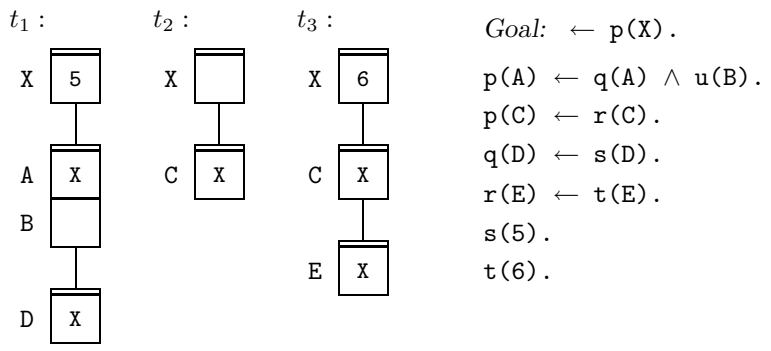
3.1 Models for OR Parallelism

Abstract models for OR parallelism fall into three broad categories. The first, called "pure" OR parallelism, consists of a parallel search of the goal tree. The earliest system of this type was designed by Haridi and Ciepielewski [11]. When a searching process comes to a choice point, it can fork new processes for each alternative. Processes proceed from that point with very little interaction. If a process cannot unify its first literal with any program clause, it simply terminates; if it derives the null clause, it announces the result and then terminates. There is no need to send any information to another active search process.

The second form of OR parallelism is based on objects called *OR processes*. In this model, a process corresponds to an object of Actors or Smalltalk [36, 41]. Each process is responsible for executing a small piece of the program. It maintains local state information, and communicates with other objects via messages.

The third form, called *search parallelism* by Conery and Kibler [21], is based on a physical partitioning of the program. Clauses are stored in different nodes of a machine such as a multiprocessor database machine. The common aspect of the models in this category is that unification is done at the nodes where the clauses are stored, while the derivation of resolvents and overall control decisions are carried out by a different process.

Two major issues in the exploitation of OR parallelism are the combinatorial explosion in the number of processes and representation of variable binding environments. Nondeterministic logic programs may have too much



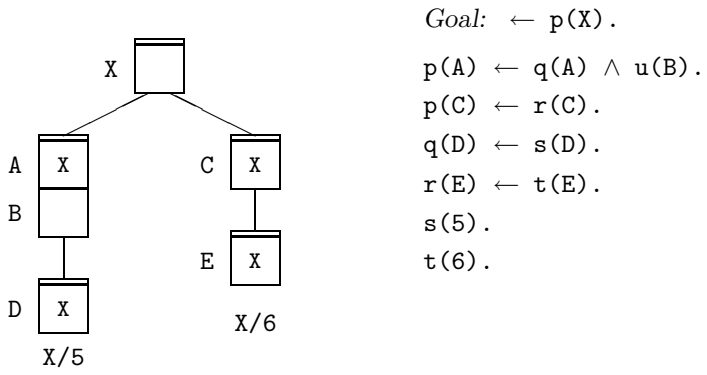
When a clause is called, an environment is pushed on the stack (which grows down the page). Environments are not popped until the clause fails. The first snapshot, on the left, shows the stack just after the call to `s(D)` *and before* `u(B)` *is attempted. Since* `D` *was unified with* `X`, `X` *was bound to* `5` *by the call to* `s(D)`. *At* t_2 `u(B)` *has failed and* `q(A)` *failed on retry, and* `r(C)` *is about to be called; note* `X` *is an unbound variable again. The right stack is the final stack, after the goal succeeds.*

Figure 3.2: Environment Stack for Sequential Prolog

parallelism. The state space for game trees and other search-oriented AI problems can easily grow so large that systems will create too many parallel processes. When physical processors work on more than one logical process at a time, some amount of task switching is to be expected, but when the number of tasks is very much larger than the number of processors, the overhead of task management will have an adverse effect on the system.

The problem in managing binding environments is illustrated in Figures 3.2 and 3.3. Figure 3.2 is a greatly simplified representation of how variables are stored in a Prolog program. Each time a clause is used, an environment containing slots for each variable of the clause is pushed on a stack, in much the same way stack frames are created for Pascal procedures. Unlike Pascal, however, the frames are not popped when the Prolog procedure exits. The frames represent substitutions, and the system needs the values assigned in substitutions when solving the remaining goals in the statement from which the clause was called. Environments are popped from the stack when the clause fails.

It would be tempting to implement binding environments for parallel processes by allowing new processes at a choice point to share all the previous



In an OR-parallel system, a new process is started at a choice point in the program, such as the call to `p(X)` *here. If new processes share the existing stack, each needs its own copy of variables which are unbound at the time of the fork. In this example, the processes generate conflicting bindings for* `X`.

Figure 3.3: Environment Stack in OR-Parallel System

values in the stack. However, as Figure 3.3 shows, unbound variables in the older stack frames pose a problem, since each new process has to be able to bind these variables in unifications in its own branch of the tree.

3.1.1 Pure OR Parallelism

The problem of efficient memory representation for OR-parallel systems has been attacked by a number of researchers. The first to address the problem were Ciepielewski and Haridi, for the OR-Parallel Token Machine. In their technique, a directory is associated with each process, containing pointers to stack frames for the process [12]. Frames containing no unbound variables may be shared among many processes, while others are copied for each new process.

A technique developed by Borgwardt for an AND-parallel system has also been used in pure OR-parallel systems. In this representation, a "hash window" is used to find values of variables in older frames [3]. If a reference is made to a variable in an older frame, and the variable is bound, its value can be used, since the binding was made before the processes were split. If the variable is unbound, the system checks the hash window of the current frame (and possibly other frames on the path back to the older frame) to see if it was bound since the time of the fork.

In Lindstrom's method, variables that are unbound in the frame of the

calling process are imported into the frame of each called clause [58]. The environments of the called clauses are extended to contain slots for each unbound variable in the parent process. A similar technique devised by D. S. Warren extends the local environment at the time an older variable would have been bound. Warren also describes an optimization for this technique to give each process fast access to its copy of old variables [98]. Directory trees, hash windows, and Lindstrom's variable importation scheme were compared in a study by Crammond [22]. Crammond designed a simple virtual machine for OR-parallel execution and used it to measure the relative amount of memory and processor resources used for the three methods.

What the above techniques all have in common is a mechanism for preventing the binding of shared variables, through the use of auxiliary structures to identify shared variables and indicate where local copies are kept. A technique called *kabu-wake*[1] allows older variables to be bound; if, later, one of these variables needs to be shared by parallel processes, each process will get its own copy. In this method, when a choice point is reached, alternatives are stored in a list of potential processes, but the new processes are not started right away. The main process continues, binding older variables and keeping trail information, as in a Prolog interpreter, without regard to the fact that blocked processes will later need to see these as unbound variables. When a blocked task is finally started, it copies as much of the current stack as it needs, and then, using trail information, simulates backtracking to unbind the variables bound by the other process since the choice point. The net result is the same, since the new process has its own copy of shared variables, but the copy is made later, in a lazy fashion. This technique was used in two different systems, ORBIT (Yasuhara and Nitadori [102]) and the system of Kumon *et al*[55].

One of the ways to control the explosive growth in the number of parallel processes in OR parallelism is via the syntax of the program. In Parlog, clauses in a procedure can be terminated by semicolons or periods [17]. If a period separates the clauses, they are executed in parallel when called from an OR-parallel goal. A semicolon separating two clauses means all solutions from the clauses before the semicolon are found before starting processes to find solutions to clauses after the semicolon. For example, in the procedure

```
p ← q.
p ← r;
p ← s.
```

all solutions for `p` based on parallel calls to `q` and `r` are found first. After processes for those clauses terminate, a process will be started for `s`.

[1] A Japanese term for a technique of splitting a tree at the root for transplanting.

Ciepielewski and Haridi examined a method for pruning unnecessary branches in the search tree when the user expects just one result [13]. This technique can be viewed as a parallel analogue of cut, where all other candidate solutions for a goal are discarded as soon as one result is found.

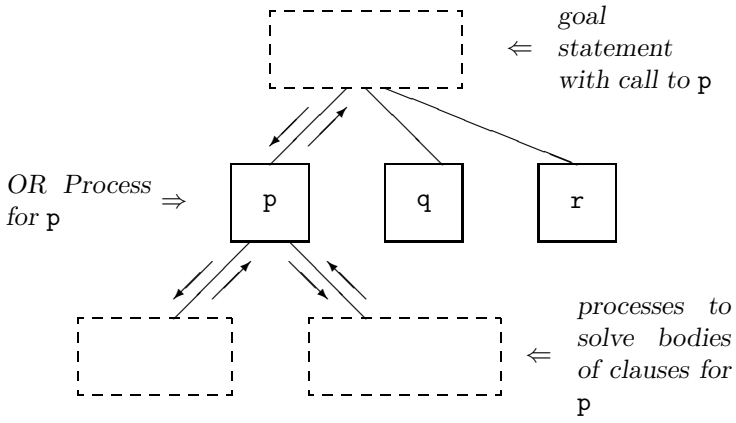
Other approaches to growth control in pure OR-parallel systems are based on directing the search from a global perspective. For the B-Log system, Lipovski and Hermenegildo investigated alternative search strategies [60]. They give a graph-oriented representation for goals, and rules for selecting a literal from a goal statement in order to expand the next level of the search tree. Instead of selecting the leftmost literal, B-Log selects the "best" literal, using heuristic information about the probability of a success at the end of a branch created with the literal. The heuristic information is global, allowing the search to vary from depth-first to breadth-first as the situation demands.

Another model based on varying the search strategy in order to control the number of processes was proposed by Li and Wah [56]. Li and Wah use heuristics to balance the cost of expanding a node with the probability of success through descendants of the node. The probabilities are obtained from prior executions of the program.

A general framework for parallel search of an abstract tree was presented by Bowen [5]. The search process uses an operator named `sons` to generate descendants of a given node in the tree, and `compsof` to break a node into component parts. If the nodes of the tree are defined to be goal lists, `compsof` defined to return the first literal in the node, and `sons` to create the resolvent based on the first literal, Bowen's interpreter would perform the normal OR-parallel goal search. AND parallelism could be introduced by having `compsof` split a node into larger pieces, and a less eager search could be implemented by various strategies in `sons`.

Clauses are compiled into dataflow graphs called assertion graphs in a model defined by Bic [2]. A "graph fitting" process is used to evaluate queries. Graph templates carried on tokens are matched against portions of the assertion graph; if the template matches a value at a node, it is passed on to the next node, otherwise it is absorbed. Multiple results are obtained by duplicating a token at a choice point in the assertion graph.

Both major issues of OR parallelism – storage management and process control – are addressed in the "goal rewriting" model of the PIE system of Goto, Tanaka, and Moto-Oka [38]. Intermediate goal statements in this pure OR-parallel model are continually rewritten in reduced form in order to keep them as small and independent as possible. Control of activity is achieved through a relation defined on active goals in a global pool.



An OR Process is an independent interpreter created to solve one literal. All solutions to the literal are sent in messages back to the parent process.

Figure 3.4: OR Processes

3.1.2 OR Processes

OR processes can be thought of as independent interpreters created to solve a goal statement consisting of exactly one literal.

OR processes work in a dynamic environment of other processes, usually called AND processes. The collection of processes and their communication channels form an AND/OR tree, where processes at one level use processes at a lower level to help solve a goal and then communicate results to their parent processes (Figure 3.4). The first interpreter based on OR processes, a precursor of the AND/OR Process Model, was described by Conery and Kibler [21]. The OR processes of this model, and the style of OR parallelism that results, will be described in detail in Chapter 5.

The contrast between OR processes and processes in a pure OR-parallel search of a goal tree is best seen in the communication requirements placed on each. The processes in the search are, conceptually, totally independent, since they do not have to communicate with one another in the basic operation of the model. When a goal is not unifiable with the head of any clause, the process simply terminates, and when it successfully derives a null clause, it notifies the user and terminates. An OR process, on the other hand, must

pass results back to its parent process and communicate with descendant processes to coordinate their role in the solution of the goal literal.

Although this communication adds overhead to the system, it also provides an opportunity for controlling the level of parallelism. OR processes can generate a continuum of activity, from purely sequential to the same maximum amount exploited by parallel search. The amount of OR parallelism exploited by the system is a function of the control structure within the OR process, the control structure in the parent process, and the protocol used when sending results from the OR process to its parent. The difference between controlling parallelism through the communication protocol, as opposed to the way it is done in the pure OR-parallel models, is that in the latter a large amount of global information is required. The decision to expand a node is based on what is happening arbitrarily far away in the goal tree. In the OR process models, growth control is a local decision, based on local factors such as the status of the communication channel between a parent and child process. The global perspective might provide the optimal search, as in the best-first search in B-Log, but it does so at the cost of global information flow.

An interpreter by Furukawa, Nitta, and Matsumoto [34] is at the sequential end of the continuum, generating a minimal amount of OR parallelism. Their form of parallelism has also been called *backup* OR parallelism The idea is that while a controlling process is using one result from an OR process, the OR process can proceed, working on a backup answer. If the backup answer is ready before the parent process needs it, the OR process blocks. Thus the OR process is just trying to stay one step ahead of its parent, creating a situation where the parent never has to wait for an answer (after the first one) while at the same time consuming as little processing resources as possible. The overall effect is a sequential, depth first interpretation where backtracking is faster due to the presence of precomputed backup results.

A system of OR processes designed to extract the maximum amount of OR parallelism is the interpreter of Lindstrom and Panangaden [59].[2] In this model, packets of information containing variable bindings are routed between OR processes and their parents and descendants. Parent AND processes are designed to accept results from the OR processes as quickly as they can be generated, so AND processes may be performing operations based on many different results simultaneously. Lindstrom and Panangaden call this effect "induced AND parallelism." There is no actual AND parallelism in this system, according to our definition, since the sequence of derivations for any one result is not shortened. This can be seen by tracing the operations

[2]In naming the nodes of their process tree, Lindstrom and Panangaden follow the convention of Nilsson [68] in labeling nodes of an AND/OR tree according to the node's relationship with its siblings, not its children. The objects called OR processes in this book are called AND processes by Lindstrom and Panangaden.

in the tree of processes required to produce any single answer. A packet reaching the root of the process tree is the result of a sequence of unifications equivalent to those performed in one of the paths from the root of the goal tree to a success leaf.

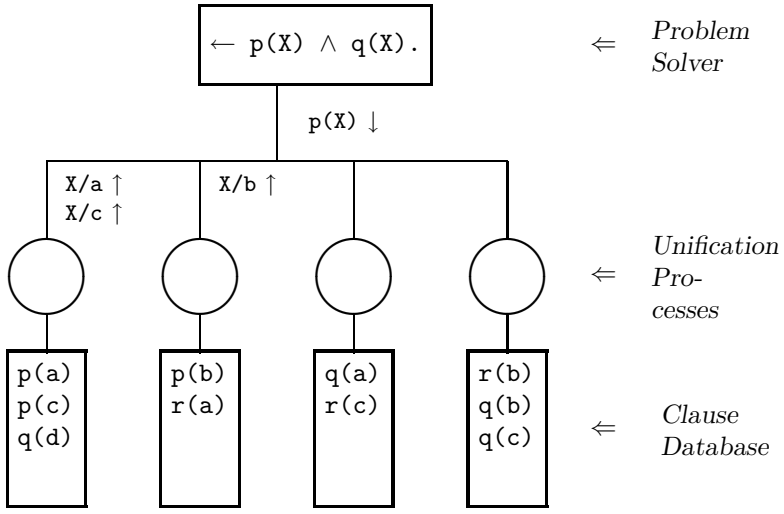
Two other systems defined in terms of routing streams of binding environments between nodes representing literals are the systems of Umeyama and Tamura [88] and Halim and Watson [39]. In the system of Umeyama and Tamura, there are no objects explicitly called OR processes – the actions carried out by OR processes in the Lindstrom and Panangaden system are built into the parent process – but the style and amount of parallelism are very similar.

The OR parallelism in the AND/OR Process Model is near the middle of the scale. OR processes try to create as many results as possible, but the results are passed back to the parent one at a time, as the parent demands them. In the system of Tamura and Kaneda [82], OR processes send results immediately instead of waiting for the parent process to request them. Buffering of results is handled at the system level, so that results are stored in a message queue if the parent is not ready to accept them. Varying the size of the queue is a means of controlling the amount of parallelism. If the queue size is one, the parallelism would be very similar to backup OR parallelism; longer queues would allow more parallel activity.

Another metric for comparing goal tree search with an OR process model is the size of the problem solved by each type of process, and the amount of space required to represent the state of the process, including variable bindings. Processes in pure OR-parallel models are analogous to Prolog interpreters in the amount and type of state information kept. At any point in time, they keep a list of goals remaining to be solved, and variable bindings are represented in a stack of environments. This results in a large structure in a single memory address space, where the value of a variable can be located anywhere in the structure.

OR processes follow the principles of modularity and scalability mentioned in Chapter 1. By using a set of small, independent interpreters, state information can be localized. Binding environments for the AND/OR Process Model, described in Chapter 7, are transportable from one memory system to another and require a minimum of references from one environment to another.

Other systems use the OR process style of OR parallelism in conjunction with some form of AND parallelism. Discussion of these is postponed until the section on AND-parallel models.



The problem solver broadcasts a literal, requesting bindings for variables in the literal. Each node performs unifications of the broadcast literal with the heads of clauses stored in the node, and for each successful unification returns a message containing the bindings.

Figure 3.5: Search Parallelism

3.1.3 Distributed Search

Once an interpreter selects a literal from a goal statement, it must find every clause with a head that unifies with the selected literal. If the logic program is very large, this operation can be enhanced by distributing the clauses in the database and performing a parallel search of the database before each inference step. The other OR-parallel models are either defined at a level of abstraction above the level where processor topology is important, or require a central database of clauses accessible by all processors, or assume identical copies of the program are stored with each processor. The systems described in this section are designed for applications where large programs must be split into smaller sections and stored in the local memories of the nodes of the multiprocessor (Figure 3.5).

The first system to use this form of parallelism was PRISM [31, 50]. The underlying architecture is ZMOB, a ring of 256 microprocessors connected by a specially designed bus. The nodes in the ring are partitioned into a central problem solver, or *PS*, an extensional database, where unit clauses are stored, and an intensional database, where non-unit clauses are stored. In each execution step, the PS selects a literal from a goal statement and

broadcasts it to every database node. Each node performs a unification of the literal and the heads of the clauses stored there; when a unification succeeds, the resulting bindings are sent back to the PS. The PS applies a set of bindings and moves on to the next literal in the goal, repeating the process. In its simplest form, the PS is a single processor, and the overall execution is a depth-first search. If the PS consists of several nodes, it is possible to have each working on several branches of the proof tree, corresponding to a pure OR-parallel execution.

Taylor, Lowry, Maguire, and Stolfo [83] described a system for a multiprocessor database machine, taking advantage of special purpose hardware to perform low level relational operations. The abstract architecture consists of a control processor (CP) and several small grain processing elements (PEs). PEs in this system store only the heads of clauses. If a clause is a non-unit clause, the PE stores a tag associating the body with the head, but the body itself is stored in the CP. The difference between this system and PRISM is that all goals in the current goal statement are broadcast simultaneously, instead of sequentially from left to right. Unifications are performed by the PEs, and results returned to the CP. The CP then executes a join operation, discarding inconsistent bindings. If an element of a tuple is based on a non-unit clause, the body of the clause is solved in the same manner. A form of AND parallelism is exploited here, since the CP simultaneously requests solutions to all literals in the body of a goal. This technique will work for database queries, but not for goal statements containing evaluable predicates or other goals that succeed only when variables in certain arguments are bound.

The system of Warren, Ahamad, Debray, and Kalé [98] is designed to work in an environment of loosely coupled processors connected by a broadcast network. In their system, one node is designated as the *master* for each top level goal. The master selects a literal from the goal, and broadcasts a request for solutions for the literal. If another node has a clause with a head that unifies with a broadcast literal, it applies the bindings required by the unification and becomes the master of the goal statement created from the body of the clause. Thus the role of PS is distributed, since any node can be the master of a goal statement and is in charge of requesting solutions to the literals in the body of the statement. Since a master processes literals left to right, sequentially, and many masters can be active at any one time, the system is also doing a parallel search of a tree of derived goals.

Another system based on distributing program clauses to different nodes and having the processors perform unifications in parallel is described by Nakagawa [65]. In this system, all clauses of one procedure are stored in the same node. When a call is made to a procedure for `p(X)`, a token carrying bindings from the environment of the call is sent to the node where the procedure is stored. The corresponding processor does the unifications of

the input literal with the heads of the clauses of `p` serially. For each one that succeeds, it sends an updated token to the processor which will solve the call following `p` in the original clause. As is the case with the induced AND parallelism in the Lindstrom and Panangaden model, what appears to be AND parallelism is really OR parallelism. A large amount of concurrent activity is generated in the solution of one clause body, but it is based on the processing of multiple results for the clause, not parallelism in deriving one result. Any given result is produced sequentially, with the token that carries the result being passed serially between the nodes where the literals of the query are stored.

3.1.4 Summary

The three styles of OR parallelism appeal to different application areas. At the abstract interpreter level, pure OR parallelism has the least communication overhead, and there is a minimum of coordination required for the parallel processes. Implementation techniques developed so far are parallel versions of the three-stack model commonly used for Prolog, and are based on the assumption that all memory is, if not physically directly shared by each processor, at least part of a single address space. Unless these techniques are adapted for non-shared memory, parallel goal search will not execute well on massively parallel machines due to the cost of accessing information in non-local memory. Also, the fact that processes proceed as quickly as possible without any interaction may also pose a problem for large programs on small machines. In a situation of unrestrained growth in the number of processes, each processor will have to execute a large number of processes, and the overhead of switching between processes could be severe. Heuristics or other information may guide the search and control the growth, but this requires global communication among the processes.

Models based on OR processes are, at the abstract level, more modular. This modularity allows for a direct mapping to a partitioned representation for environments. The communication overhead of passing results back to parent processes may make these systems slower on small problems. However, the fact that processes must communicate with each other might be turned to advantage if protocols can be adapted for use in controlling the exponential growth of the number of processes in larger programs.

OR process models and the parallel search model make no provisions for distribution of clauses. There is an implicit assumption that when a process needs to unify a goal with heads of candidate clauses, it has access to all candidates in the system. This means either distributing all clauses to all processors in the system, or providing for non-local access. The strength of the distributed search models is in the partitioning of the program. The parallelism lies in having all nodes looking for candidates for unification with

a broadcast goal, and in doing the unifications in parallel at the nodes. For databases of even a moderate size, partitioning will be essential, since each tuple in an explicit relation is represented by a unit clause in the logic program.

The drawback to search parallelism is the communication involved during unification. When the control processor broadcasts a request for unification for a literal, it must also broadcast current bindings for the variables in the literal. If variables can be bound to large structures, this may be quite costly. However, in database applications it is common to restrict values of variables to atomic terms, simplifying both the unification algorithm and the representation of bindings. This fact, plus the ability to distribute large programs over many nodes, make this form of parallelism attractive to database applications.

3.2 Models for AND Parallelism

AND parallelism is the parallel solution of more than one goal in a given goal statement. The central problem in implementing this form of parallelism is management of variables occurring in more than one literal of the goal statement. In a goal statement such as

$$\leftarrow \texttt{p(X)} \wedge \texttt{q(X).}$$

the variable `X` occurs in both goals. To solve the goal statement we need to find a value for `X` that satisfies both `p` and `q`. Most abstract models for AND parallelism handle this problem the same way: they allow only one of the goals to bind the shared variable, and postpone solution of the others until the variable has been bound. The models differ in scheduling of goals, the rules for binding shared variables, and whether the goals that bind variables can be nondeterministic.

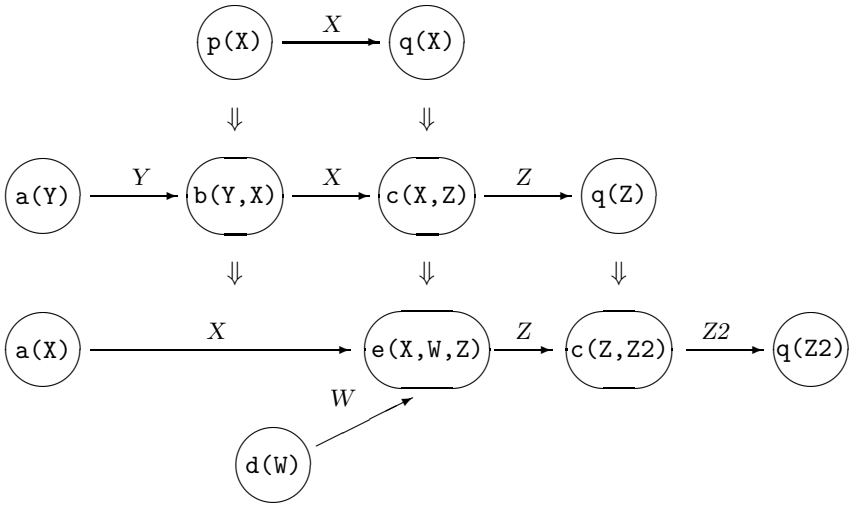
3.2.1 Stream Parallel Models

The coroutine control strategy described in Section 2.5.4 has been developed into AND-parallel control in a number of systems. In the behavioral interpretation of a clause,

$$\texttt{p} \leftarrow \texttt{q} \wedge \texttt{r.}$$

means "process `p` can be replaced by the system of processes `q` and `r`" [15, 77]. In this interpretation, literals in a goal statement are processes, and variables occurring in more than one literal are communication channels between the literals (Figure 3.6). This form of AND parallelism was labeled *stream parallelism* by Conery and Kibler [21]. Early work on the semantics of a logic

```
← p(X) ∧ q(X).
p(X) ← a(Y) ∧ b(Y,X).
q(X) ← c(X,Z) ∧ q(Z).
b(X,X).
c(X,Y) ← d(W) ∧ e(X,W,Y).
```



The behavioral reading of a clause `p ← q ∧ r` *is "process* `p` *can be replaced by processes* `q` *and* `r`*." A variable shared by two literals is a communication channel between the corresponding processes. The double arrow in this diagram indicates process replacement, a single arrow is a communication channel. The three rows of the figure show the system at three different times. Note that if two processes communicate directly, there will be a direct path between two of their descendants.*

Figure 3.6: A Goal Statement as a Network of Processes

program as a network of parallel processes is described in papers by van Emden and de Lucena Filho [33] and Hogger [43].

In Parlog [17], the successor of IC-Prolog, producers and consumers operate in parallel instead of in the interleaved sequential manner of IC-Prolog. Clause bodies in Parlog have two components. The *guard* is a list of literals at the front of the body, separated from the remainder of the body by a colon. The following portion of the sieve of Eratosthenes is taken from Clark and Gregory:[3]

```
filter(filter-num,[num|list],[num|f-list]) <-
       ~divides(filter-num,num) :
       filter(filter-num,list,f-list).
filter(filter-num,[num|list],f-list) <-
       divides(filter-num,num) :
       filter(filter-num,list,f-list).
```

The colon is a commit operator. Denotationally it is the same as conjunction, so the clause is solved only if all literals in the guard and body are solved. The guards are used to determine the clause that will be used to solve a call to the procedure. When a procedure is called, parallel processes are set up to evaluate the guards of all clauses in the procedure. If all guards fail, the call fails. Otherwise, when one succeeds, the system terminates the others and *commits* to the body of the clause with the successful guard. In the example, when a goal statement contains a call to `filter`, processes are started to evaluate the two guards. The tilde is a negation operator, implementing negation as failure. As soon as one of the guards is solved, the system terminates the other guard process, and uses the corresponding body to finish the call.

In general, the guards are not mutually exclusive, as they are in the above example. The policy of committing to the body of the clause with the first guard that evaluates to true, and discarding all other choices, is known as committed choice nondeterminism [28, 42]. It has also been called "don't care" nondeterminism, as opposed to the "don't know" nondeterminism of Prolog and the OR-parallel models. Parlog also allows "don't know" nondeterminism, via an evaluable predicate named `set`, which evaluates its argument in an OR-parallel fashion.

Crammond and Miller [23] have designed a virtual machine model for Parlog. What they refer to as AND processes and OR processes are control abstractions for managing parallel operations in a global environment. In keeping with the Parlog model, when a guard terminates successfully, the global environment structure is updated. Nodes representing other guards are pruned, and the successful guard is merged with its grandparent conjunct.

[3] In Parlog, variable names start with lower case letters.

Concurrent Prolog, developed by Shapiro [78], is another system based on parallel execution of coroutines using guards and committed choice nondeterminism. Coordination of goals with variables in common is done with *read-only variables* identified by a question mark. In the goal statement

$\leftarrow$ `p(X?)` $\wedge$ `q(X).`

the occurrence of `X` in the first goal is marked as read-only. If solution of a goal would bind a read-only variable, the goal is postponed until some other goal binds the variable. In this example, execution of `p` and `q` proceeds in parallel. If a step in the solution of `p` would unify `X` with a non-variable term t_1, execution is suspended until a step in the solution of `q` binds `X` to a term t_2. The execution of `p` then resumes in the unification of t_1 and t_2.

One can use co-routine control structures to implement objects [77]. The clauses of a procedure each implement one method (a behavior defined for a specific type of message), messages to the object are represented by an input list, and the internal state of an object is represented by parameters in recursive calls. For example, the following procedure implements a string:

```
string([append(S)|Msgs],Si) :-
       concat(S,Si,So), string(Msgs?,So?).
string([substringp(S,Ans)|Msgs],Si) :-
       substringp(S,Si,Ans), string(Msgs?,Si).
```

The first parameter is used for a stream of messages sent to the object, and the second represents the object's internal state, in this case a list holding the characters of the string. An instance of a string is created by calling `string` with two parameters, the stream used to carry input messages and the initial state of the string:

```
... string(M1?,"abc") ...
```

Another process sends `string` a message by creating a partial binding for `M1`. If `M1` is bound to `[append("d")|M]`, the first clause is selected (a null guard is always true, and the other choices are discarded), `concat` creates a new internal state for the string, and the recursive call has the effect of replacing the original string by a new string with the updated internal state. The read-only variables on the message streams in the calls to `string` prevent the object from instantiating this parameter; messages must come from other processes.

What makes Concurrent Prolog and Parlog different from other object oriented programming languages is the use of logic variables to implement two-way communication [16, 78]. If an unbound variable is sent as a parameter in the message, the object can return information to the sender by binding the variable. For example, if the message for the string is `substringp("bc",X)`

the call to `substringp` in the body of the clause can bind `X` to `yes` or `no` depending on whether `"bc"` is a substring of the current string.

The language GHC (for *G*uarded *H*orn *C*lauses) is another language exploiting parallelism and committed choice nondeterminism (Ueda [87]). Instead of using modes or read-only variables, GHC synchronizes calls via rules for executing guards. For example, guards may not bind variables of the head of the clause, so if a guard ever reaches a state where it would bind a head variable, it blocks until the variable is bound by some other process.

The Distributed Logic language of Monteiro is based on an extension of Horn clause resolution [64]. Monteiro defines the notion of a *distributed clause* and a resolution-based inference step for goals statements and distributed clauses. A distributed clause has the form

```
p1, p2, ... pn <- q1, q2, ... qn.
```

with n literals on the left and right sides. This is equivalent to the n Horn clauses

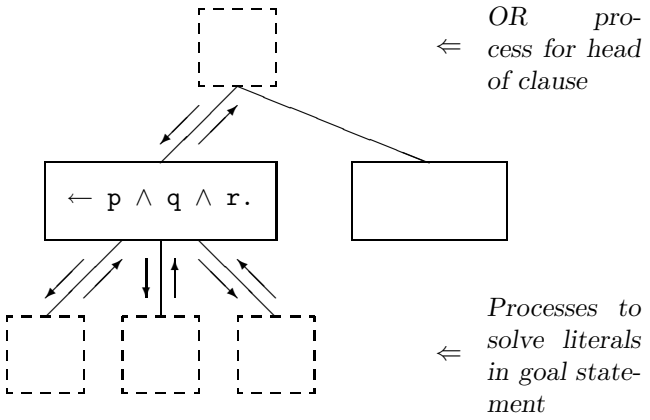
```
p1 <- q1.
    ⋮
pn <- qn.
```

In an execution step based on a distributed clause, a set of literals is chosen from the current goal statement. Each literal of the set must unify with a literal in the head of the distributed clause. Goal statements and the bodies of clauses can be annotated to specify the order of reduction. The utility of the distributed clause syntax is that it allows variables to be shared among the clauses for purposes of synchronization and communication.

3.2.2 AND Processes

AND processes, like OR processes, are independent interpreters solving small portions of a program (Figure 3.7).

In the AND/OR Process Model, an AND process solves the body of a clause by creating an OR process to solve each literal. AND parallelism in this model is a matter of having more than one OR process active at a time. When literals share a variable, only one, called the *generator* for the variable, is allowed to bind it. One difference between the AND/OR Process Model and the stream parallel models such as Parlog is that all steps in the solution of a generator are completed before any of the consumers start. This means there is no overlapping of execution when the shared variable is bound to a large structure in a series of partial bindings. In the stream models, parallel processes are started simultaneously for each literal in the body of a clause, with consumers blocking until the shared variable



An AND Process coordinates the solution of a clause body. It creates an OR process to solve each individual literal (see Figure 3.4).

Figure 3.7: AND Processes

is bound to a nonvariable terms. In most AND process models, the literals are ordered prior to execution, and processes are started by the parent only when generators have completed execution.

Another difference is in the number of results: the stream models generate just one solution per procedure call, due to the nature of committed choice nondeterminism, but AND processes can generate a sequence of results, through an operation analogous to backtracking. The difference between passing a number of values through a communication channel in the stream models and passing a number of values through multiple solutions in the AND process models lies in the semantics of the clause. Consider the clause

```
p(X,Y,Z) ← q(X,Y) ∧ r(Y,Z).
```

For the coroutine models, assume `Y` is a communication channel. Semantically, the meaning of the generator literal, $\mathbf{D}(\mathtt{q})$, contains one tuple; the value of `Y` is a single complex term created through a series of partial bindings. In the AND process models, `Y` will be bound to as many different terms as it takes to find solutions for `r(Y,Z)`. $\mathbf{D}(\mathtt{q})$ will have one tuple for each of these values, rather than one tuple with a complex term containing each of the values as subterms.

The two forms of AND parallelism were combined in an extension to the AND/OR Process Model by Borgwardt [3]. The memory representation

and control mechanism of this system allow overlapping production and consumption of a structure, as in the stream models. If the consumer fails when processing part of the structure, the producer must start over on a new term, which may or may not have acceptable elements from the first structure. In other words, the producer starts work on a new stream, rather than replacing previous elements of the current stream. Referring to the example clause, $\mathbf{D}(\mathtt{q})$ would have more than one tuple, where the value of `Y` in each tuple is a complex term created as a stream. Another system combining the two forms of AND parallelism is PM, by Singh and Genesereth [79].

Allowing literals to provide multiple results complicates the AND-parallel control. Consider the following clause:

$$\mathtt{p(X,Y)} \leftarrow \mathtt{q(X)} \wedge \mathtt{r(Y)}.$$

Given a call `p(A,B)` where `A` and `B` are both unbound, if there are n_q results for `q(X)` and n_r results for `r(Y)`, there will be $n_q \times n_r$ solutions of the goal. Prolog creates this cartesian product of results through backtracking; each time `q(X)` succeeds, the solution of `r(Y)` starts all over again. In an AND-parallel system, where `p` and `q` are being solved simultaneously, another mechanism must be used to create the cross product of results.

The technique used in the AND/OR Process Model is a form of intelligent backtracking adapted for parallel control. An assumption made by the original implementation cause it to miss some valid combinations of tuples [19]. Improved algorithms have been published by Woo and Choe [100] and Chang, Despain, and DeGroot [10].

The method of ordering literals for parallel execution in AND processes in the AND/OR Process Model is dynamic. An execution order is determined at runtime, when the clause is invoked, and may change depending on the binding of terms by descendant OR processes (Section 6.1). To avoid this runtime overhead, DeGroot devised a technique for ordering literals at compile time, and checking the binding status of variables to make sure the ordering remains valid [25]. The runtime overhead is minimal, and the amount of parallelism lost due to a more conservative ordering may not be too high. The idea of ordering literals at compile time, and using this information to guide backtracking in a sequential system, was explored by Chang and Despain [9].

Three are three other models based on compiling clauses into dataflow graphs. Ito *et al*[45, 46] compile clauses into lower level dataflow operators. Clause heads are compiled so the first level of unification is done in parallel, but a later consistency check operator negates much of the effect of parallel unification. Multiple solutions are generated by a join operation defined for streams of solutions from OR processes.

The model of Kacsuk [47] is based on compiling clauses into history sensitive dataflow operators. Operationally, the system is very close to the AND/OR Process Model. The kinds of tokens correspond to the types of

messages in the AND/OR Process Model, and the actions of OR nodes in the dataflow graph mirror the actions of OR processes. AND-parallel operations are synchronized to a greater extent, exploiting less parallelism as a result. The method for backtracking or otherwise generating all results is not specified.

Bic's model [2] was described previously, in the section on OR parallelism. The compiled graph is the dual of the graphs created in the other dataflow models. In an assertion graph, the collection of ground terms in arguments of assertions in the program are the nodes of the graph, and arcs connecting nodes are labeled with procedure names.

Other AND-parallel models based on the notion of independent processes have been developed. AND processes in the Sync model of Li and Martin [57] do not use a parallel backtracking algorithm. Instead, they perform an incremental join operation on the values returned by the parallel solution of the literals. Analysis of the body of the clause is used to order the literals, so the problems cited for the system of Taylor *et al* [83] will not arise here. The ordering of literals also leads to improvements in the complexity of the algorithm for joining the results.

In the REDUCE/OR Process Model of Kalé [48], "reduce processes" incorporate the actions of both AND and OR processes. Instead of one descendant OR process per literal, there is one descendant per *instance* of a literal. An instance is created for each possible unification of a literal with the head of a program clause. Each instance sends results back independently, leading to a higher degree of parallelism in the reduce process, since multiple results are coordinated by OR processes in the AND/OR Process Model. As in the Sync model, multiple results from a process are formed by a join operation on the results from descendant processes. The order of solution of literals is determined by a *data join* graph, the dual of a literal dataflow graph – arcs represent literals and nodes represent ordering constraints.

Another model incorporating both OR parallelism and AND process style AND parallelism was developed by Wise [99]. This system, named Epilog,[4] uses annotations on variables to determine the order of solution of literals. As in Concurrent Prolog, solution of a literal containing a variable labeled with a question mark is prevented from binding the variable, so `p(X?,Y!,Z@)` will not be solved until `X` is bound by solution of some other literal. The annotation on `Y` in this example means `Y` must be unbound before the goal is solved, and it will be bound during the solution. The annotation on `Z` means `Z` must be bound to an atomic term before solving the goal. Another mechanism for scheduling literals in Epilog is the use of thresholds. If, for example, `p(X,Y,Z)` has a threshold of two, when any two of the three arguments become instantiated the AND process will start a descendant process to solve

[4] Not to be confused with the Epilog language of Pereira [71].

the literal and bind the third variable. It is not mentioned how or if Epilog creates the cross product of terms for nondeterministic goals.

3.2.3 AND Parallelism in the Goal Tree

The models for AND parallelism surveyed so far are all defined in terms of processes that work on small independent parts of the computation. The next two systems are similar to pure OR parallel systems in the sense that they are defined in terms of parallel operations in a global search of a tree of derived goals.

Dembinski and Maluszinski have defined a method for parallel expansion of the goal tree [27]. Programs in this model are annotated with mode information. Any literal on the frontier of the tree is a candidate for expansion. The operations of this model are defined in terms of interleaved operations, but, since the effects of reductions are local, there is potential for parallel expansion. OR parallelism is not exploited; instead, the system uses an intelligent backtracking method to generate all solutions.

The H-Prolog system of Nakamura [66] has a parallel search style of parallelism for both AND and OR parallelism. This system uses a global tree of resolvents where AND-parallel node expansions are done by generating resolvents for a number of literals within a node. The next level of nodes in the search space is formed by joining all combinations of resolvents and throwing away those with inconsistent bindings. For example, if three literals are expanded from a goal, generating i , j, and k resolvents, respectively, the next level in the search space will have up to $i \times j \times k$ resolvents. The complexity of the join operation is proportional to the size of the resolvents. In a purely deterministic algorithm, such as divide and conquer, where the input is split into two subproblems, the tree of resolvents after the join will have a branching factor of one, *i.e.* the "tree" is a linear list. The length of the resolvent grows exponentially with the depth of the tree as the input problem is divided and more goals are added to the goal statement, and then gets smaller as the primitive problems are solved. In nondeterministic problems allowing some amount of OR parallelism, the problem is compounded by having a number of large resolvents at each level. Multiple solutions are obtained from the set of surviving resolvents, so this system correctly handles that problem. The policy for selecting nodes for AND-parallel expansion is not specified.

3.2.4 Summary

Two different styles of AND parallelism are emerging. One style, called stream parallelism in [21], is exploited by Parlog, Concurrent Prolog, and other languages oriented toward system programming, where committed choice

nondeterminism is a valuable programming technique. The other style, typified by the AND processes of the AND/OR Process Model, is oriented toward a more exploratory style of nondeterminism, replacing the backtracking method of sequential systems with a semi-intelligent backward execution algorithm. The AND process style is designed to work in conjunction with OR process style OR parallelism.

As is the case with OR-parallel models, the process models are, at the conceptual level, more modular. The binding environments and process state are smaller and independent of one another. In the stream parallel model, shared variables are communication channels, created when a clause is invoked and passed on to descendant computations. As a result, goals at arbitrary locations in a goal tree must be coordinated, so that when one binds a variable the other is signaled to proceed. Support for the communication channel will constrain the design of runtime environments for these models.

Unlike the case with OR-parallel models, there have been attempts to design systems to use both styles of AND parallelism. In the best of both worlds, when producer literals create large structures, execution of producers and consumers would overlap, and consumers would be able to reject pieces of the structure, forcing producers to generate different values for those pieces.

The combination of both styles of AND parallelism has an aesthetic appeal, as well. Logic programming is distinguished from other models of computation by logical variables and nondeterminism. Stream parallel models take advantage of partial bindings allowed by logical variables, using this feature for object oriented programming and other elegant techniques, but at the expense of exploratory nondeterminism. AND process models are the opposite, sacrificing parallelism when partial bindings are present but able to exploit nondeterminism. Definition of a unified model featuring both distinguishing characteristics of logic programs is a worthwhile goal.

3.3 Low Level Parallelism

Systems exploiting low level parallelism seek to speed up the operations in a standard interpreter, as opposed to modifying the overall search strategy by performing more than one inference at a time. A global view of control in such systems still shows a depth-first search of a goal tree.

Development of techniques for speeding up sequential logic programs is worthwhile in its own right. When Prolog programs are compiled instead of interpreted, an order of magnitude improvement is seen [97]. It is reasonable to expect low level support at the machine level to provide an additional order of magnitude. Research in this area is bound to pay dividends for parallel systems as well. Definitions of microprogrammed or VLSI algorithms and machine level representation of terms and binding environments will certainly

be useful in parallel machines.

Another potential benefit is the use of special purpose sequential machines as the building blocks in a parallel machine. Some parallel machines, such as the one built by Kumon, *et al*, are collections of single processor Prolog machines. One advantage is that when the system is overloaded, it can switch to a sequential operating mode, so that it does not generate more parallel tasks. If every node in the system is busy, there is no point in further distributing work, and communication overhead will be less in sequential mode.

However, this idea must be implemented carefully in parallel logic machines. It would be a good idea to switch *operating modes*, from a parallel mode to one where less parallel activity is generated. It would be a bad idea to switch *models*, for example from pure OR parallelism to depth-first search. Different models generate different sets of results, depending on whether or not they tolerate infinite branches. Results found in a pure OR-parallel system may not be generated by a depth-first search. If the system switches models depending on system load, users will be faced with a situation where a result is returned when the system is not busy, but not when the system is overloaded; a procedure that works by itself might not succeed when embedded in a large program.

Techniques for exploiting low level parallelism fall into two broad categories. The first is parallel unification. Since unification is the heart of logic programming, the "inner loop" executed during every step of the program, it makes sense to speed it up as much as possible. The second category is parallelism in the control mechanism, in other words instruction pipelining.

Intuitively one would think it difficult to unify different arguments of two input literals in parallel, since bindings in the arguments must be consistent. When a term such as `p(A,f(g(A)))` is unified with another term, the algorithm must check for consistent bindings of `A`, and possibly look inside structures to determine bindings. Formal arguments for why unification is "probably not parallelizable" can be found in an article by Dwork, Kanellakis, and Mitchell [30]. However, as Lindstrom points out, the average case for unification in logic programs does allow parallelism [58]. In that paper Lindstrom presented a parallel unification algorithm based on synchronizing access to variables in a stack frame. An atomic "check and bind" operation assigns a value to a slot if the slot is unbound, or returns the current value so a consistency check can be performed if the slot is already bound.

The approach taken by Robinson is to divide unification into two phases, a pattern matching phase followed by variable dereferencing and binding [75]. Pattern matching is handled by a special purpose content addressable memory; it returns information about the variables that must be dereferenced to the control processor. Robinson discusses how such a memory could enhance the execution of a Prolog processor in the construction and access of Prolog

style binding environments.

The idea of compiling Prolog clauses into lower level machine instructions is due to Warren [90, 95]. In this technique for implementing programs, unification is carried out by a sequence of virtual machine instructions. Calls to a clause are compiled into instructions to place terms in argument registers, and clause heads are compiled into instructions to test the contents of the registers against the terms in the head. Calls to a general unification procedure are thus replaced with sequences of instructions tailored to each individual clause head. The instructions are executed sequentially, so this corresponds to a sequential unification. Low level parallelism in this system is obtained from pipelining the virtual machine instructions. Tick and Warren designed a three-stage E-unit as the heart of instruction execution in their system [85]. Early simulation results of a system with two E-units were discouraging [84]. The goal was to see if low level dispatching of unification instructions would naturally lead to parallelism in unification due to different arguments being unified in different E-units. However, since the E-units were synchronized between procedure calls, the percentage of time both were busy was quite low.

Other machines designed for sequential Prolog execution, but exploring new representations and methods which may eventually have a bearing on parallel machines, are PLM [29], PEK [49], PSI [81], and HPM [67].

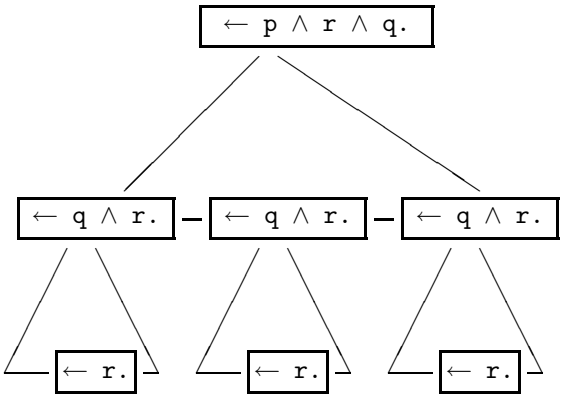
3.4 Chapter Summary

Two basic forms of parallelism in logic programs were identified in this chapter, and numerous systems exploiting both forms were surveyed.

OR parallelism is parallelism in creating multiple results. When there is more than one way to solve a goal, OR parallelism can be used to compute the results in parallel. OR parallelism serves the same purpose in parallel models that backtracking does in Prolog. It is associated with the exploratory nondeterminism of search problems, a valuable technique in AI programming.

AND parallelism is parallelism in computing any one result. It is associated with parallel execution of deterministic or committed choice nondeterministic algorithms. It corresponds to the parallelism found in functional programs. If we are to exploit parallelism in a logic program for a deterministic algorithm, one based on divide and conquer for example, it must be through AND parallelism.

Models combining AND and OR parallelism typically pay a penalty in the form of higher control overhead. Pure OR-parallel models (corresponding to parallel search of a goal tree) require very little coordination among search processes. Adding AND parallelism requires policies for deciding which goals to solve and methods for checking the consistency of the results. AND-



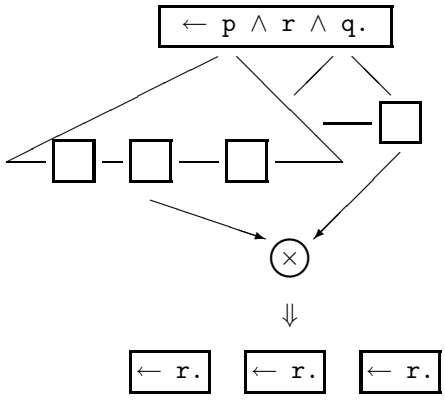
The area enclosed in a triangle below a node for a goal G *represents a portion of the goal tree used to derive a solution for one literal* L *in* G. *The bottom of the triangle represents a cut containing only fail nodes or active nodes where the goal statement is* G *with* L *resolved away. There will be* n *such goals, where* n *is the number of solutions of* L. *In this example, if the solution of* `q` *does not depend on* `p`, *the subtrees rooted at these nodes represent* n *identical computations of* `q`.

Figure 3.8: Duplicate Computations in a Goal Tree

parallel models that allow multiple results from OR parallel components require a join operation or other method for collating results. It is a good question as to whether the overhead is worth it. Why not design a system to exploit one form or the other, depending on the most likely type of application?

From the perspective of AND-parallel models, the argument to include OR parallelism is the same as the argument to include exploratory nondeterminism. Adding nondeterminism would add control overhead in the form of choice points and backtracking information if implemented sequentially, or multiple environments if implemented in a parallel fashion. This form of nondeterminism is one of the two things that distinguishes logic programming from other models of computation. If it is left out of the system, large classes of interesting problems become much harder to program, among them natural language parsing, database queries, and state space searches.

From the perspective of an OR-parallel system, the argument to include AND parallelism is more compelling. If a subset of goals at a node in the goal tree can be solved in parallel, there will be some speedup in the overall solution. More importantly, if they are *not* solved in parallel at the higher



In this diagram, two independent derivations are started simultaneously, each based on a single literal. The results of each derivation are combined to form processes to solve the remaining goal from the original goal statement. When `p` *and* `q` *are independent,* `q` *is solved once, not once for each solution for* `p`*. This is the strategy used in the AND/OR Process Model.*

Figure 3.9: Combined AND and OR Parallelism

level of the tree, they will be solved many times over in descendant nodes. This is illustrated in the tree of Figure 3.8. The root node has three goals:

$\leftarrow$ `p` $\wedge$ `q` $\wedge$ `r.`

If there are n solutions to `p`, there will be exactly n nodes in the tree containing the same goal statement, with the goals `p` $\wedge$ `q`. These will be descended from the nodes corresponding to the last step in any solution of `p`. If solution of `p` determines the solution of `q`, for example if the original goal is

$\leftarrow$ `p(X)` $\wedge$ `q(X)` $\wedge$ `r(X).`

and each solution of `p` binds `X` to a different term, then the n nodes represent n different problems. If solution of `q` is independent of `p`, *e.g.*, the goal statement is

$\leftarrow$ `p(X)` $\wedge$ `q(Y)` $\wedge$ `r(X,Y).`

then we will solve the same problem n different times. The best plan is to solve `p` and `q` in parallel when they are independent. The interpreter should grow trees for both `p` and `q` and combine the results to set up solution of `r` (Figure 3.9). The next three chapters will describe how the AND/OR Process Model provides a framework for this type of parallel solution based on a combination of OR parallelism and AND parallelism.

Chapter 4

The AND/OR Process Model

In the AND/OR Process Model, a logic program is solved by a dynamic set of processes that communicate via messages. Processes can be described as actors, objects with discrete states updated in an atomic operation triggered by the receipt of a message from another process [41]. A process will be created to solve a small portion of a logic program. Its state transformations reflect its progress in executing the portion of the program, and it will terminate after all solutions to its subproblem have been found.

There are two types of processes. An *AND process* is created to solve a goal statement, a conjunction of one or more literals. An *OR process* is created by an AND process to solve exactly one of those literals. If there is a nonunit clause in a procedure for the literal, an OR process for the literal will start an AND process for the body of the clause. A computation can be described by an AND/OR tree of processes, with the initial goal statement defining an AND process at the root of the tree. Messages are used to start and cancel descendants and return results to higher levels of the tree.

This chapter lays out the basic requirements of AND and OR processes and the types of messages they generate. The processes defined in this chapter are sequential in nature: at any step, a process will send a message to one other process, and wait for a response to the message before continuing. With sequential processes, the system simply mimics a depth first interpreter, replacing procedure calls by messages. The purpose of this chapter is to show that logic programs can be interpreted by independent, cooperating processes as opposed to a sequential interpreter searching a tree of resolvents. Later chapters will introduce parallelism by showing how a process sending multiple messages in certain steps executes its portion of the program in parallel.

4.1 Oracle

The decomposition of a logic program into a set of AND and OR processes is based on the notion of an *oracle*, an interpreter or machine which solves a problem in one step relative to the interpreter that consults it [44].

The use of oracles in defining independent computations in logic can be illustrated using the following definition of a function to compute the sum of the squares of two integers:

```
ssq(X,Y,Z) ← product(X,X,X2) ∧ product(Y,Y,Y2) ∧
       sum(X2,Y2,Z).
```

A goal using this clause to compute the sum of the squares of the integers `1` through `4` is:

```
← ssq(1,2,A) ∧ ssq(3,4,B) ∧ sum(A,B,C).
```

The first four goal statements derived from this goal by a depth first interpreter, given suitable definitions of `product` and `sum` to do arithmetic operations in one step, are:

```
← product(1,1,X2) ∧ product(2,2,Y2) ∧ sum(X2,Y2,A) ∧
       ssq(3,4,B) ∧ sum(A,B,C).
← product(2,2,Y2) ∧ sum(1,Y2,A) ∧ ssq(3,4,B) ∧
       sum(A,B,C).
← sum(1,4,A) ∧ ssq(3,4,B) ∧ sum(A,B,C).
← ssq(3,4,B) ∧ sum(5,B,C).
```

After four steps, the interpreter has solved `ssq(1,2,A)` and bound the variable `A` to the term `5`. All four steps are part of the solution of `ssq(1,2,A)`; no resolutions in this sequence are based on any other literal from the initial goal statement. The last goal statement shown above contains every literal except `ssq(1,2,A)` from the initial goal. We say the literal `ssq(1,2,A)` has been *resolved away*.

An important thing to notice about this sequence is the set of variables in the resulting goal statement after the first literal has been resolved away: the only variables from the initial goal instantiated during this sequence of resolutions are those occurring in the literal that was resolved away. No other variables from the original goal statement can be instantiated. In this example, the variables of the original goal are `A`, `B`, and `C`; only `A` occurs in `ssq(1,2,A)`, so `A` is the only variable from the initial goal statement that can possibly be bound when resolving away `ssq(1,2,A)`.

Another important aspect of this sequence is the set of possible bindings for `A`. The possible values for `A` come from the tuples of $\mathbf{D}_1(\texttt{ssq})$, the denotation of `ssq`. What this implies is we can find a binding for `A` by consulting

an oracle for a solution to `ssq(1,2,A)` and then bind all other occurrences of `A` in the original goal to the value supplied by the oracle. In other words, we can derive the last goal statement in the example in one step if we consult an oracle for the solution of the first literal.

For the general case, consider an interpreter that resolves away a literal L from a goal statement G_0:

$$L_0 \wedge \ldots L_i \wedge L \wedge L_j \wedge \ldots L_m$$

After a number of resolutions, we are left with a derived goal statement G_k containing every literal from G_0 except L:

$$L_0 \wedge \ldots L_i \wedge L_j \wedge \ldots L_m$$

This interpreter could have derived G_k from G_0 in one step instead of n steps by consulting an oracle to provide a tuple from $\mathbf{D}_1(L)$, constructing a positive literal $L^{'}$ with the terms from this tuple, and generating G_k by resolving G_0 with $L^{'}$. The correctness of this operation is based on the observation that a procedure for a literal L with n solutions can be replaced by n unit clauses, each corresponding to a tuple from $\mathbf{D}_1(L)$; the positive literal $L^{'}$ is one of these assertions.

An overview of the solution of a goal statement in the AND/OR Process Model is as follows. An AND process is an interpreter created to solve a conjunction of goals, which it does by starting an OR process to solve each goal in the conjunction. OR processes are oracles with respect to an AND process; they return solutions which, as far as the AND process is concerned, are derived in a single step. One source of parallelism in the model is due to the fact that an AND process can consult more than one OR process at a time. Under certain conditions, an AND process can start an OR process for one literal and then start another OR process while the first is still working on its goal. For example, given the first goal statement of this section, a parallel AND process would create two OR processes to work simultaneously on the calls to `ssq`, and when both are done start an OR process for `sum`.

4.2 Messages

All messages sent during a computation are either from a process to one of its immediate descendants, or from a process to its parent. Messages are never sent between siblings or non-immediate ancestors. Messages sent downward in the tree are *start*, *redo*, and *cancel*; messages sent upward are *success* and *fail*.

The start message is self-explanatory. When a process has reached a state where it has identified an independent subproblem, it creates a descendant process (with appropriately defined initial state) and sends it a start message.

A success message is sent by a process to its parent when it has solved the task given to it. The task is represented by a set of literals. The success message contains a copy of the set of literals, with variables instantiated as necessary. For example, if the subproblem is to solve the literal `p(X)`, and it can be solved by binding `X` to `0`, the success message will contain the term `success(p(0))`. Note that when the process sending the message is an OR process, the argument of the message is the positive literal L' alluded to in the previous section.

A fail message is sent when a process cannot solve its problem, or after it has produced the last solution. After sending a fail message, the process terminates.

When a process has received an answer from a descendant, and later finds it cannot use that answer, it sends a redo message to the descendant, telling it to solve its subproblem in another way. This means the parent needs a different set of bindings for the variables in the subproblem.

Finally, a process may reach a state where it will never use any success messages a descendant may send, in which case it sends the descendant a cancel message. Any process reading a cancel message terminates immediately.

4.3 OR Processes

An OR process is a process created to solve exactly one literal. The basic requirements for an OR process will be described in this section. A detailed description of a parallel OR process, one with many descendants operating in parallel, is the subject of the next chapter.

An OR process for a literal L must search the entire program for a clause with a head unifiable with L. A sequential OR process searches the program linearly, from top to bottom, stopping when it encounters a clause with a head that unifies with L. If there are no such clauses, the process sends its parent a fail message and terminates.

There are two cases to consider when a unification succeeds, depending on whether the clause is a unit clause or not. If L unifies with a unit clause, the OR process can immediately construct a success message for its parent. For example, if L is `p(a,X)`, and the program contains a unit clause

```
p(Z,b).
```

the OR process sends `success(p(a,b))`.

If L unifies with the head of an implication, the OR process starts a descendant AND process to solve the body of the implication. For example, if L is `p(a,X,Y)`, and the program contains the implication

```
p(V,W,c) ← q(V) ∧ r(W).
```

an AND process will solve the goal statement

$\leftarrow$ `q(a)` $\wedge$ `r(W).`

If the descendant AND process sends the message `success(q(a)` $\wedge$ `r(b))`, denoting the fact that `X` was bound to `b`, the OR process will send its own parent a success message, in this case `success(p(a,b,c))`.

It is important to note that the parent of the OR process does not receive any information about the direction of the solution of the goal until it receives a success or fail message. In particular this means none of its variables are bound in the unifications performed by the OR process. The variables in an AND process are bound only when it receives a success message containing the bindings. This is an important consideration for parallel OR processes, since conflicting bindings may be generated by different clauses in the procedure.

If an OR process receives a redo message from its parent, it must solve its problem another way. Again, there are two possibilities, depending on whether or not the previous answer was based on a unit clause. If the previous answer was formed from a unit clause, the sequential OR process must resume its search for another clause to unify with L. This could lead to the creation of a new descendant (if L unifies with the head of another implication), an immediate success (if L unifies with a unit clause), or failure (if there are no more clauses with heads that unify with L). If the previous answer was obtained from a nonunit clause, *i.e.* it was based on a success from an AND descendant, the descendant is sent a redo message, and the OR process waits for a response from the descendant.

When an OR process receives a fail message from an AND descendant, it must use another clause to solve L. As in the case where a redo message arrives after a solution by a unit clause, the process resumes the search for another clause to unify with L, leading to a new descendant to replace the failed descendant, or an immediate success, or failure.

An OR process sends a cancel message to its descendant only when it receives a cancel from its own parent.

4.4 AND Processes

An AND process must solve all of the literals in the goal statement given to it by its parent. The literals are solved by descendant OR processes, one for each literal. A sequential AND process solves the goal statement much the same as a depth-first sequential interpreter. An OR process is created for the leftmost literal of the goal statement. If the OR process sends a success message, bindings in the answer are applied to the remaining unsolved literals, and an OR process is started for the next literal in the goal

statement. If the OR process sends a fail, another solution must be found for the most recently solved literal, so a redo message is sent to the OR process for that literal.

An AND process sends its parent a success after all descendant OR processes have sent successes. A sequential AND process fails if its first literal cannot be solved, *i.e.* if the OR process for the leftmost literal sends a fail message. When an AND process receives a redo from its parent, it in turn must send a redo to one of its own descendants; in a sequential AND process, this will be the process for the rightmost literal.

4.5 Interpreter

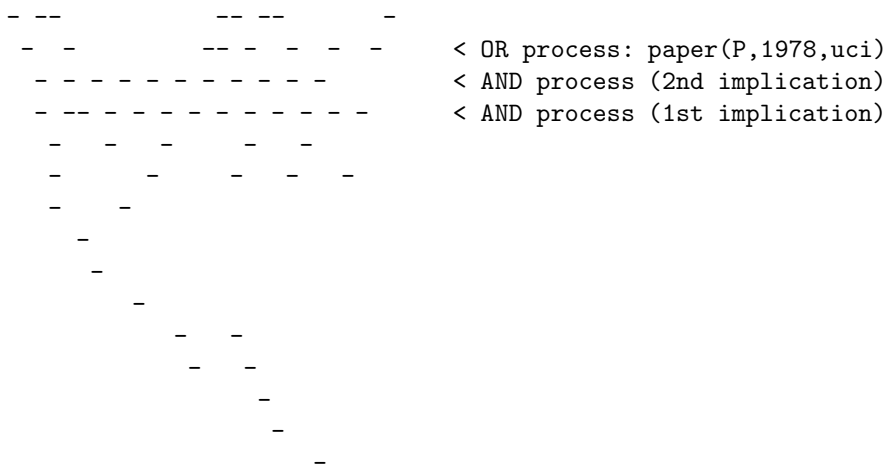
A complete interpreter based on the AND/OR Process Model, executing logic programs by decomposing them into AND and OR processes, has been implemented in DEC-10 Prolog. Since both AND and OR processes can be either parallel or sequential in nature, there are actually four different interpreters. All interpreters share the same kernel of scheduling procedures, performance measuring routines, message passing primitives, and other low level supporting code. The measurements and examples shown in the figures in this chapter are from the interpreter APOP (*A*nd *P*arallel – *O*r *P*arallel).

After it solves a problem, the interpreter prints out the number of processes created, and, for each process, the number and size of each kind of message sent. Associated with each process and each message is a "time stamp," represented as an integer. The interpreter is able to use this information to generate plots, such as the one in Figure 4.1, showing the relationship between the transformations performed on the processes. The vertical axis represents the number of processes. The horizontal axis represents time, with the transformations of one process plotted on one line. The interpreter records the fact that each transformation takes one time unit. When a message bearing time stamp t triggers a transformation causing other messages to be sent, the new messages will have time stamp $t + 1$. If the interpreter transforms process number P at time t, a dash will be plotted at coordinates (P, t). Note that if P sends a message to Q as part of the transformation plotted at (P, t), there will be a dash at $(Q, t + 1)$ as a transformation of Q is triggered by this message. The plot in Figure 4.1 was produced by the solution of

$\leftarrow$ `paper(P,1978,uci).`

with interpreter APOP and the program of Figure 2.2 (page 10).

The plots provide an estimate of the amount of parallelism possible. Wherever there are two dashes plotted for the same time (same column), there is the possibility that two processing elements could be performing




```
15 processes executed 62 steps in 28 time units:  2.22
Message Summary: 69 messages sent, using 573 characters.
```

Process	Succ/Size	Fail	Redo	Start	Cancel
1	3/91	1	3	1	0
2	3/85	1	2	2	0
3	1/35	1	4	5	4
4	1/58	1	5	6	5
5	4/93	1	0	0	0
6	4/93	1	0	0	0
7	1/23	1	0	0	0
11	1/26	1	0	0	0
12	1/19	1	0	0	0

Plot and message summary for all three solutions to ← `paper(P,1978,uci)` *(see program in Figure 2.2). The transitions of process 2, the parallel OR process for* `paper(P,1978,uci)`*, are explained in detail in Chapter 5. The transitions of process 4, the parallel AND process for one of the clauses in the definition of* `paper`*, are described in detail in Chapter 6. Processes 8–10 and 13–15 were edited out of the table; they all failed immediately, sending only the one fail message.*

Figure 4.1: Sample Interpreter Output

the corresponding state transitions in parallel. Dividing the total number of steps by the time required to generate a result is a measure of the amount of parallelism in the program. The plots are not based on an actual parallel execution. Such plots could only be generated by a system with an unbounded number of processing elements, each dedicated to solving just one process, where each processor is capable of passing a message to any other processor instantaneously. This interpreter was built to see if there is parallelism to be found in the execution of logic programs; there is if the ratio of steps to time is greater than one. The problem of mapping processes onto processing elements, of "distributing the dashes" for a parallel solution on a physical network of processing elements, will be discussed in Chapter 7.

4.6 Programming Language

Many of the extensions to the formalism of logic programming included in most Prolog systems are meaningful only in single processor, sequential systems. Most notable are `assert` and `retract`, goals that modify the database of clauses in the program, and the cut operation, used to guide the global search process.

The language supported by the AND/OR Process Model also has some extensions to the formalism, but only where no assumption is made about the number of processors available to interpret the program or about whether the processors have access to a common memory. The underlying hardware is presumed to be a collection of asynchronous, autonomous *processing elements*, or *PEs*, each with its own local memory and its own copy of the program being interpreted.

The extensions to logic programming supported by the interpreter are:

- The evaluable predicate `is`, for performing arithmetic operations (Section 2.4.1).
- Definition of $\leftarrow$ and $\wedge$ as infix operators, and the DEC-10 Prolog evaluable predicates `=..` and `call`, in order to describe higher order functions.
- Negation as failure.
- Conditional expressions.

The last two extensions, negation and conditional expressions, are implemented in Prolog through use of the cut symbol (Section 2.4.3). The same behaviors can be implemented in the AND/OR Process Model, but by using specially defined processes instead of cut.

Negation as failure is implemented by a special OR process for literals of the form `not(G)`. This OR process creates an AND descendant to solve `G`. If the descendant returns a fail message, the OR process sends `success(G)` to its own parent; if the descendant sends `success(G)` the OR process sends fail to its parent and cancel to the descendant. This definition of negation is comparable to the usual definition in Prolog, in the sense that it is a correct interpretation only when `G` is ground (contains no unbound variables).

Conditional expressions can be written in DEC-10 Prolog, using a right arrow and semicolon as infix predicate symbols:

```
f :- p -> q ; r.
```

The Prolog interpreter treats this operationally as as if it were defined by two separate clauses, the first containing a cut symbol:

```
f :- p, !, q.
f :- r.
```

One way of implementing conditional expressions in the AND/OR model also involves translation into two new clauses, but neither contains a cut symbol:

```
f ← p ∧ q.
f ← not(p) ∧ r.
```

Another method for implementing conditionals, by using special AND processes, is presented in Chapter 7.

4.7 Chapter Summary

The AND/OR Process Model provides a framework for execution of logic programs where an interpreter solves a goal by dividing it into independent pieces for solution by other interpreters. The independent interpreters are processes communicating with their parent processes via messages. The processes in this model are objects with state variables that are updated in an atomic operation when a message triggers a state transformation. The description presented in this chapter concentrated on the behavioral requirements of the processes, specifying how a process must respond to the different types of messages. A global perspective of the computation performed by these sequential processes would show the same computation carried out by a depth first interpreter, since the same inferences are generated, and in the same order.

A major difference between the process model and the depth first search is in the binding environments used to represent values of variables. In typical depth first interpreter, a highly intertwined, global stack of environments is built. The binding stack and its supporting data structures require a single

memory space in the underlying hardware. The AND/OR Process Model, on the other hand, presents a method for interpretation by small, asynchronous, and logically independent processes communicating only through messages. Variable bindings are localized, stored in the state variables of the processes. Thus the first step in the design of a highly parallel architecture for logic programs has been taken: it has been shown how a logic program can be executed by independent interpreters. The next step is to show how these interpreters can exploit parallelism by starting a number of processes to carry out subtasks simultaneously.

Chapter 5

Parallel OR Processes

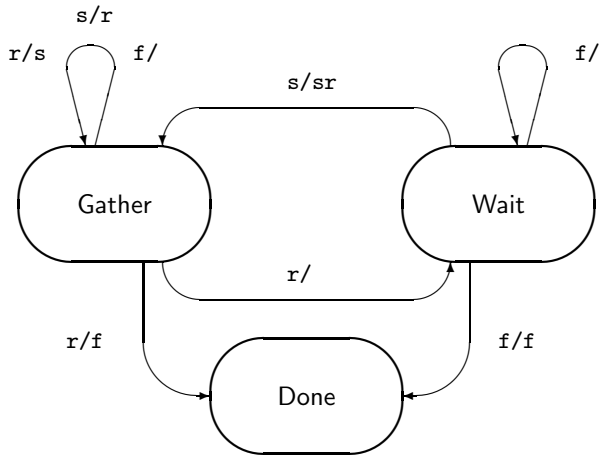
An OR process is an independent interpreter created to solve a goal statement of exactly one literal. An OR process created to solve an n-ary literal $\mathtt{p}(\mathtt{X}_1 \ldots \mathtt{X}_n)$ is expected to return every tuple in the set $\mathbf{D}_1(\mathtt{p})$, *i.e.* it must construct bindings for the variables of the literal through a proof of a goal containing only this literal.

If a procedure for `p` is defined by more than one clause, a sequential OR process operates like a depth first interpreter, returning results according to an order defined by the text of the program. All answers based on the first clause are returned before the first result from the second clause, and so on up to the last tuple of values defined by the last clause.

Relations are *unordered* sets of tuples, so the ordering of tuples according to the lexical order of clauses is not part of the meaning of a predicate. A parallel system could construct $\mathbf{D}_1(\mathtt{p})$ by interleaving tuples defined by the various clauses in the procedure. The parallel OR processes defined in this chapter attempt to construct answers based on all clauses simultaneously, returning results as the messages from the descendant processes arrive. The order of tuples depends only on the timing of the incoming success and fail messages from descendants.

5.1 Operating Modes

When an OR process is first created, it assumes its parent AND process is waiting for an answer. The first result constructed by the OR process should be sent via a success message to the parent. After this, however, the OR process should save the answers, not sending the next one to the parent until it receives a redo message. The OR process acts as a message center, deciding when to transmit results and when to store them.



A label `x/y` *on an arc means incoming message* `x` *triggers the transition, message* `y` *sent as a result of the transition. The choice of which transition to take depends on the type of incoming message and the values of internal state variables.*

Figure 5.1: Modes of an OR Process

An OR process is in *waiting mode* when its parent is waiting for an answer, and is in *gathering mode* when the parent is busy, using a result sent previously. Processes will switch back and forth between these two operating modes. The rules for changing from one mode to the other are based on the order of success and redo messages received, and on the number of tuples constructed but not yet sent to the parent.

5.2 Execution

The diagram of Figure 5.1 summarizes the transitions between waiting and gathering modes. State transitions in the figure have labels of the form X/Y, meaning the transition was triggered because the OR process received message X, and as a result of the transition it is transmitting message Y, where X and Y are either `f` for fail, `s` for success, `c` for cancel, or `r` for redo. The details of the rules for executing state transitions are given in the procedures of Figures 5.2 and 5.3.

Each state transition depends on the type of input message, the current operating mode, and values in other internal data structures. The internal state variables of OR processes contain unchanging information such as the

OR Process Algorithms

L: The literal being solved by the OR process.

DL, WL, SL, Mode: The state variables of the OR process.

To process a start message:

1. Initialize *DL*, *SL*, and *WL* to empty lists.
2. Repeat for each clause C in the procedure for L in which the head unifies with L:
   (a) If C is a unit clause, apply to substitution created during unification to the head of C and add this instance to *WL*.
   (b) If C is a non-unit clause, apply the substitution to C, start an AND process for the body of C, and add a record for this process to *DL*.
3. If *DL* and *WL* are both empty, send a fail message to the parent and terminate (no clause head matched L).
4. If *WL* is empty, set *Mode* to *waiting*; otherwise, remove one result from *WL*, add it to *SL*, send it in a success message to the parent, and set *Mode* to *gathering*.

Figure 5.2: Starting an OR Process

goal being solved and parent process ID, as well as dynamic information representing the state of the computation. The dynamic information consists of the following state variables:

DL: A list of descendant processes.

WL: A list of results waiting to be sent to the parent process.

SL: A list results already sent.

Mode: The operating mode, either waiting or gathering.

When the OR process receives a start message from its parent, it attempts a unification of its literal L with the head of each clause in the procedure for L. If there is no clause with a head unifiable with L, the OR process fails immediately. Otherwise, a descendant AND process is created for each implication with a head that unifies with L. If L unifies with the head of at least one unit clause, a success message based on the unit clause can be

sent to the parent process and the OR process goes into gathering mode, otherwise it goes into waiting mode.

Whenever a parallel OR process receives a success message from a descendant, it responds by sending the descendant a redo, causing it to immediately start working on its next answer. If the OR process was in waiting mode when it received the result, it sends it to its parent, otherwise it stores it in *WL*.

Whenever an OR process receives a fail message from a descendant, the corresponding item is removed from the descendant list. If the descendant sending the fail message is the last active descendant and the parent is waiting, the OR process fails.

By definition, an OR process is in waiting mode if its parent is waiting for an answer, so a redo message from the parent in this case signals a system error. Otherwise, if there are results in *WL*, one of them is sent back. If there are no results but there is a possibility of future successes, *i.e.* there are active descendants, the OR process switches to waiting mode. Finally, if there are no active descendants and no further results in *WL*, the process sends a fail message and terminates.

5.3 Example

The plot in Figure 4.1 (page 69) showed the complete solution of the goal

← `paper(P,1978,uci)`

Process 2 in that example was the OR process created to solve the single literal in this goal. In this section we will examine each state transition in the execution of process 2 in detail. The text of the program is given in Figure 2.2 on page 10.

Figure 5.4 shows the states of the process after each message it receives. In front of each state is a number enclosed in brackets used to identify the transition, the message that causes the transition, and the time stamp on the message. A summary of the state generated by processing the message is given on the following lines, showing the values of the state variables representing the mode (gathering or waiting), the descendant list *DL*, the waiting list *WL*, and the list of answers sent *SL*.

Each item in *DL* shows the ID of an active descendant process and the head and body of the clause used to create the descendant. If and when the descendant returns a success message, the argument of the message will be a copy of the body of the clause solved by the descendant. The argument is unified with a copy of the body in *DL*, thus instantiating the variables in the head; the instantiated head is used as the argument in the success message sent back to the parent of the OR process. Note that variables in the separate

OR Process Algorithms

To process a success message:

1. Send a redo message to the process that sent the success.
2. Create a success message M based on the argument of the incoming message.
3. If *Mode* = *waiting*, append M to SL, send M to the parent AND process, and set *Mode* to *gathering*. (Transition labeled `s/sr` from waiting mode in Figure 5.1.)
4. If *Mode* = *gathering*, append M to WL. (Transition labeled `s/r` from gathering mode.)

To process a fail message:

1. Remove from DL the entry for the failed process.
2. If *Mode* = *gathering*, or *Mode* = *waiting* and DL is not empty, no steps are taken. (Transitions labeled `f/` from gathering mode and `f/` from waiting mode.)
3. If *Mode* = *waiting* and DL is empty, send a fail message to the parent and terminate. (Transition labeled `f/f` from waiting mode.)

To process a redo message:

1. If *Mode* = *waiting*, signal a system error and terminate.
2. If WL is not empty, move a message from WL to SL and send it to the parent in a success message. (Transition labeled `r/s` from gathering mode.)
3. If WL is empty but DL is not, set *Mode* to *waiting*. (Transition labeled `r/` from gathering mode.)
4. If WL and DL are both empty, send a fail message and terminate. (Transition labeled `r/f` from gathering mode.)

Figure 5.3: State Transitions of an OR Process

```
<1> (after 'start' from Process 1, T = 1):
   Goal: paper(P,1978,uci)
   Mode: gathering
   DL:   3:[paper(P,1978,uci),[tr(P,uci),date(P,1978)]]
         4:[paper(P,1978,uci),[date(P,1978),author(P,A),loc(P,uci)]]
   SL:   [paper(xform,1978,uci)]
   WL:

<2> (after 'redo' from Process 1, T = 4):
   Goal: paper(P,1978,uci)
   Mode: waiting
   DL:   3:[paper(P,1978,uci),[tr(P,uci),date(P,1978)]]
         4:[paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   SL:   [paper(xform,1978,uci)]
   WL:

<3> (after 'success([date(eft,1978)...])' from Process 4, T = 14):
   Goal: paper(P,1978,uci)
   Mode: gathering
   DL:   3:[paper(P,1978,uci),[tr(P,uci),date(P,1978)]]
         4:[paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   SL:   [paper(eft,1978,uci),paper(xform,1978,uci)]
   WL:
```

Figure 5.4: States of a Parallel OR Process

clauses of *DL* are independent, so a success from one descendant does not interfere with the variables in the *DL* entries for other descendants.

The first transition occurs when the process receives the start message from its parent, process 1. Three clauses have heads that unify with the goal literal, `paper(P,1978,uci)`. The unit clause `paper(xform,1978,uci)` is one the clauses with a matching head, so the message

```
success(paper(xform,1978,uci))
```

is sent to process 1 and used to initialize *WL*. The process is in gathering mode in the output state. Processes 3 and 4 are created to solve the bodies of the other two clauses. *WL* is empty since there was just the one unit clause for `paper`.

The second transition is triggered by a redo message from the parent. There are no answers in *WL*, so the process goes into waiting mode until a message arrives from either active descendant.

Transition `<3>` occurs when one of the descendants, process 4, sends a success message. The body of the success message is unified with the body stored for the *DL* entry of process 4, creating the solution `paper(eft,1978,uci)`.

```
<4> (after 'success([tr(df,uci)...])' from Process 3, T = 15):
   Goal: paper(P,1978,uci)
   Mode: gathering
   DL:   3:[paper(P,1978,uci),[tr(P,uci),date(P,1978)]]
         4:[paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   SL:   [paper(eft,1978,uci),paper(xform,1978,uci)]
   WL:   [paper(df,1978,uci)]

<5> (after 'redo' from Process 1, T = 17):
   Goal: paper(P,1978,uci)
   Mode: gathering
   DL:   3:[paper(P,1978,uci),[tr(P,uci),date(P,1978)]]
         4:[paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   SL:   [paper(df,1978,uci),paper(eft,1978,uci),paper(xform,1978,uci)]
   WL:
```

Figure 5.4 (cont'd)

Since the parent is waiting, this answer is not put on *WL* but sent immediately to process 1 and appended to *SL*. Process 4 is sent a redo message, and the OR process goes back to gathering mode.

While the process is still in gathering mode, the other descendant sends a success. A solution based on this message is added to *WL*, the descendant is sent a redo message, and the process remains in gathering mode.

Transition <5> is triggered by a redo message from the parent. There is a solution ready for it in *WL*, so it is moved from *WL* to *SL* and sent to the parent.

The success message and redo message processed in transitions <4> and <5>, respectively, could have arrived in either order. If they had arrived in the opposite order than described above, the current state of the OR process would be the same. The intermediate state would be different – the process would be in waiting mode temporarily until the success message arrived – but the net result is the same, with the process in gathering mode waiting for a fourth solution, having sent the first three back to process 1.

The next message received is a redo message from the parent (transition <6>). *WL* is empty, so the OR process goes into waiting mode.

The next transition occurs when one descendant sends a fail message. The record of the descendant (process 3) is removed from *DL*, and the OR process remains in waiting mode.

Once again, the messages that trigger transitions <6> and <7> could arrive in either order, and the net result would be the same.

Finally, the last remaining descendant, process 4, sends a fail message.

```
<6> (after 'redo' from Process 1, T = 20):
   Goal: paper(P,1978,uci)
   Mode: waiting
   DL:   3:[paper(P,1978,uci),[tr(P,uci),date(P,1978)]]
         4:[paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   SL:   [paper(df,1978,uci),paper(eft,1978,uci),paper(xform,1978,uci)]
   WL:

<7> (after 'fail' from Process 3, T = 23):
   Goal: paper(P,1978,uci)
   Mode: waiting
   DL:   4:[paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   SL:   [paper(df,1978,uci),paper(eft,1978,uci),paper(xform,1978,uci)]
   WL:

<8> (after 'fail' from Process 4, T = 26):
   <done>
```

Figure 5.4 (cont'd)

There are now no more active descendants, and the OR process cannot derive any more solutions. It sends its parent a fail message and terminates.

5.4 Chapter Summary

From the perspective of an AND process, a parallel OR process is similar to the sequential OR process defined in the last chapter. Both are independent interpreters, created to solve a goal containing exactly one literal. If the goal is not solvable, an OR process returns a fail message. If the goal is solvable, the process responds with a success message carrying bindings representing one tuple from the denotation of the literal. Further results can be obtained by sending the process a redo message. After all results are returned, it sends back a fail message.

The difference between parallel and sequential OR processes is that parallel OR processes have multiple descendant processes active at any one time. They act as message centers, queuing success messages from their descendants in a list named *WL* until they are needed by their parents. The processes described here also save the success messages sent to the parent in a list named *SL*. The list is not strictly necessary, but it was included in the interpreter because it provides useful debugging information. It gives a good indication of the state of the process, in terms of how far along the process is in the

complete solution of its goal.

In the protocol described here, whenever a descendant sends a success message, the OR process immediately responds with a redo message. An interesting project for the future would be to experiment with other protocols as a means for controlling the amount of OR parallelism. When the system is busy, and *WL* contains enough answers, the OR process could simply wait, sending the redo at a later time.

Sequential OR processes are sensitive to the order of the clauses in a program. The order of the answers sent to a parent is a function of the relative order of the clauses, and the results are always generated in the same sequence. The order of results from a parallel OR process depends on the order of arrival of messages from its descendants, and could vary from one execution to the next.

One of the ramifications of interleaving results from different clauses is that parallel OR processes may return more results than sequential interpreters. A simple example is in a program containing the clause

```
p ← p.
```

Any tree of resolvents containing a node with a call to `p` will have an infinite branch. If there are other clauses following this one in the procedure for `p`, derivations based on these clauses will be to the right of the infinite branch. An interpreter performing a depth first search will be caught in an infinite loop, never returning any results based on the remaining clauses. Sequential OR processes will also be trapped in an infinite computation by this clause: the body is used to start an AND process for the body, in this case the single call to `p`; the AND process will start an OR process to solve `p`, and the cycle repeats, generating an infinite branch in the dynamic AND/OR tree of processes.

Parallel OR processes, on the other hand, are able to derive solutions in this case. A parallel OR process creates the same infinite subtree as a sequential OR process, but the parallel process obtains solution from AND processes corresponding to other clauses for `p` or other clauses in the procedure that called `p`. The AND/OR tree is expanded in parallel below an OR process, and even though one of the subtrees may be infinite, other subtrees to the right can provide solutions.

However, the use of parallel OR processes is not a guarantee that all provable results will be generated. It is still possible to write a set of clauses where the null clause is derivable through a series of resolutions, and the control strategies of the AND/OR process model are not capable of deriving any solution. These pathological cases will be described at the end of the next chapter, since they concern the method for solving literals in the body of a clause in parallel.

Chapter 6

Parallel AND Processes

Sequential AND processes, as defined in Chapter 4, simply mimic sequential interpreters by solving their subgoals one at a time, from left to right. AND parallelism is exploited in the AND/OR Process Model by having an AND process create more than one OR process in certain transitions, and then coordinating the responses to success and fail messages from these descendants until all literals have been successfully solved.

A "brute force" method for AND parallelism in this model would be to immediately create an OR process for every literal. There are three reasons why this will not be effective, all based on the fact that solution of one literal often binds variables in other literals.

The first problem is the requirement that every occurrence of a variable must be bound to the same term in any solution. For example, given the goal statement

$$\leftarrow \texttt{p(A,B)} \wedge \texttt{q(B,C)} \wedge \texttt{r(C,A).}$$

the AND process has to find tuples `<A,B,C>` that satisfy all three predicates at the same time. When OR processes for the three literals are working independently, they may bind the variables to conflicting terms. A solution for `p(A,B)` might be `p(0,1)`, and a solution for `q(B,C)` might be `p(2,3)`; these are valid by themselves, but do not constitute a solution of the entire goal statement. The AND process must signal the OR processes to continue until a consistent binding is found for `B`.

A second argument against solving all literals at once is that by postponing the solution of a literal until some of its variables are bound via the solution of other literals, the corresponding OR processes may be more efficient: there are often fewer solutions, and fewer fruitless choices made in constructing those solutions (Section 2.5).

Finally, and of most practical importance, some goals fail if an attempt is made to solve them before a sufficient set of variables are instantiated. These are the literals with thresholds or mode declarations (Section 2.4.1). For example, in the goal statement

```
← length(L,N) ∧ X is 2 * N.
```

the goal of multiplying `N` by two fails unless `N` is instantiated to an integer in the solution of the first goal. In this case it makes sense to postpone solution of the second goal until the first is complete.

In the context of the AND/OR Process Model, then, an effective method for achieving AND parallelism is a problem of correctly ordering the literals in the body of a clause, of deciding which literals must be solved sequentially and which can be done in parallel. The implementation of AND parallelism defined in this chapter has three major components. There is an *ordering algorithm* that automatically decides, based on the current state of the goal list, the solution order of the literals. The *forward execution* component actually creates the descendant OR processes; it handles success messages, determining which literals can be solved as a result. The third component, known as *backward execution*, handles fail and redo messages, deciding which literal(s) must be re-solved before continuing forward execution.

6.1 Ordering of Literals

The basis for the ordering of literals in the body of a clause is the sharing of variables. Whenever two or more literals have a variable in common, one of the literals will be designated the *generator* for the variable, and it will be solved before the others. The solution of the generator literal is intended to create a value for the corresponding variable. After the generator has been solved, the other literals containing the variable, the *consumers*, may be scheduled for solution. A generator will be defined for every variable in a goal statement. It is possible that the solution of a generator will not bind the variable, and consumers will still have a variable in common; this situation is discussed in Section 6.2.

Generators and consumers are similar to the lazy producers and eager consumers of IC-Prolog [18]. The term "generator" is used here, since their action is more closely related to generators in other languages (see, for example, Alphard [101]). They produce a *sequence* of independent terms, as opposed to parts of a single complex term through a series of partial bindings. Note that a literal can be the generator of some variables and a consumer of others. This is especially true when the literal is a function call, when some of the variables are bound to input arguments and the others, uninstantiated

when the call is made, will be bound to output values by the execution of the function.

There are similarities in the scheduling of literals in the AND/OR Process Model and the Parlog and Concurrent Prolog languages discussed in Chapter 3. On the surface, it appears those systems do not require literals to be ordered, since processes are started for all literals simultaneously when the clause is invoked. However, some of those processes immediately block, waiting for input via the solution of other literals. In the AND/OR Process Model, we delay creation of processes for the literals that would block immediately. The mechanisms are different, but the basis for scheduling is the same. On the one hand a process is created and then blocked until a shared variable is bound, on the other the process is not created until the processes it depends on have completed.

The differences reflect the different styles of programs the systems are designed for. Parlog and Concurrent Prolog programs are either deterministic, where it is known that processes will not fail, or the nondeterminism is of the committed choice variety. The emphasis is on sending values to consumers as quickly as possible, while overlapping execution with producers when the values are large structures. The AND/OR Process Model is oriented toward nondeterministic programs. Solution of nondeterministic goals requires a complex pattern of messages between literals, routed through the AND process. The explicit ordering is used to control operations in both forward and backward execution.

6.1.1 Dataflow Graphs

Generator and consumer relationships can be represented by a *dataflow graph*. In these graphs there is one node for each literal in a clause, and a set of directed arcs for each variable. The arcs go from a generator to each literal that consumes the corresponding variable. An *immediate predecessor* of a literal L is a generator for one of the variables in L. A predecessor, in general, is either an immediate predecessor or a predecessor of an immediate predecessor. Successors and immediate successors are defined similarly.

The head of the clause corresponds to the literal being solved by the parent OR process, and is included in the dataflow graph. It is the generator of every variable occurring in the head that is bound when the process is created, and the consumer of each of its unbound variables. There are two nodes in the graph for the head literal. The node labeled `HG` represents the head in its role as generator, and the node labeled `HC` is the head as consumer. Representing the head with two nodes serves two purposes, both leading to simpler algorithms for forward and backward execution: it allows graphs to be acyclic, and we avoid having to treat the head as a special case. Since the algorithms use this information, the head is included in the state information

of a parallel AND process and will be drawn in pictures of the graph.

6.1.2 The Ordering Algorithm

There are a number of rules one can use to identify generators. The first, mentioned above, is the head of a clause is the generator for all variables instantiated when the clause is invoked.

Second, some of the literals in the body may have I/O modes (Section 2.4.1). The modes of evaluable predicates are already known by the system, and mode declarations may be supplied for user-defined functions as part of the program. A good example is the evaluable predicate `is` from DEC-10 Prolog, with the mode declaration

```
mode(is,[?,+]).
```

This declaration shows that goals with predicate `is` have two arguments. A plus means the corresponding term must be a ground term when a goal for `is` is solved. In other words, `is` can never be the generator for a variable occurring in a term in this argument position. A minus (not shown here) means the corresponding argument must be an uninstantiated variable that will be bound during solution of the literal. In this case, we know the literal is the generator of the variable. If a variable occurs in "minus mode" in more than one literal it is an error, detectable at compile time. A question mark in a mode declaration means the mode is neither plus nor minus, *i.e.* the literal can be either a producer or a consumer. Given the above mode declaration and the goal

```
← ... A is B + C ...
```

a system may designate this call to `is` as the generator of `A` (but it is allowed to choose another goal containing `A`), but this call cannot be the generator for either `B` or `C`.

The two rules just described – the head is the generator of variables that are bound when the procedure is called, and mode declarations cannot be violated – are the only two strict rules for assigning generators. If the constraints imposed by these rules are violated, the clause will fail. By themselves, however, the two rules are not sufficient to designate generators for every variable in the body of a clause, since we do not always have mode information and clauses are often called without any head variables bound. The two strict rules can be augmented by heuristics in order to complete the process and make sure every variable has a generator. Different heuristics lead to different orderings, enabling more or less parallelism in the solution, but in all cases the AND process will not fail due to incorrect ordering.

A number of heuristics given in Section 2.5 for efficient sequential execution can be adapted for ordering literals in a parallel AND process. One

The Literal Ordering Algorithm

B: The set of literals remaining to be ordered.

G: The set of variables for which generators have been specified.

U: The set of variables for which generators are not known.

Static Analysis (when a clause is loaded into the system):

1. Initialize G to be the empty set, and U to a set containing every variable in the clause.
2. For each literal L which has a set of arguments A in positions marked with mode `-`, if any variable in A is in G, signal a mode violation. Otherwise assign L as the generator of all variables in A, and move the variables from U to G.
3. For each literal L which has an argument with a mode declaration of `+`, declare L to be a non-generator of the corresponding variable.

Dynamic Analysis (after a start message):

1. Initialize B to contain every literal in the body of the clause which is not a generator (as defined in the static analysis).
2. For each variable V bound in the head, if a body literal is the generator of V, signal a mode violation, otherwise add V to G, remove it from U, and assign the head as generator of V.

   In the following steps, a *qualified* literal is a literal which contains a variable V currently in U but is not a non-generator of V.
3. *(Apply heuristics).* Repeat until $B = \emptyset$ or $U = \emptyset$:
   (a) *(Connection Rule).* Make a set of literals LS with every qualified literal in B which has at least one variable in U and one variable in G.
   (b) *(Leftmost Rule).* If LS is empty, let LS be the singleton set containing the leftmost qualified literal in B.
   (c) If LS is empty, terminate with failure. Otherwise, for every literal L in LS, assign L as the generator of every variable occurring in L which is also in U. Remove these variables from U and add them to G, and remove L from B.

Figure 6.1: The Literal Ordering Algorithm

heuristic currently implemented is the *connection rule*, a special case of the rule that calls for selection of the literal with the largest number of instantiated variables. When the connection rule is applied, G is the set of variables with generators already designated, and U is the set of variables without generators. The generators of the variables in G form a partial graph. The connection rule attempts to "connect" a literal to the partial graph by finding a literal containing variables from both G and U. This literal will be designated the generator for all its variables in U (Figure 6.1).

The last rule currently implemented is the *leftmost rule*, which simply says the first (leftmost) literal containing a variable in U should be the generator of that variable. This is a reasonable heuristic, since, in Prolog programs, solution of a goal containing unbound variables often binds the variables. When the leftmost rule selects the leftmost occurrence of a variable in U it is selecting a literal that would see the variable as unbound if the clause was solved with the ordering generated so far. This rule is also a useful safety feature, since, by itself, it guarantees every variable will have a generator. If all other rules fail to designate a generator, this one can be applied.

Since mode declarations are known before a clause is called, the second rule can be applied at "compile time," when the clause is first loaded into the system. The other rules are applied at runtime, when the AND process is created, since they depend on the pattern of variable instantiation in the clause and this is established by the unification done by the parent OR process. As mentioned in Chapter 3, literal ordering is done completely at compile time and all dynamic ordering is avoided in the systems of Chang, Despain, and DeGroot [10] and DeGroot [25].

An important requirement for the forward execution algorithms is that dataflow graphs need to be acyclic. If mode declarations do not specify a cycle in the graph, the dynamic steps will make an acyclic graph. If there is a cycle of negative mode declarations, an acyclic graph is impossible. If there is a cycle of positive mode declarations, an acyclic graph is possible only if literals outside the cycle bind each variable in the cycle. For example, in `p(X,Y)` $\wedge$ `q(Y,X)` if `p` and `q` have plus modes for their first arguments, a cycle of positive modes exists: `p` cannot bind `X` and `q` cannot bind `Y`. An acyclic dataflow graph is possible only if other literals in the body become generators of `X` and `Y`. The check for an empty set of literals in the last step of the ordering algorithm is a hedge against negative cycles. If such cycles are detected at compile time, this check can be omitted.

6.1.3 Examples

The ordering algorithm will be illustrated by four examples, each showing a different pattern of variable instantiation in the body of a clause.

Call:
$\leftarrow$ `f(A,B).`
Clause:
`f(X,Y)` $\leftarrow$
`g(X)` $\wedge$ `h(Y).`

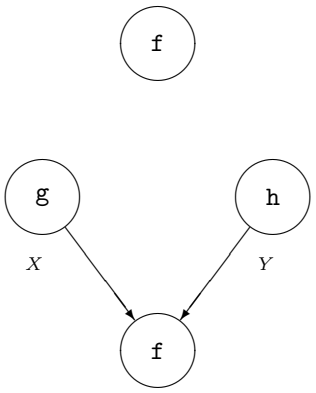


Figure 6.2: Graph for Disjoint Subgoals

Disjoint Subgoals

`f(X,Y)` $\leftarrow$ `g(X)` $\wedge$ `h(Y).`

If neither `X` nor `Y` is bound when the clause is called, or if they are bound to terms not containing a variable in common, the literals are independent. Neither is a predecessor of the other, as shown in Figure 6.2 for the case when both `X` and `Y` are uninstantiated when the process is created. The leftmost rule was used to designate `g(X)` as the generator of `X` and `h(Y)` as the generator of `Y`. Note that if there are n_g solutions for `g(X)` and n_h ways of solving `h(Y)`, then $\mathbf{D}_1$(`f`) will contain $n_g \times n_h$ pairs of `X` and `Y` values. The remaining pairs, after the first, will be created in response to redo messages; the method used to enumerate all pairs is described later in the section on backward execution.

Shared Variable

`gf(X,Z)` $\leftarrow$ `f(X,Y)` $\wedge$ `p(Y,Z).`

The two subgoals have the variable `Y` in common, and no call to `gf` can ever cause `Y` to be instantiated when a process is started. If, when the AND process is created, `Z` is instantiated but `X` is not, the connection rule selects `p(Y,Z)` as the generator of the shared variable `Y`. Otherwise `f(X,Y)` is designated, either through the connection rule (if only `X` is instantiated) or the leftmost rule (if neither or both head variables are instantiated). The graph in Figure 6.3 shows the graph built when `Z` is bound but `X` is not. This is an example where the connection rule leads to an efficient ordering described in Section 2.5, based on the number of instantiated variables in each literal in a database query style program.

Call:
← `gf(G,a).`
Clause:
`gf(X,Z)` ←
`f(X,Y)` ∧
`p(Y,Z).`

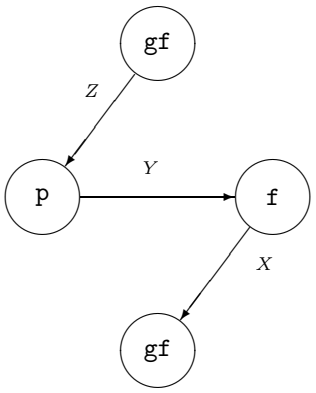


Figure 6.3: Graph for Shared Variables

Deterministic Function

```
f(P,Q) ← div(P,P1,P2) ∧ f(P1,Q1) ∧ f(P2,Q2) ∧
         comb(Q1,Q2,Q).
```

This clause illustrates the general form of a "divide and conquer" style function expressed as a clause. On every call, `P` will be bound to a term representing the input problem, and as a result of the call `Q` will be bound to a term representing the output of the function. The optimal ordering of subgoals is: divide problem `P` into independent subproblems `P1` and `P2`; then solve `P1` and `P2` in parallel via the recursive calls, instantiating `Q1` and `Q2`; when both are done, construct answer `Q` from partial answers `Q1` and `Q2`. This sequence of events is implied by the picture in Figure 6.4. In the next section, on forward execution, we will see how the AND process actually carries out the parallel actions in this order. This graph can be produced by repeated application of the connection rule; mode declarations are not required. In general, however, mode declarations are needed to produce the desired ordering for functions.

It is interesting to note that if `f`, `div`, and `comb` are relations, not functions, and it is meaningful to call `f` with `Q` bound and `P` unbound, a similar graph is built, the only difference being the direction of all the arcs. In other words, if we call this procedure to "run backwards" to solve for `P` given `Q`, the execution according to the graph we generate will be the same as traversing "backwards" in the graph of Figure 6.4.

Map Coloring

```
color(A,B,C,D,E) ←
```

Call:

```
← f(xx,Q).
```

Clause:

```
f(P,Q) ←
  div(P,P1,P2)
∧
  f(P1,Q1) ∧
  f(P2,Q2) ∧

comb(Q1,Q2,Q).
```

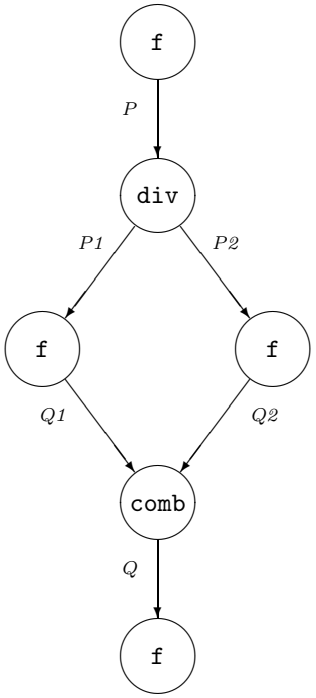


Figure 6.4: Graph for Deterministic Function

```
next(A,B) ∧ next(C,D) ∧ next(A,C) ∧ next(A,D) ∧
next(B,C) ∧ next(B,E) ∧ next(C,E) ∧ next(D,E).
```

The goal of this procedure (Figure 6.5) is to see if there is an assignment of one of four colors to each region of a map, such that no two adjacent regions have the same color. The procedure for `next` is simply twelve ground assertions, one for each legal pair of adjacent colors. For example, `next(red,blue)` is asserted, but `next(green,green)` is not. Also, for each clause `next(`c_1`,`c_2`)` we need the corresponding clause `next(`c_2`,`c_1`)`. Assuming every map can be colored by four colors, then, the procedure for `next` has twelve clauses.

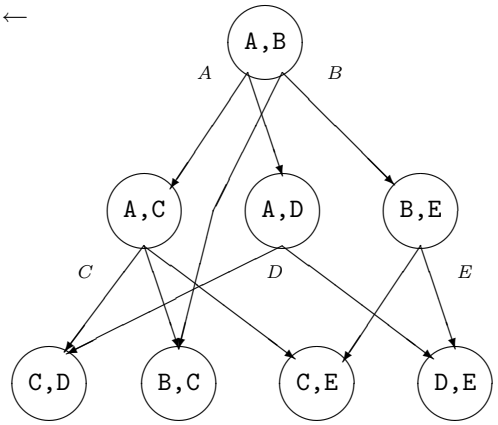
The calls to `next` will succeed only if the arguments have been (or can be) instantiated to terms representing different colors. There is one call to `next` for each internal border in the map. This formulation of the map coloring problem as a logic program was originally given by Pereira, Porto,

Call:

```
←
color(A,B,C,D,E).
```

Clause:

```
color(A,B,C,D,E) ←
  next(A,B)
∧
  next(C,D)
∧
  next(A,C)
∧
  next(A,D)
∧
  next(B,C)
∧
  next(B,E)
∧
  next(C,E)
∧
  next(D,E).
```



All nodes are calls to `next`*; the labels show the arguments of the call. The head node* `HG` *does not generate any values. The nodes in the top two rows are all immediate predecessors of* `HC`*.*

Figure 6.5: Graph for Map Coloring

and Bruynooghe in their papers on intelligent backtracking [7, 73, 74].

When this procedure is called with none of the variables in the head instantiated, the graph in Figure 6.5 is created. The literal ordering shown in the figure was produced by first using the leftmost rule to designate `next(A,B)` as the generator for both `A` and `B`. After that the connection rule was used to identify the three literals in the middle row of the graph as generators of the other three variables, leaving the remaining four literals as consumers. The role of non-generator literals in this problem is to check if the assignments made by generators are valid for the rest of the map.

Not unexpectedly, the first values from the generators in the middle row form an unacceptable combination of values for some of the consumers on the bottom row. The third and fourth literals, working independently and in parallel, assign the same color to regions `C` and `D`, so one of the assignments will have to change. There are a number of difficult problems presented by this example, as the AND process tries to coordinate the four generators in order

to create, eventually, every five-tuple of colors that satisfy the constraints of this goal list. The general principles for solving this problem will be explained in the section on backward execution. A more detailed description of a parallel solution of this problem will be given in Chapter 7.

6.2 Forward Execution

A forward execution step in an AND process consists of selecting certain literals for solution and starting OR processes for those literals. When a success message arrives from one of the processes, OR processes can be started for other literals. After all literals are solved, the AND process sends a success message to its own parent. If any of the descendant OR processes fail, or if the parent sends a redo message, then the backward execution algorithm is invoked (Section 6.3). The overall goal of a parallel AND process is the same as a sequential AND process – obtain a success message from the OR process for each literal in the body. What differs is the order the processes are created, and the handling of fail messages.

6.2.1 Forward Execution Algorithm

Forward execution can be explained with a mixed metaphor of graph reduction and dataflow. Graph reduction, as a technique for executing functional programs, refers to the reduction of a graph representing an expression to a simpler graph representing the value of the expression; typically the simpler graph is a single node. Here, we reduce the graph to an empty graph, containing no nodes or arcs. The reduction step corresponds to resolving away a literal from the body of the clause. We are interested in the substitutions required to perform the reductions, but not the final form of the reduced graph.

The dataflow metaphor is used to explain the passing of information between literals. When a generator is solved, the bindings for its variables "flow" to other literals containing the variables. Also, the firing rules of dataflow are used to order the solution of the literals. A literal is *enabled* when it has it has received values on all of its incoming arcs, *i.e.* when all of its predecessors have been resolved away. As soon as a literal is enabled, an OR process is created to solve it. This firing rule explains the need for an acyclic graph; if there is a cycle in the dataflow graph, no literal in the cycle can become enabled.

Literals in the body of a parallel AND process can be in three states: *blocked*, *pending*, or *solved*. A literal is blocked when an OR process has not yet been created for it. A literal is pending when an OR process has been created for it but has not yet sent back any message. Finally, a literal is in

The Forward Execution Algorithm

G: The dataflow graph. *pred(L)* is the set of predecessors of literal L, and *succ(L)* is the set of successors of L in G.

Solved, Blocked, Pending: State variables of the AND process, representing the status of each literal.

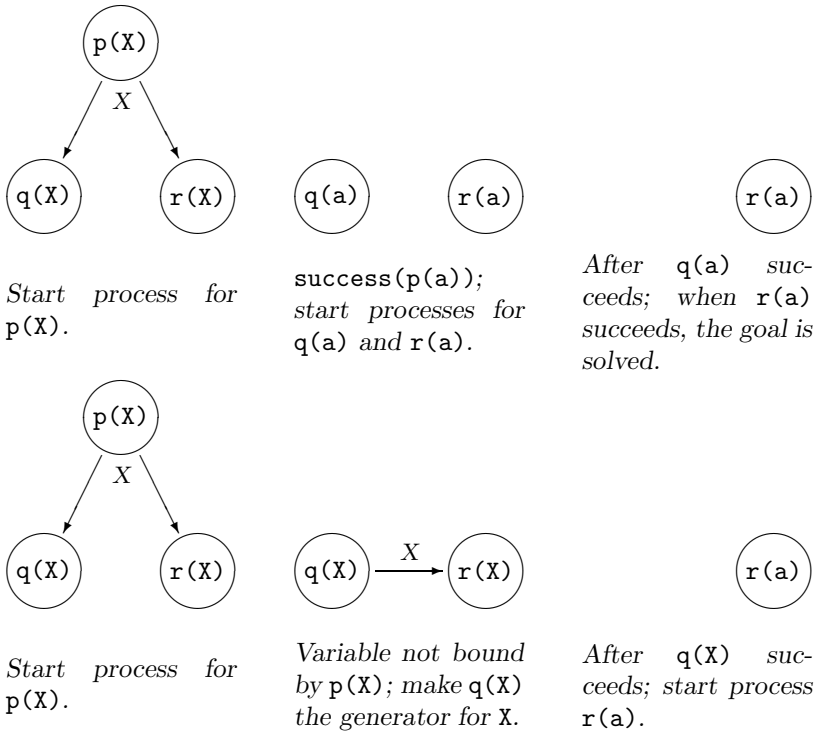
To process a start message:

1. Call the literal ordering algorithm to create an initial graph G.
2. Initialize *Solved* to {`HG`}, *Pending* to the empty set, and *Blocked* to the set containing every literal in the body.
3. For every literal L such that $pred(L) \subseteq Solved$ start an OR process and move L from *Blocked* to *Pending*.

To process a success message:

1. Apply the bindings contained in the message to the body.
2. Move the solved literal from *Pending* to *Solved*.
3. If all literals are in S, send a success message to the parent process, else continue.
4. If the solved literal is a generator, and the terms bound by the generator contain unbound variables, apply the heuristics of the literal ordering algorithm to designate generators for these variables and update the dataflow graph.
5. For each literal $\{L | L \in Blocked, pred(L) \subseteq Solved\}$ start an OR process and move L from *Blocked* to *Pending*.

Figure 6.6: Forward Execution Algorithm



Two different sequences of reductions of the goal ← `p(X)` ∧ `q(X)` ∧ `r(X)`. *In each case,* `p` *is the generator of* `X`. *In the bottom sequence,* `p` *does not bind* `X` *to a ground term, so another generator has to be assigned for* `X`.

Figure 6.7: Sample Graph Reductions

the solved state after an OR process has been created for it and the process has sent back a success message. The status of a literal can be represented by using sets to represent the blocked, pending, and solved literals. An enabled literal is a blocked literal with all predecessors in *Solved*, in other words *pred(L)* ⊆ *Solved*. When the process is initialized, all body literals are placed in *Blocked*, *Pending* is empty, and *Solved* contains only `HG`, the head of the clause.

A success message from an OR process for a literal L has the general form `success(`$L\theta$`)`, where $L\theta$ is a copy of L with (possibly) some variables bound. The forward execution step – the graph reduction operation – is accomplished by resolving $\neg L\theta$ with the current set of literals in the body of the clause. If L is a generator of a set of variables, then some of those variables may

be instantiated in $L\theta$. Using the imagery of the dataflow graph, we can envision tokens flowing along arcs from L to the consumers, as the resolution of $L\theta$ with the remaining literals causes those variables to be bound in the resolvent.

The algorithm in Figure 6.6 shows the heuristics of the literal ordering algorithm being applied after every success message. This is necessary for cases when a generator does not bind its variable `V`, or else binds it to a non-ground term containing a variable `V2`, for example when a goal `p(V)` is solved by a unit clause `p(f(V2))`. When there is more than one consumer of `V`, they will have a common variable in `V2` after the generator is solved. Since literals with variables in common are not solved in parallel, and since every variable must have a generator, the ordering algorithm must be called again to select one of the consumers of `V` to be the generator for `V2` (Figure 6.7). When the generator binds `V` to a ground term, one that contains no variables, this step can be omitted. The current implementation of the interpreter makes the simplifying assumption that generators bind variables to ground terms, and all subsequent discussions will be based on this assumption.

6.2.2 Solution of a Deterministic Function

The combination of literal ordering and forward execution is sufficient for parallel solution of clauses implementing deterministic functions. The distinguishing characteristics of clauses for deterministic functions are that there is only one solution for each combination of inputs, the clause does not fail when given a legal combination of input values, and the subgoals in the body are also deterministic functions. These attributes mean AND processes for deterministic functions never invoke the backward execution algorithm.

Matrix multiplication is a good example of a deterministic function that has a parallel solution. A logic program for this function is shown in Figure 6.8. The head of the procedure is `mm(A,B,C)`. When called, `A` and `B` will be bound to terms representing matrices, and after the call, `C` will be instantiated to their product. A matrix is represented as a list of rows, where a row is represented by a list of integers.

The top level of the function is a call to transpose one argument, followed by a call to a procedure that actually multiplies the matrices. A call `transpose(B,BT)` binds `BT` to the transposed version of `B`; `BT` it is a list of columns instead of a list of rows. After transpose succeeds, the problem is to distribute all possible pairs of rows of `A` with columns of `BT` to the inner product function. This is done by the two auxiliary functions, `mmt` and `mmc`. The internal structure of these two procedures is identical: there are two literals in the body of each; one literal is a call to a lower level function with the first element of the input list, while the other literal is a recursive call with the remainder of the list. The dataflow graphs for both functions

To multiply two matrices, transpose the second, then form all inner products:

```
mm(A,B,C) ← transpose(B,BT) ∧ mmt(A,BT,C).
```

Pair up rows of `A` *with columns of* `B`*, compute inner product:*

```
mmt([],_,[]).
mmt([A1|An],B,[C1|Cn]) ← mmc(A1,B,C1) ∧ mmt(An,B,Cn).
mmc(_,[],[]).
mmc(A,[B1|Bn],[C1|Cn]) ← ip(A,B1,C1) ∧ mmc(A,Bn,Cn).
ip([],[],0).
ip([A1|An],[B1|Bn],C) ← ip(An,Bn,X) ∧ C is X + A1*B1.
```

To transpose a matrix, call `columns` *to divide it into two parts: the first column and the rest of the columns; then transpose the rest.*

```
transpose([[]|_],[]).
transpose(M,[C1|Cn]) ←
       columns(M,C1,Rest) ∧ transpose(Rest,Cn).
columns([],[],[]).
columns([[C11|C1n]|C],[C11|X],[C1n|Y]) ←
       columns(C,X,Y).
```

Figure 6.8: Program for Matrix Multiplication

have two independent literals that can be solved simultaneously. The inner product function used here is sequential in nature, since the results of the multiplications are summed serially.

In analyzing the bodies of `mmt` and `mmc` we see the recursive call can be done at the same time as the call to the lower level function, so the time required to solve a problem of size n is proportional to the time required to solve the largest subproblem, rather than the sum of times to solve both subproblems. The time required to compute the product of the two matrices in a call to `mmt` is the time required to distribute the last of the row/column pairs to process that performs an inner product, plus the time required to do that inner product. For the multiplication of $n \times n$ arrays, this is $O(n+n+n)$, or $O(n)$. The parallelism seen here is identical to the parallelism obtained in a dynamic dataflow system [37]. The plots produced by the interpreter showed the ratio of number of steps executed per simulated time increasing at a rate proportional to n^2. This example supports the claim that parallelism in deterministic functions can be exploited by the AND parallelism of the AND/OR Process Model.

6.3 Backward Execution

Backward execution coordinates the actions of generators after the AND process receives a fail or redo message. Backward execution in a parallel AND process replaces the linear backtracking of a sequential AND process. The purposes of each are the same: after a failure, determine which literal should be re-solved, and send the corresponding OR process a redo message. The difference is that backward execution is directed by the dataflow graph of the clause and not the syntactic order of the clause body. The result is a more efficient system and a method that works well in a parallel system.

6.3.1 Generating Tuples of Terms

In Chapter 2, an interpreter for logic programs was defined as a system to construct the tuples of the denotation of a goal list. The tuples in the denotation are a subset of all possible n-tuples of terms; the role of backward execution is to coordinate the generators so that every tuple in the denotation is produced, if possible, while at the same time generating as few extraneous tuples as possible. A perfect method would generate only the tuples in the denotation of the goal statement.

A new tuple is formed each time a generator binds a variable to a new term. The key to an efficient backward execution method is identifying the correct generator for making the next tuple; the proper choice can cut out many erroneous tuples. Nested for-loops in a procedural language provide one model for identifying this generator. As an example, consider a nested loop implementation of the map coloring problem of Section 6.1.3:

```
for A := Red to Blue do
 for B := Red to Blue do
  for C := Red to Blue do
   for D := Red to Blue do
    for E := Red to Blue do
     if Next(A,B) and ...  and Next(D,E) then
      Writeln('success(A,B,C,D,E)');
```

In this program, initial values are assigned to all variables, making the initial tuple `<red,red,red,red,red>`. At each step, the current tuple is tested by the boolean expression in the body of the loop. The second tuple is created by assigning the innermost variable, `E`, its next value. Eventually, `blue`, the last value, is assigned to the innermost variable. The next tuple is obtained by *resetting* the variable `E` to its first value while assigning the next-innermost variable `D` its next value. In general, whenever there are no more values for a variable, the previous (outer) variable is given a new value

and all later variables closer to the body of the loop are reset to their initial values.

The drawback to this procedure for generating tuples is the large number of invalid tuples. All 5^4 5-tuples of colors are generated, where the first $3^4 = 81$ have the form `<red,red,C,D,E>`. Since `A` and `B` cannot have the same color – there is a literal `next(A,B)` in the test – the Pascal program blindly generates 81 unusable tuples. In the logic program, however, the same procedure used to test terms can be used to generate them, meaning "obviously" wrong tuples are never generated. The term `next(A,B)` is the generator of `A` and `B`, and it never instantiates both `A` and `B` to the same color, thus effectively preventing the construction of a large number of useless tuples. The Pascal version can be made more efficient by moving tests closer to the head of the loop as possible, but it still cannot be as efficient as Prolog, in terms of the number of tuples generated.

The backward execution algorithm described here borrows the concepts of ordering variables and resetting inner variables to previous values, and incorporates them in a method that retains the advantages of the Prolog implementation. It is more efficient than Prolog's linear backtracking. By selecting a backtrack literal according to the dataflow graph instead of the program text, we get an effect closely related to intelligent backtracking (Section 2.5.3). Further, by knowing when to reset a variable, we can sometimes avoid recomputing its value.

6.3.2 Definitions for Backward Execution

The backward execution algorithm makes extensive use of the dataflow graph of a clause when deciding which generator to select for a redo message and which generators should be reset. The algorithm uses "traditional" relations among literals, such as predecessor and descendant relations, but it also uses some special purpose relations.

All of the relations are static with respect to the dataflow graph. Once the graph is constructed, the relations can be determined. If the graph is constructed completely at compile time, which is possible when enough mode declarations are provided, one data structure can be built and shared by all AND processes that solve instances of the same clause body.

We need a representation for the literals when they are used as arguments of the relations. The technique used is to refer to a literal by a term `#N`, where `N` is the place the literal occupies in the text of the clause. For example, in the clause for the map coloring problem in Figure 6.5 on page 92, `#2` refers to `next(C,D)`.

The first new relation is defined in terms of a *linear ordering* of generators. We will say literal L_1 *follows* L_2 if L_1 occurs later in the linear ordering than L_2. The only constraint on the ordering is that if literal L consumes a

variable V, L must follow the generator of V in the linear ordering. Given this constraint, either a breadth first or preorder traversal of the graph is a reasonable way to generate the linear ordering. The current implementation uses a breadth first traversal.

When a literal L fails, one of the generators must compute new bindings for its variables. While it would seem the predecessors of L in the dataflow graph are the only candidates, in fact we must consider a larger set. *candidates(L)* is the set of generators considered for selection as the backtrack literal when L fails. G is a candidate for L if G is an immediate predecessor of L or a predecessor of a successor of L. Examples showing the reasoning behind this extended definition will be given later.

In addition to the relations between literals, we need to add a new data structure, a set of *marks* for each generator, to the representation of an AND process. During backward execution a generator will be marked with the identity of a literal if the generator is a possible cause of the failure of the literal.

The reset operation borrowed from the nested loop model must effectively restart a generator. The generator does not have to produce the values in the same order after a reset; it only has to provide all possible values when they are required. Suppose a generator has created the bindings $t_1 \ldots t_i$ for a variable at the time it is reset. The reset may come before the generator is finished, *i.e.* the entire sequence might contain $n > i$ terms. After the reset the generator must start over on a new sequence of terms, but the new order does not have to be the same as the old. Furthermore, it is free to generate a term t_j, $j > i$, that was not in the original sequence, before returning all values from the original sequence.

6.3.3 The Backward Execution Algorithm

The backward execution algorithm is presented in Figures 6.9 and 6.10.

When a fail message is received from the OR process for a literal L, the OR process for one of the generators must bind its variables to different values. The first part of backward execution consists of identifying this generator, which will be called *BL*. The second part consists of determining how to update the variables generated by *BL*, and the third involves resetting other generators.

Selection of the generator to redo is based on marks on the literals in the candidate set for L. A mark X on a generator G means G may be directly or indirectly responsible for the failure of X. The first thing to do is add L to the set of marks on each predecessor of L. The selected generator is the generator latest in the linear ordering marked with L or a successor of L. The reason we check for successors of L can be explained by an example based on the graph of Figure 6.5 on page 92. Suppose the process solving

The Backward Execution Algorithm

Solved, Pending, Blocked: State variables of the AND process, representing the status of the literals.

marks(i): The set of marks on literal *i*.

Procedure *back-up(FL):*

FL: The literal corresponding to the failed process.

1. Add *FL* to the marks of each literal in *pred(FL)*.
2. Let *BL* be the latest literal in the linear ordering with a set of marks containing a literal in $\{FL\} \cup succ(FL)$. If there is no such set of marks, *BL* is `HG`.
3. If *BL* is `HG`, the AND process fails, otherwise continue.
4. Call *next-result(BL)*; if it fails, make the recursive call *back-up(BL)*, otherwise continue. (*next-result* is defined in Figure 6.10)
5. Initialize a set *MV* to be the set of variables generated by *BL*, and let $marks(BL) = \emptyset$.
6. Work toward the end of the literal ordering, starting from *BL*, and do the following to each literal *L*:
   (a) If *L* consumes a variable in *MV*, cancel the OR process for *L*, set *marks(L)* to $\emptyset$, and move *L* to the set of blocked literals. If *L* is a generator, add the variables generated by *L* to *MV*.
   (b) If *L* is a generator in *candidates(BL)* and does not consume a variable in *MV*, set *marks(L)* to $\emptyset$ and call *reset(L)* (Figure 6.10). If the values of the variables generated by *L* change as a result of the reset, add them to *MV*.
7. For each literal $\{L | L \in Blocked, pred(L) \subseteq Solved\}$ start an OR process and move *L* from *Blocked* to *Pending*.

After receiving a fail message from process for *L*: Call *back-up(L)*.

After receiving a redo message: Call *back-up(*`HC`*)*.

Figure 6.9: Backward Execution Algorithm

`next(D,E)` fails, we attempt to obtain another result from the generator of `E`, and it also fails. At this point, if we simply check for literals marked with the ID of `next(B,E)`, the failed generator, we would backtrack to the top node in the graph. However, this would ignore the fact that `next(D,E)` had failed earlier. We still need to obtain further results from other predecessors of this node, since `next(D,E)` might succeed given new values for `D`. We have to backtrack to the generator of `D` and reset the generator of `E`.

There are cases when it is correct to backtrack to the root of the graph in the above situation. If `next(B,E)` fails before ever generating a value for `E`, it means the value of `B` is not acceptable. In this case, `next(D,E)` would not be started yet (since `next(B,E)` was not solved), so its failure could not have marked `next(A,D)`, and the backward execution algorithm will select the root of the graph. By checking for the failed literal or a successor of the failed literal in the set of marks, we correctly distinguish the cases where a generator fails because it has no further bindings for the variables it generates as opposed to failing because it cannot be solved with the combination of inputs it was given.

After identifying the generator that should change the values of the variables it binds, procedure *next-result* of Figure 6.10 is called. A straightforward implementation of backward execution would obtain the next result from the selected literal by sending a redo message to the corresponding OR process. For reasons to be explained in the discussion at the end of the chapter, parallel AND processes maintain a cache of results from each generator. What the next result procedure does is check the cache to see if there are any usable values on hand, and if not, send a redo message.

Next, the AND process must decide which generators must be reset after selection of *BL* as the backtrack literal. This potentially means every generator following *BL* in the linear ordering. However, not all generators need to be reset. We have to reset only those that contribute information, along with *BL*, to the solution of any successors of *BL* – in other words, the literals with *BL* in their candidate set. Since X is in the candidate set of Y if Y is in the candidate set of X, a literal is reset if it is in the candidate set of *BL* and it follows *BL* in the linear ordering.

The cleanup phase of a backward execution step is to scan the literals after *BL* in the linear ordering, resetting requisite generators and canceling the OR process for each literal that consumes variables modified during this procedure. After this pass over the literals, a forward execution step is taken, since a canceled literal may be enabled again if all its predecessors have up to date bindings through a reset.

The backward execution algorithm is invoked by a redo message from the parent of the AND process as well as a fail from one of its descendants. No special treatment is required for this event. The failed literal is the head of the clause, which means we have to back up from `HC`, the node representing

Result Cache Algorithms

Old For each generator, the list of results sent from the OR process and used to set the value of the corresponding variables.

New For each generator, the list of future bindings for its variables. Generally these are "recycled" values, not values just arrived from the OR process and waiting to be applied.

Procedure *next-result(L):* Return *OK* if the variables generated by L can be given new bindings, or if the process for L can potentially send new bindings.

1. If *New(L)* $\neq \emptyset$, add the current bindings to *Old(L)*, and remove a set of bindings from *New(L)* and make them the current bindings. Return *OK*.
2. If $L \in$ *Pending*, add the current bindings to *Old(L)*, remove L from *Solved*, and return *OK*.
3. If the process for L has failed, return *FAIL*.
4. Add the current bindings to *Old(L)*, send the process for L a redo message, move L from *Solved* to *Pending*, and return *OK*.

Procedure *reset(L):* Return *TRUE* if the variables generated by L change values as a result of the reset.

1. If $L \in$ *Blocked*, do nothing, return *FALSE*.
2. If *Old(L)* $= \emptyset$, do nothing, return *FALSE*.
3. Move all bindings to *New(L)*, setting *Old(L)* to the empty list. Remove a set of bindings from *New(L)* and make them the current bindings. Add L to *Solved* and return *TRUE*.

Figure 6.10: Maintaining a Cache of Results

the head as a consumer in the dataflow graph. The immediate predecessors of `HC` are marked, one of the generators is selected, and the cleanup operation is performed.

6.4 Detailed Example

In the example execution plotted in Figure 4.1 on page 69, process 4 was created to solve the body of the clause

```
paper(P,1978,uci) ←
        date(P,1978) ∧ author(P,A) ∧ loc(A,uci,1978).
```

The steps in the interpretation of this process, from the original ordering of literals through the generation of all solutions, will be explained in detail in this section. This example is intended to illustrate the details of all three components – literal ordering, forward execution, backward execution – in a parallel AND process for a fairly simple clause. More complex situations that may arise in a parallel AND process will be discussed in the following section. Another detailed example of a parallel AND process is in Chapter 7, where the solution of the map coloring problem is described.

6.4.1 Ordering

The clause used to define the initial state of process 4 is

```
paper(P,D,I) ←
        date(P,D) ∧ author(P,A) ∧ loc(A,I,D).
```

There are no mode declarations, so no literals are designated as generators or non-generators in the static analysis. All variables are placed in U.

When the process is created, variables `D` and `I` are bound to `1978` and `uci`, respectively. `HG` is designated the generator of these variables, G is $\{\texttt{D}, \texttt{I}\}$, and U becomes $\{\texttt{P}, \texttt{A}\}$.

The connection rule is applied to try to "connect" a set of literals to the head, by looking for literals containing variables in both G and U. The first pass through the list of literals finds two that meet this criterion. `date(P,D)` contains `P`, a variable with no generator yet, and `D`, a variable generated by the head, so it is designated as the generator of `P`. Similarly, `loc(A,I,D)` becomes the generator of `A`. U is now empty, so the ordering algorithm terminates.

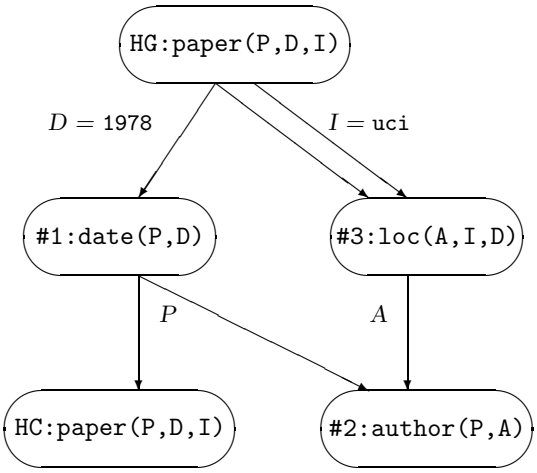
The dataflow graph and the relations defined by it are shown in Figure 6.11.

Clause:

paper(P,D,I) ← date(P,D) ∧ author(P,A) ∧ loc(A,I,D).

Call:

← paper(X,1978,uci).



Linear Ordering: [#1,#3,#2]

Candidates[1]: {HG, 3}
Candidates[2]: {1, 3}
Candidates[3]: {HG, 1}
Candidates[HC]: {1}

Figure 6.11: Graph for Detailed Example

```
<1> (after 'start' from Process 2, T = 2):
   Goal:     [paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   Solved:   [HG]
   Pending:  [#1:6,#3:7]
   Blocked:  [#2]
   Frame:    [P:_,D:1978,I:uci,A:_]
   Marks:    [HG:[],#1:[],#3:[]]

<2> (after 'success(date(pro,1978))' from Process 6, T = 4):
   Goal:     [paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   Solved:   [HG,#1:6]
   Pending:  [#3:7]
   Blocked:  [#2]
   Frame:    [P:pro,D:1978,I:uci,A:_]
   Marks:    [HG:[],#1:[],#3:[]]

<3> (after 'success(loc(kling,uci,1978))' from Process 7, T = 5):
   Goal:     [paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   Solved:   [HG,#3:7,#1:6]
   Pending:  [#2:9]
   Blocked:  []
   Frame:    [P:pro,D:1978,I:uci,A:kling]
   Marks:    [HG:[],#1:[],#3:[]]
```

Figure 6.12: States of a Parallel AND Process

6.4.2 Forward Execution

The individual state transitions are summarized in Figure 6.12. Items in the blocked, pending, and solved lists are terms of the form `#N:P`, where `#N` identifies a literal, and `P` is the ID of the OR process currently solving literal `#N`. When the process for a literal fails, the term representing the literal becomes `#N:_` until a new process replaces the old one.

The first forward execution step is shown in transition `<1>`. Literals `#1` and `#3` are both enabled – the predecessor set for each is a subset of {`HG`} – so we immediately start OR processes for `#1` and `#3`. Those literals are moved from the blocked list to the pending list. The state of the AND process after transition `<1>` shows process 6 solving literal `#1`, and process 7 solving `#3`.

Transition `<2>` occurs when `success(date(prolog,1978))` arrives from process 6. This is from the process for literal `#1`, so `#1` is added to the solved list. Since `#3` is also a predecessor of `#2`, and `#3` is not yet solved, no new processes are created on this step.

The next transition occurs when `success(loc(kling,uci,1978))` ar-

```
<4> (after 'fail' from Process 9, T = 7):
   Goal:     [paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   Solved:   [HG,#1:6]
   Pending:  [#3:7]
   Blocked:  [#2]
   Frame:    [P:pro,D:1978,I:uci,A:_]
   Marks:    [HG:[2],#1:[2],#3:[]]

<5> (after 'fail' from Process 7, T = 9):
   Goal:     [paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   Solved:   [HG,#3:_]
   Pending:  [#1:6]
   Blocked:  [#2]
   Frame:    [P:_,D:1978,I:uci,A:kling]
   Marks:    [HG:[2,3],#1:[],#3:[]]

<6> (after 'success(date(eft,1978))' from Process 6, T = 11):
   Goal:     [paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   Solved:   [HG,#1:6,#3:_]
   Pending:  [#2:11]
   Blocked:  []
   Frame:    [P:eft,D:1978,I:uci,A:kling]
   Marks:    [HG:[2,3],#1:[],#3:[]]
```

Figure 6.12 (cont'd)

rives from process 7, currently solving `#3`. `#3` is added to the solved list, and now a process (number 9) is created for `#2`. Note that after bindings from the first two answers have been applied, `#2` is `author(prolog,kling)`, which will fail, triggering backward execution.

6.4.3 Backward Execution

Backward execution often requires cancel messages to be sent to descendant OR processes. After a parent sends a cancel message, it can ignore any subsequent messages received from the descendant. This situation may arise when a descendant sends a message, but the message has not yet been processed by the time the parent decides to send the cancel message. In the discussion below, *replacing* a process P means sending P a cancel message, creating a new process P' for the same literal (but using new bindings), and using the process ID of P' in place of P.

In the linear ordering of this clause, the generator of `A` comes after the generator of `P`, so `A` corresponds to the "innermost" variable. Each time `#2`

fails, we will first try to obtain another value from #3, the generator of A. When that fails, we ask for a new value of P and reset A.

Transition <4> is the first backward execution step. When the AND process receives the fail message from the process for #2, it adds #2 to the marks on all predecessors of #2. The generator latest in the linear ordering marked with #2 is #3, so #3 is the literal selected to generate new bindings. The *next-result* procedure is called, and since the cache of unused bindings for #3 is empty, a redo message is sent to the process for #3. #3 is moved from solved to pending since we are waiting to see if it can provide another value for A. The marks are removed from #3 but left on #1.

Transition <5> is triggered by a fail from the process for #3, meaning there is no additional binding for A that satisfies loc(A,uci,1978). HG, the immediate predecessor of #3, is marked. We search through the generators, starting from the end of the linear ordering, looking for #3 or #2 in a set of marks. #1 qualifies since it is marked with 2. #1 is moved from solved to pending, and the process for #1 is sent a redo message. The marks are removed from #1. The literals following #1 in the literal ordering are #3 and #2; since #3 is in *candidates(1)* it is reset, and #2 is canceled (this has no effect since #2 was blocked). Since #3 was solved once, we can use the first value it sent as the current value of A, and we can move #3 to the solved list. Notice the term representing #3 in the solved list: since the process for the literal failed, we cannot send it another redo message.

Transition <6> takes place when a success message arrives from the process for #1 with the second binding for P. #1 is added to the list of solved literals, and a new process can be created for author(eft,kling), the current instantiation of #2. The states of the literals are now: #1 and #3 solved, #2 pending.

When the process for #2 sends success (transition <7>), all literals have been solved. A success message containing a copy of the goal statement with the bindings {P/eft, D/1978, I/uci, A/kling} is sent to the parent OR process.

6.4.4 Additional Solutions

The next message processed by the AND process is a redo message from its parent. The "failed literal" in this case is HC, the head in its role as consumer. Only literal #1 is marked, since the head of the clause is not concerned with further values of #A, as indicated by the relationships in the dataflow graph. #1 is selected as the backtrack literal, and its marks removed. It is moved from solved to pending. #3 is reset, and again we consider it solved. #2 is canceled: the process for #2 is sent a cancel message (since any further processing of literal #2 will be based on new values, if any, from #1 and #3), and #2 is moved to the blocked list. The new state is: #1 pending, #3 solved,

```
<7> (after 'success(author(eft,kling))' from Process 11, T = 13):
   Goal:     [paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   Solved:   [HG,#2:11,#1:6,#3:_]
   Pending:  []
   Blocked:  []
   Frame:    [P:eft,D:1978,I:uci,A:kling]
   Marks:    [HG:[2,3],#1:[],#3:[]]

<8> (after 'redo' from Process 2, T = 15):
   Goal:     [paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   Solved:   [HG,#3:_]
   Pending:  [#1:6]
   Blocked:  [#2]
   Frame:    [P:_,D:1978,I:uci,A:kling]
   Marks:    [HG:[2,3,HC],#1:[],#3:[]]

<9> (after 'success(date(df,1978))' from Process 6, T = 17):
   Goal:     [paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   Solved:   [HG,#1:6,#3:_]
   Pending:  [#2:13]
   Blocked:  []
   Frame:    [P:df,D:1978,I:uci,A:kling]
   Marks:    [HG:[2,3,HC],#1:[],#3:[]]
```

Figure 6.12 (cont'd)

#2 blocked. Note that we do not care if #2 can be solved more than once with the previous bindings; this will be discussed further in Section 6.5.5.

Transitions <9> through <12> show how the AND process tries unsuccessfully to get another value for P. In transition <9> a success message arrives from the process for #1 with new value for P, and the new value is used to start a new process for #2. In transition <10> a fail message arrives from this process, and #3 is chosen as the backtrack literal. As in transition <4>, the AND process checks for unused answers from the process for #3, and finds none. This time, however, there is no longer a process for #3 (it sent a fail message back in transition <5>), so we cannot send it a redo. Since there are no unused results from the process for #3, and no way of obtaining further answers, we treat this as a failure of #3. The immediate predecessors of #3 are marked, and we look for a generator marked with either #3 or #2. This results in the selection of #1 as the backtrack literal. The new state of the AND process thus has #1 pending, #2 blocked, and #3 solved (due to the

```
<10> (after 'fail' from Process 13, T = 19):
   Goal:     [paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   Solved:   [HG,#3:_]
   Pending:  [#1:6]
   Blocked:  [#2]
   Frame:    [P:_,D:1978,I:uci,A:kling]
   Marks:    [HG:[2,3,HC],#1:[2],#3:[]]

<11> (after 'success(date(fp,1978))' from Process 6, T = 21):
   Goal:     [paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   Solved:   [HG,#1:6,#3:_]
   Pending:  [#2:15]
   Blocked:  []
   Frame:    [P:fp,D:1978,I:uci,A:kling]
   Marks:    [HG:[2,3,HC],#1:[],#3:[]]

<12> (after 'fail' from Process 15, T = 23):
   Goal:     [paper(P,1978,uci),[date(P,1978),author(P,A),loc(A,uci)]]
   Solved:   [HG,#3:_]
   Pending:  [#1:6]
   Blocked:  [#2]
   Frame:    [P:_,D:1978,I:uci,A:kling]
   Marks:    [HG:[2,3,HC],#1:[],#3:[]]

<13> (after 'fail' from Process 6, T = 25):
   <done>
```

Figure 6.12 (cont'd)

reset of #3).

In transition <11> we get a new value for P and start a new process for #2. This process will also fail, so transitions <11> and <12> are essentially the same as <9> and <10>. After transition <12> we are waiting for a fifth value of P from the process for #1.

Finally, in transition <13>, a fail message arrives from the process for #1, indicating there are no more ways to solve date(P,1978). The head of the clause is the only predecessor of #1; #1 is added to the set of marks on HG. HG is the only possible backtrack literal – it is the only literal with a mark of #1 or #2 – so the AND process fails.

6.5 Discussion

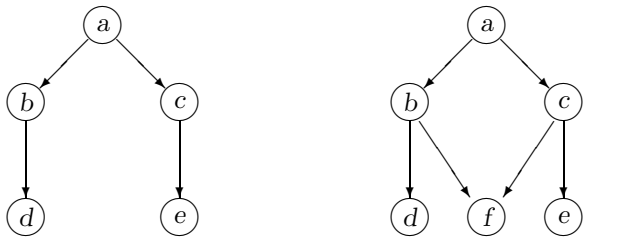
The backward execution algorithm described in this chapter is the result of a series of improvements in the technique for generating multiple results by an AND process. Each improvement leads to fewer tuples generated for consideration by consumer processes or fewer steps taken to generate the tuples.

A sequential AND process uses sequential backtracking, with all the faults of Prolog and other sequential systems. The basic parallel model uses a graph directed backtracking algorithm, where the backtrack literal is determined by the set of marks on generators. This leads to a form of intelligent backtracking since the backtrack literal is directly or indirectly responsible for the failure of the failed literal. If a literal fails because the bindings for its input variables are invalid, the backtrack literal is a predecessor of the failed literal. If a literal fails after generating all possible values for the variables it binds, the graph is used to select a different generator. In the basic model, as in a nested loop model, every generator following the backtrack literal in the linear ordering is reset, and every literal that consumes variables modified by a reset is canceled.

The algorithm presented in this chapter improves on the basic model through the use of candidate sets and a result cache to save steps. Using the candidate sets to determine which generators need to be reset leads to fewer reset literals and in turn to fewer descendant OR processes. Using a cache not only saves the results generated by descendants, it also leads to fewer resets. These improvements and other aspects of parallel AND processes are discussed in the following sections.

6.5.1 Relative Order of Incoming Messages

The order of arrival of success and fail messages can have an effect on future state transitions. In general, if a fail message and a success message are both waiting to be processed, fewer steps will be executed if the fail message is handled first. This is because a transition based on a fail message often cancels descendant processes, and one of the canceled processes might be the one that sent the success. If the fail is processed first, and the process sending the success is canceled, the AND process can ignore the success message. However, if the success message is handled first, a complete step is executed before the transition specified by the fail is taken. The rules for state transitions are such that the relative order of arrival does not matter; either way, the system will eventually be in the same state after processing both messages. However, the computation will generally be more efficient if fail messages are processed first when the process has a choice.



Candidates(c) $= \{a\}$ *Candidates(c)* $= \{a, b\}$

(a) **(b)**

Figure 6.13: Candidate Sets

6.5.2 Definition of Candidate Set

Figures 6.13a and 6.13b illustrate the reasoning behind the definition of the set of candidate literals for backward execution. A literal is a candidate for i if it is an immediate predecessor of i or a predecessor of a successor of i:

$$candidates(i) = ipred(i) \cup cp(i) - \{i\}$$

where

$$cp(i) = \bigcup_{x \in succ(i)} pred(x)$$

A non-generator literal has an empty set of successors, so the candidate set for one of these literals is the set of its immediate predecessors.

One use of this set is in identifying the literal to back up to after i fails. In Figure 6.13a, if c fails it is clear we want to back up to the root of the tree. However, in Figure 6.13b, when c fails we might have to back up to literal b, because f may have failed earlier, and by backing up to b and resetting c we are moving to the next combination of values for f to consume.

We do not always want to back up to b after c fails, however. If c fails without producing any results, we should back up to the root, because this means c consumes an unsatisfactory value. The algorithm presented here

correctly distinguishes these two cases by using the set of marks on the literals. When c fails, we check the marks on the candidates of c. If c had never succeeded, there could be no marks from f on b, so a is selected for backtracking.

The other use of the candidate set is in determining the generators to reset after a backtrack step. Referring again to Figure 6.13a, it would make no sense to reset c when b is selected for backtracking after d fails. In Figure 6.13b, however, c is reset in this situation. The reason is that when b and c have a descendant in common, the descendant needs to see all combinations of values from b and c, and this is achieved by resetting c whenever b gets a new value.

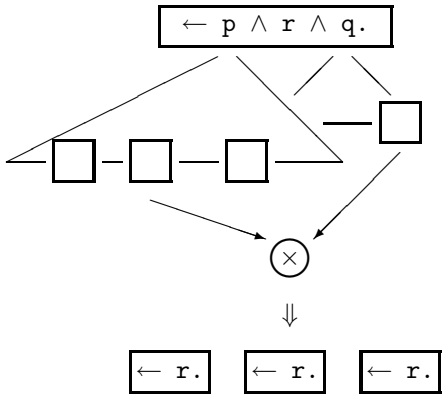
Another feature of using the candidate set for selecting a backtrack literal is the processing of multiple independent failures. In Figure 6.13a, suppose d and e both fail when given initial values by their respective generators. Redo messages will be sent to the OR processes for b and c. In this example, the AND process directs backtracking independently in separate parts of the graph. If either branch ever fails back to the common ancestor node, the other is terminated, since the root will be generating a new value for both its consumers.

The net effect of these rules for backward execution is a better choice of backtracking points. The techniques outlined here are useful in a sequential system, as well. A graph for the body of a clause can be created at compile time, and the graph used later to direct backtracking in the sequential system [10]. The set of backtracking points is not as efficient as the set generated during intelligent backtracking [6, 73]. When a literal `p(X,Y)` fails, we back up to the generator for `X` or `Y`, whichever occurs later in the linear ordering, without knowing which binding actually caused the failure. Chang *et al* coined the term *semi-intelligent* to describe this form of backtracking.

6.5.3 Result Cache

In the current implementation, AND processes maintain a cache of results from each generator. The cache could be kept by the OR processes, in the form of the "sent list" described in the previous chapter, since information about the number of results generated is useful status information for the OR process as well.

The existence of the cache complicates the message protocol between AND and OR processes. A situation may arise where a pending literal is reset by the AND process. If there are previous bindings for the variables generated by this literal, the bindings will be used and the literal added to the set of solved literals. The literal is now in two sets, *Pending* and *Solved*; it must remain in the pending set because we should handle the response when it eventually arrives. The way to handle this is to have the message



p and q are generators which can be solved in parallel. The backward execution algorithm arranges the combination of values to make instances of r. The values generated by q are computed once and stored in the cache, as opposed to being recomputed each time p generates a new value.

Figure 6.14: The Effect of Goal Caching on Recomputation

interface of the AND process intercept messages from processes for literals currently in *Solved*. If the message is a success, the bindings in the message are simply added to the *New* list of the cache for the literal. If the message is fail, the literal is marked as failed, and the *next-result* procedure will later take this into account.

The primary purpose of the cache is to save bindings created by independent OR processes in order to avoid recomputing the same values over and over. The goal tree of Figure 3.9, which illustrated the effects of combining AND and OR parallelism, is redrawn here in Figure 6.14.

Assume the goal at the root of the tree is based on the clause

```
f(X,Y) ← p(X) ∧ q(Y) ∧ r(X,Y).
```

where `p` is the generator of `X` and `q` is the generator of `Y`. `p` and `q` are independent and can be solved simultaneously. If there are n_p solutions to `p` and n_q solutions to `q`, the backward execution mechanism of the AND process will create the product of bindings of `X` and `Y` and start (one at a time) $n_p \times n_q$ different OR processes for `r`. If the results from `q` are saved in a cache, `q` can be solved once and the n_q results stored in the cache. In Prolog and pure OR parallel systems, `q(Y)` has to be computed n_p different times, once for each solution of `p`. An example of this was seen in the detailed example, where the literal `loc(A,I,D)` was solved once and the same binding for `A` used four times.

The parallel AND process still gives rise to duplicate computations, however. Consider a partial goal statement

$$\leftarrow \ldots \texttt{f(X)} \wedge \texttt{g(Y)} \wedge \texttt{p(X,Y)} \ldots$$

Suppose `p(X,Y)` succeeds using one particular combination of `X` and `Y` values, but fails for others. If `f` and `g` are reset, the AND process will cancel `p` and start a new process with the reset values of `X` and `Y`, not taking into account whether this combination of `X` and `Y` was previously successful or not. A future optimization might be able to use the cache to avoid this source of recomputation. In the basic model, generators are not required to bind variables in the same order after a reset. When a generator makes a series of bindings $t_1 \ldots t_i$ up to the time it is reset, it is not required to use t_1, or any other value in the sequence, as the first binding after the reset. It makes sense to reset `f` and `g` to the first acceptable combination of values in the cache, rather than an arbitrary combination that may not satisfy `p`. The information about the acceptable combination is lost in the reset in the general model. However, through the use of a result cache we might be able to order the bindings, defining the notion of "first acceptable combination" and remembering which combinations were valid. Incorporating the ordering into the communication protocol and backward execution algorithm, and measuring its effectiveness are topics for future research.

6.5.4 Infinite Domains

A requirement for backtracking is to be able to generate as many tuples of values as possible. When the domains of the variables are finite, sequential backtracking and nested loop models satisfy the requirement, but when one or more domains are infinite, these models are not able to generate all tuples. For example, consider two predicates, `gi` and `gf`. The denotation of `gi` is the infinite sequence $\{\texttt{i}_1, \texttt{i}_2, \ldots\}$, and the denotation of `gf` is the finite sequence $\{\texttt{f}_1, \texttt{f}_2, \ldots \texttt{f}_m\}$. In the goal list

$$\leftarrow \texttt{gf(F)} \wedge \texttt{gi(I)} \wedge \texttt{pf(F)} \wedge \texttt{pi(I).}$$

`gf` is the generator of `F`, `gi` the generator of `I`, and `I` is the innermost variable. Suppose `pf` succeeds only for the second value of `F`, but `pi` succeeds for any value of `I`. Since no tuple with $\texttt{f}_1$ can succeed (*i.e.* $\texttt{pf(f}_1\texttt{)}$ fails), Prolog and the equivalent nested loop program in a procedural language will never solve this goal. After `pf` fails, the interpreter backtracks to `gi`, and it generates another of the infinite number of values for `I`. All tuples created will be of the form $\texttt{<f}_1, \texttt{i}_j\texttt{>}$, with the interpreter stuck in an infinite loop generating the $\texttt{i}_j$. Parallel AND processes have a chance of succeeding in this example, since when `pf(F)` fails a redo message is sent to the generator for `F`, while the generator for `I` is reset. Thus there are cases, even when infinite domains

are involved, where parallel AND processes construct all successful tuples. Parallel AND processes are still not perfect, however. If the consumer is `p(F,I)`, the linear ordering may specify that the generator for `I` is to be sent a redo message before the generator of `F`, and thus the parallel AND process is also caught in an infinite loop.

The intelligent backtracking interpreter of Pereira and Porto can also avoid infinite loops, since the generators of infinite domains may be skipped on backtracking. Their interpreter may even succeed when parallel AND processes fail, because their interpreter analyzes the *cause* of a failure. If $\mathtt{p}(\mathtt{f}_1, \mathtt{i}_1)$ fails because the unification of $\mathtt{p}(\mathtt{f}_1, \mathtt{i}_1)$ does not succeed when `F` is bound to $\mathtt{f}_1$, their interpreter knows to backtrack into the solution of the generator of `F`. At present, all a parallel AND process knows is that $\mathtt{p}(\mathtt{f}_1, \mathtt{i}_1)$ failed, this literal has two predecessors, and that one of them must be redone; which one is determined solely by the linear ordering. Incorporation of information about the cause of a failure in a fail message, and use of such information, is a subject of future research.

6.5.5 Multisets of Results

Up to this point we have been referring to $\mathbf{D}_1(\mathtt{p})$, the denotation of a procedure `p`, as as a *set* of answers. The set of results actually constructed by interpreters should be referred to as a *multiset*, or *bag*, since some of the tuples can occur more than once. It is interesting to compare the AND/OR Process Model and Prolog with respect to the creation of duplicate tuples. In Prolog, duplicates may occur as a side effect of the control strategy, in situations where the logic of the program does not specify multiple occurrences of the same tuple. If a multiset is constructed in the AND/OR Process Model, however, it is based only on the logical structure of the program. The following program and goal statement illustrate the difference:

```
← p(A).
p(A) ← q(A) ∧ r(B).
q(0).
r(1).
r(2).
```

If the goal is proved with a depth-first control strategy, as in Prolog, the binding `A/0` is generated twice. The first answer is found by unifying `q(A)` with the unit clause `q(0)`, and then unifying `r(B)` with `r(1)`. If told to backtrack to create a second answer, Prolog will re-solve `r(B)`, this time by binding `B` to 2, and it once again reports success for `p(0)`.

If the clause for `p` is solved with an AND process using the backward execution algorithm presented in this chapter, `p(0)` is proved only once. When a redo is sent to the AND process, `q(A)` is chosen as the backtrack

literal, since the unbound variable in the head is `A` and `q(A)` is the generator of `A`. `q(A)` fails to give a second binding to `A`, so the AND process fails after providing just one result.

Next consider a very similar program, where one of the assertions for `r` has been replaced by a duplicate assertion for `q`:

```
← p(A).
p(A) ← q(A) ∧ r(B).
q(0).
q(0).
r(2).
```

Now both interpreters report two solutions, each with `A` bound to `0`. The difference between the two programs is that in the second, the structure of the program directs an interpreter to produce a multiset, but in the first the creation of the multiset is a side effect of the control strategy. In the first program, Prolog is producing a multiset of `A` values by binding `B` to different terms, not by binding `A` to different instances of the same term. In the AND/OR Process Model, the number of bindings for a variable is determined by the number of ways the value can be derived. It does not depend on the context of the proof; when a goal is solved in the context of another, independent goal, the number of solutions of the additional goal does not influence the number of solutions reported for the first.

Multisets are useful in database applications. For example, suppose Sally had a score of `95` on a test. This could be represented in a database as `score(sally,midterm,95)`. If we want a list of all midterm scores, in order to compute the mean, we can use the following query, using an "all solutions" predicate similar to Prolog's `bagof`:

```
← bag(N,S^score(S,midterm,N),B) ∧ mean(B,M).
```

This goal will succeed by binding `L` to the list of all values of `N` that satisfy the specified goal.[1] When more than one student scores `95`, we certainly want the list to contain the number `95` once for each student, or the calculation of the mean will be incorrect.

[1] `bag` is a special OR process, similar to Prolog's `bagof`, collecting all answers from its descendants and putting them in a list for its parent. The caret operator tells the interpreter to disregard bindings for `S`.

6.6 Chapter Summary

Parallel solution of the body of a clause is based on a dataflow graph for the literals in the body. A literal ordering algorithm automatically generates the graph, without requiring annotations specifying the relative order of solutions. If a literal fails when mode constraints are violated, and the modes are unavailable, the algorithm may generate an illegal graph, but otherwise the graph describes a valid parallel solution. Heuristics can be used to influence the algorithm, leading to more or less parallelism.

When literals all succeed, which is often the case when the clause implements a deterministic function, the solution of the body is straightforward. AND parallelism in the bodies of these clauses corresponds to the parallelism found in an equivalent program written in a functional language.

In nondeterministic functions and relations, it is not always the case that literals can be solved on the first attempt. When a literal fails, an interpreter must re-solve a previously solved literal, in order to create new bindings for the input variables of the failed literal. The backward execution mechanism in parallel AND processes determines which literals must be re-solved in response to failures. Although the backward execution algorithm is complicated, the processing steps are relatively simple and do not involve complex control decisions. The next chapter has an overview of an implementation technique for backward execution where steps are based on simple bitset operations, and other implementation techniques that will lead to an efficient AND/OR process interpreter on a multiprocessor.

Chapter 7

Implementation

The overall goal of this research is the design of a multiprocessor computer architecture for parallel execution of logic programs. The research takes the language first approach, summarized in the introduction, in which the designer starts with an abstract model of computation, defines a method for interpreting programs of the model in parallel, and finally starts the design of a computer based on the parallel execution model. Previous chapters described the research in the early steps of this top down process, research that culminated in the definition of a method for interpreting logic programs that automatically divides the program into independent pieces for parallel solution. The implementation techniques presented in this chapter represent the beginnings of the design of the lower levels.

The techniques are a summary of ideas from four different implementations. Two of the systems – the Prolog version [19] and one written in C – are complete interpreters. The other two are partial implementations, intended to test different algorithms in isolation. An interpreter written in Modula-2 was used to test the bitset representation and corresponding algorithms for backward execution described later in the chapter.

At this next lower level, a logic program appears to the system as a collection of independent processes communicating via messages. When a process receives a message, it will be transformed into another state, and possibly generate messages for other processes. Each state transformation is an atomic operation; if a message arrives at a process while a transformation is in progress, it is queued with other incoming messages until the process is ready to accept it. The abstract model has been designed so that the order of acceptance of incoming messages does not matter. We will use operating system terminology when describing the execution of a process. A process is *running* when it is being transformed from one state to another, it is *blocked* when it is waiting for a message, and it is *ready* when it has an incoming

message but no processor in the system is (yet) transforming it into its next state.

The machine at this level is a network of homogeneous *processing elements*, or *PEs*. We will assume each PE has a large local memory for storing the static program code and a subset of the processes and messages. We will continue to specify algorithms and representations without relying on a common memory or global address space. Common memory may eventually be the best way to implement the binding environments or process allocation strategies discussed in this chapter; at this point, however, we will continue the policy outlined in the first chapter and design for systems without common memory. For examples of some of the implementation techniques possible in a shared-memory multiprocessor, see the papers by Borgwardt [3, 4].

7.1 Overview of the Interpreter

An AND process is created to solve a goal statement, either the user's top-level goal or the body of a clause. The process maintains a single binding environment, known as the current environment, to represent the values assigned to each variable of the goal statement. The current environment was called the "frame" in the states shown in Figure 6.12.

OR processes solve one of the goals from their parent's goal statement. When an OR process is started, it uses a unique copy of the parent's current environment in each unification with a candidate clause. Thus each unification is free to bind variables in the parent environment without worrying about conflicting bindings. When the OR process sends a success message back to its parent, the argument is one of these copies; the AND process then uses the copy to update the current environment with the values created by the descendant OR process. If the AND process is a sequential process, the environment sent by the OR process becomes the new current environment; no merging of values is necessary.

Each unification with a candidate clause involves two environments: the copy of the parent environment described in the previous paragraph, and a new frame. The new frame contains one empty slot for each variable of the candidate clause. If the unification is successful, and the clause has a body, the new frame becomes the initial frame of an AND process for the body.

Even when an OR process has its own copy of its parent's frame, situations arise where the unification that starts a descendant would bind a shared variable in an ancestor frame. One of the techniques discussed in Chapter 3 for handling this situation must be used to give each OR process its own copy of shared variables. The method currently being used resembles Lindstrom's variable importation scheme [58], except it has been designed to work in a distributed memory environment [20].

7.2 Parallel AND Processes

The state of a parallel AND process consists of static information such as the ID of the parent OR process and dynamic information such as the status of each literal in the goal statement. Techniques for efficiently representing the dynamic information will be presented in this section. The techniques were developed as part of an interpreter written in Modula-2, intended to show that most of the dynamic information used for backward execution could be represented in bitsets. A bitset is a set of atomic items, such as integers. If the universe of all possible elements is small, the set can be represented in a single memory word, and elementary operations such as union and intersection performed with one machine level instruction.

As in other recent implementations of AND processes based on ordering literals according to a dataflow graph, we will assume the graph is static. In the abstract model, the graph is dynamic, since new generators are designated when a generator returns an unbound variable or a term containing an unbound variable. We will assume the graph is constructed at compile time and does not change during execution [25].

Instead of storing a list called the linear ordering as part of the process, the literals can be labeled with an *index* according to their position in the ordering. `HG`, the node corresponding to the head as generator, has index 0, and `HC` is given an index higher than any body literal. For example, the main example in the last chapter was an AND process created to solve the body of the clause

$$\texttt{paper(P,D,I)} \leftarrow \texttt{date(P,D)} \wedge \texttt{author(P,A)} \wedge \texttt{loc(A,I,D).}$$

In the abstract model, the literals were numbered according to their occurrence in the body, *e.g.* `#3` was `loc(A,I,D)`. The linear ordering of the clause, determined by breadth-first traversal of the dataflow graph, was `[#1,#3,#2]`. In the compiled representation, `author(P,A)` has an index of 3, since it appears third in the linear ordering. This technique for labeling literals will simplify many of the steps of backward execution. After a backtrack literal is selected, an AND process must find all literals that follow the backtrack literal in the linear order. Using indices, this means any literal with an index higher than the index of the backtrack literal, so the cleanup phase of a backward execution step now involves a linear scan of an array of literal information starting from the index of the backtrack literal.

The dynamic state variables of the AND process consist entirely of bitsets or arrays of bitsets accessed by literal indices. The sets of solved, pending, and blocked literals are, naturally, bitsets. The dataflow graph is stored as an adjacency matrix, with the relevent information about each literal stored as a bitset. `pred[i]` is the set of predecessors of the literal with index `i`. Candidates and marks are also bitsets.

Bitset Operations:

Sets are represented by fixed-length words. The most significant bit in a set is numbered 0. If bit n is a 1, item n is a member of the set, otherwise it is not a member. The operation *last-element(S,i)* returns the highest index j of an element in S with index less than i:

$$(S(j) = 1) \wedge (j < i) \wedge (\neg\exists k : (k > j) \wedge (k < i) \wedge (S(k) = 1))$$

If bit i is the lowest index of any element in S, *last-element(S,i)* returns -1.

Procedure *select(FL)*:

FL: The index of the literal which failed.

*succ(i)***:** The set of successors of literal i in the dataflow graph.

1. Construct a set $MS = \{FL\} \cup succ(FL)$.
2. Let L be *last-element(candidates(FL),FL)*.
3. While $L \geq 0$:
   (a) If $marks(L) \cap MS \neq \emptyset$ then return L.
   (b) Set L to *last-element(candidates(FL),L)*.
4. Return 0 *(this return is taken when the head does not generate any values, and thus is never marked; return 0 when the root of such a graph fails).*

end procedure

This algorithm is used in the Modula-2 implementation of the AND/OR Process Model. It is called to select a literal for backtracking as part of backward execution.

Figure 7.1: Set Operations in Backward Execution

Some operations on bitsets, such as intersection and union, can be performed efficiently with most processor instruction sets, requiring just a single instruction. Some processors, such as those in the National Semiconductor 32000 series, have instructions such as "find first set bit" that will make other operations on bitsets very efficient. The use of bitsets in a backward execution step is shown in Figure 7.1. The *last-element* function could be implemented efficiently as a search from low order to high order bits for the first set bit (1 value) before a given index.

As an example of the use of indices for representing processes, and as a final example of a parallel AND process for a nondeterministic goal statement, the remainder of this section contains a detailed description of the solution of the map coloring problem first mentioned in Section 6.1.3. The description is based on traces of the solution produced by the Modula-2 implementation of parallel AND processes.

The dataflow graph for the clause is reproduced here as Figure 7.2. The index of a literal is shown next to the literal in the program, and the nodes in the graph are labeled with these indices. In the ensuing discussion, the notation `#N` will mean the literal with index `N`. Also, we introduce the notation `@N` to stand for the OR process created to solve `#N`.

The dataflow graph is created, at compile time, by applying the leftmost rule to assign `#1` as the generator of `A` and `B`, and then the connection rule designates the other three generators. In the first forward execution step, the only enabled literal is `#1`. This process succeeds, returning the message `success(next(green,yellow))`, binding `A` to `green` and `B` to `yellow`. This enables processes for the three generators in the middle row of the graph.

All three of the new processes will succeed, creating the following bindings: {`C/yellow`, `D/yellow`, `E/green`}. Of the consumers on the bottom row of the graph, only `#7` and `#8` can be solved with these values; `#5` and `#6` will fail. The AND process takes a different sequence of steps depending on which of the fail messages arrives first. The two traces to be described explain what happened in each case. Eventually the AND process reaches the same state, no matter which fail arrives first. Intuitively, one would expect the process to succeed in fewer steps if the fail from `@6` arrives first, since the generator of `C` is earliest in the linear ordering. Obtaining a value from a generator toward the front of the linear ordering corresponds to updating an outer variable in a nested loop, effectively skipping useless tuples created by updating inner variables. When the AND process reads the fail message from `@6` before the fail from `@5`, one less step is required in the AND process itself, and additional savings are realized by not sending messages to descendants in the step that was skipped.

Case 1: `#6` fails first. In this trace, processes `@1`, `@4`, and `@2` sent success messages, in that order, and then the fail from `@6` arrived. The state of the AND process at this point was: `#1`, `#2`, and `#4` solved; `#3`, `#6`, and `#7`

pending; #5 and #8 blocked. #6 was added to the marks on #1 and #2, and #2 was selected as the backtrack literal. A redo message was sent to @2, and the literals with indices greater than 2 were processed as follows:

#3: Reset; since there were no values in the cache for #3, and #3 was pending, the reset had no effect, and D was not added to the list of modified variables.

#4: Reset; also not affected because its cache is empty.

#5: Canceled, because it consumes the variable generated by the backtrack literal; note that because of this step, the message from @5 was ignored when it arrived.

#6: Canceled (it was the failed literal, so this has no effect).

#7: Canceled; it also consumes C.

#8: Not affected.

The new state had #1 and #4 solved, #2 and #3 pending, and #5 through #8 blocked.

Next a success from #3 arrived, binding D to yellow, and a process for #8 was started. A success from #2 arrived, carrying the second binding for C. Processes for the remaining three consumers were started. At this point, the frame contained the first correct tuple of values; all pending consumers returned success messages, and the AND process was able to send a success to its parent. In all, ten steps were required: one to handle the fail from @6, and nine to handle the success messages (one from each literal, plus the extra for the second binding from @2).

Case 2: #5 fails first. In this execution trace, all four generators succeeded before the fail from @5 was read. The state of the AND process at this point was: #1 through #4 solved, #5 through #8 pending, and none blocked. When the fail message arrived, #5 was added to the set of marks on #1, #2, and #3; #3 was selected as the backtrack literal, and @3 was sent a redo message. The processing of the literals following #3 was:

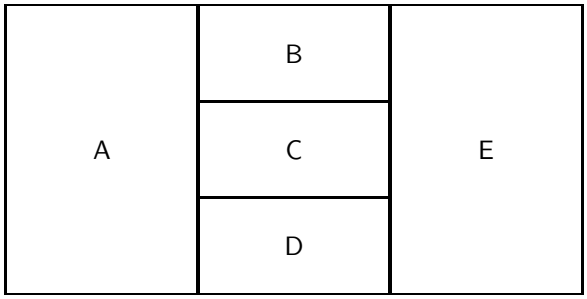
#4: Reset; cache was empty, so the reset had no effect.

#5: Canceled (it was the failed literal).

#6: Not affected.

#7: Not affected.

#8: Canceled; to be replaced when new value of D sent by @3.

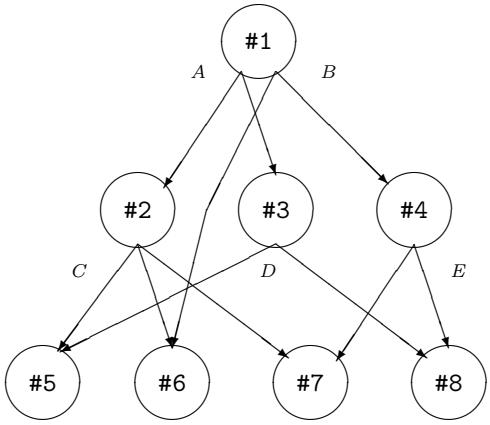


Call:

```
← color(A,B,C,D,E).
```

Clause:

```
color(A,B,C,D,E) ←
(1)  next(A,B) ∧
(5)  next(C,D) ∧
(2)  next(A,C) ∧
(3)  next(A,D) ∧
(6)  next(B,C) ∧
(4)  next(B,E) ∧
(7)  next(C,E) ∧
(8)  next(D,E).
```




```
next(green,yellow).
next(green,red).
next(green,blue).
next(yellow,green).
next(yellow,red).
next(yellow,blue).
next(red,green).
next(red,yellow).
next(red,blue).
next(blue,green).
next(blue,yellow).
next(blue,red).
```

Candidate Sets:

```
1:  {2,3,4}   2:  {1,3,4}
3:  {1,2,4}   4:  {1,2,3}
5:  {1,2,3}   6:  {1,2}
7:  {1,2,4}   8:  {1,3,4}
```

Figure 7.2: Map Coloring Program

The new state showed #1, #2, and #4 solved; #3, #6, and #7 pending; #5 and #8 blocked.

The next message was the fail from @6. #6 was added to the marks on #1 and #2, a redo message was sent to @2, and the literals after #2 were:

#3: Reset; this time the reset has an effect, since the first binding for D is in the cache. This binding is reinstated as the current value of D, and #3 is added to the set of solved literals.

#4: Reset; no old values in the cache, so the value of E is not changed.

#5: Canceled; since it was blocked, there is no need to send a cancel message.

#6: Canceled (it was the failed literal).

#7: Canceled; it consumes the variable generated by the failed literal.

#8: Canceled, since it consumes D, and D was modified by the reset of #3.

This trace shows an instance of the interaction of the cache and the message protocol. #3 was added to the solved set in this step, and @3 is working on the next solution of #3 so it is also still in the pending set. The state variables showed #1, #3, and #4 solved, #2, #3, and #8 pending (a new process was started for #8 after the old one was canceled), and #5, #6, and #7 blocked.

The next message was a success from #3 with the second binding for D. Since #3 was in the solved set, the new binding was added to the *New* list in the cache for #3 and no further steps were taken. Next, @2 sent the new binding for C, and processes were started for the three blocked literals. From this point on the remaining steps were the same, with all four pending literals about to send success messages. This trace showed one more step than the first trace, as a result of the processing of the extra fail message (this does not count a step for storing the extra, unused, success message from @3).

7.3 Process Allocation

Ideally, as soon as a process goes into the ready state, some PE will perform the prescribed state transition. This is unlikely if the PE storing the process has any other ready processes. When some PEs have a set of ready processes while others have only blocked processes, the system will not be executing as efficiently as possible. The *process allocation* policy is the mechanism that decides where in the network of PEs each process will be executed.

Process allocation schemes can be either static or dynamic. A static allocator maps processes onto processors at compile time. A dynamic allocator uses runtime information about the relative load on each machine to dynamically schedule a task for execution somewhere in the network. Desirable

attributes of the allocation scheme are decentralization, locality, and evenness. Decentralization means there should not be a central PE or authority that decides where a process will execute. A centralized mechanism is a bottleneck when there are a very larger number of processes, and is a vulnerable point in terms of system reliability. Locality means processes that communicate directly should be close to each other physically, so messages from one to the other will not have to travel very far in the network, no matter what the topology is. Finally, when goals are distributed evenly, every PE will have the same amount of work to do at all times.

A dynamic process distribution method proposed by Burton and Sleep for the ZAPP system [8] shows potential for use with the AND/OR Process Model. A similar idea was used in the Rediflow system by Keller, Lin, and Tanaka [52]. In this method there is no notion of *assigning* a newly formed process to a PE; rather a PE must take the initiative to find work for itself. When a PE is idle, it sends requests to its neighbors, asking for work. When a PE sees a request for work, and has a pool of ready tasks, it can send part of the pool to the idle neighbor. In this fashion ready processes "migrate" to an idle PE for execution. Each PE must be able to work on its own to solve any subproblem, in the event no neighbor requests work.

This distribution model is, obviously, decentralized. Locality can be enforced by limiting the number and distance of moves made by a process. An even distribution of work depends on strategies for deciding which parts of a problem to keep and which parts to let go. As an example of how processes in the AND/OR Process Model could be evenly distributed, consider an AND process for solving a typical problem involving tail recursion, where `X` is bound and `Y` is unbound in a call to `p`:

$$\texttt{p([X1|Xn],[Y1|Yn])} \leftarrow \texttt{q(X1,Y1)} \wedge \texttt{p(Xn,Yn).}$$

In a parallel AND process, both goals on the right hand side are solved in parallel, and thus two OR processes are created and sent start messages at the same time. The PE solving the AND process could keep the OR process that solves `q(X1,Y1)`, and let the OR process for `p(Xn,Yn)` migrate to a neighbor. `Xn` is a list of terms; usually each term in this list has the same general structure as `X1`. Using this policy of sending the "tail" part of the tail recursion, the problem could unfold along a "line" of PEs, each solving one of the goals `q(Xi,Yi)`, and the work would be apportioned evenly. Since the only message passing in the system is between parent/descendant pairs, the locality of message transfers is not affected by this policy.

There is a tradeoff here, since in these problems the term `Xn` is a list of terms, each of the form `X1`. Depending on the representation of terms, the list in `p(Xn,Yn)` is likely to be larger than the terms in `q(X1,Y1)`. The tradeoff is that in order to spread the work evenly, the larger goals must be passed from one PE to the next.

Another potential difficulty is related to the topology of the underlying network. The above scenario of unfolding a "line" of work is very likely when the topology is a ring, where PE_i has only two neighbors, PE_{i-1} and PE_{i+1}. However, in a more richly connected network it is not clear what path from PE_i to PE_j would correspond to a "line." Defining a set of policies to help each PE decide which processes to send to its neighbors, and analyzing the tradeoffs involved in the context of various topologies, is the subject of future work and simulations.

When large problems are being solved, the system will reach a point where each PE will be actively working on a problem. Since each PE is busy, no requests for work will be transmitted, and no more subproblems will be passed around the network. Thus one immediate advantage of this method is that message traffic is not strictly a function of problem size. When a large problem is solved, there will be a large amount of traffic as subproblems are initially spread around, but eventually a point will be reached where each PE is busy working on its own part of the overall problem. Other techniques for dynamic allocation are based on a mechanism for assigning tasks to processors that will perform the task. An example is the hashing function that maps activity names into processor IDs in the Irvine Dataflow system [37]. If an assignment function is used, new tasks are mapped onto processors independently of the amount of message traffic, and as the problem grows the number of messages grows along with it.

7.4 Growth Control

7.4.1 Conditional Expressions

An idea explored by Page, Conant, and Grit [69] for controlling growth through rules for executing conditional expressions can also be used in parallel logic programs. Given a conditional expression in a functional language, such as

```
f(X) = if p(X) then g(X) else h(X)
```

we have two choices for scheduling the evaluation. The "eager beaver" policy is to evaluate `p`, `g`, and `h` all in parallel. As soon as the value of `p` is determined, the computation of either `g` or `h` can be aborted if it is still active. This policy wastes a lot of resources, since one branch of the conditional is always ignored. The more restrained approach would evaluate `p` first, and then either `g` or `h`, depending on the outcome. The overall evaluation of `f` is slower, but no resources are wasted.

In an earlier chapter we saw how to write the equivalent conditional expression as a logic program:

```
f(X,Y) ← p(X) ∧ g(X,Y).
f(X,Y) ← not(p(X)) ∧ h(X,Y).
```

A parallel OR process for the goal `f(a,Y)` would create two AND descendants simultaneously. One of these would start an OR process for `p(a)` and the other would start an OR process for `not(p(a))`. Since, presumably, one of these always fails, this is a waste of resources.

A better technique for this situation is to introduce a conditional operation such as the `->` operator of DEC-10 Prolog. In the clause

```
f(X,Y) :- p(X) -> g(X,Y) ; h(X,Y).
```

if `p(X)` succeeds, the system continues with `g(X,Y)`, otherwise it solves `h(X,Y)`. In a parallel system, an AND process for this clause could create an OR process for `p(X)`; then, depending on the result, it would create a process for either `g(X,Y)` or `h(X,Y)`. Note that the dataflow graph for the body of this clause has three independent nodes if `X` is bound when the clause is called. When the system is not busy it could start all three at the same time; when the load is high, it could be more conservative and wait until `p` is solved or not before starting a process for one of the other two.

7.4.2 Process Priorities

A second inhibitor of parallelism is simply to switch to a sequential computation when the PEs start to become loaded. However, as discussed in Chapter 3, it would be a mistake to have a PE start using a depth first interpreter when it thinks the system is heavily loaded, since a depth first interpreter does not always generate as many results as a parallel interpreter. If the system switched models depending on current workload, a procedure might generate a result when used in a small program and fail to generate the same result when called from a larger program.

The PEs should keep executing according to the rules of the AND/OR Process Model, but in a mode where the processes selected for execution are, in general, the processes that would execute in a sequential system. This can be implemented by assigning a priority to each new process. For example, when an OR process creates more than one AND descendant when the system is busy, it could give the process for the first clause in the procedure a higher priority than the remaining processes, and the OR process could pass its priority level on to its descendants.

7.4.3 Message Protocols

An idea mentioned briefly in Chapter 3 is the use of communication protocols for growth control. In the basic model, OR processes immediately send a redo

message to their AND descendants each time the descendant sends a success. This protocol keeps the system busy in anticipation that all results will be needed as soon as possible.

The back-up OR parallelism in the system of Furukawa, Nitta, and Matsumoto takes a much more conservative approach. In this system an OR process tries to stay one jump in front of its parent, by keeping just one additional result at any time. An interesting area for future research is the development of protocols that allow varying amounts of parallel activity, based on programmer annotations or system load factors. Another possibility to explore is the use of bounded buffers in the communication paths between processes [15]; if a descendant tries to send a success on a full channel, it would be blocked, unable to process any of its incoming messages, until there is room in the channel.

7.4.4 Secondary Memory

In spite of efforts to control growth, it is inevitable that a point will be reached when there is too much activity in the system. When that happens, blocked processes will have to be stored in secondary memory until they are ready for execution.

Simulations done so far show that as the AND/OR tree of processes is formed, the processes toward the top of the tree will be idle while descendants at the frontier of the tree actively carry out their tasks. Eventually success messages work back to the top of the process tree. This observation can form the basis of an efficient use for a secondary memory.

When a PE's memory starts to become filled with processes, blocked processes can be written out to the secondary memory. The processes written out should be those at the top of the tree. When room is made available in main memory again, processes can be read back in. One can envision a "vacuum" effect here as the processes are brought back in as the amount of memory devoted to active processes shrinks. The decision of which processes to bring back should be based on how close they are to the current frontier of the AND/OR tree of processes. The system can anticipate their need, before any active process actually sends a message to one of them. A similar idea is used in the FFP machine, where symbols that expand past the end of the array of L cells are stored in stacks in virtual memory [62]. Secondary memory systems in von Neumann architectures are not so predicatable. If a program makes a reference to information not currently in main memory, it is blocked until the information is retrieved. There is no way to anticipate which information currently on disk will be needed next, so information stays there until there is a demand for it. In the AND/OR Process Model, the regular structure of process interconnection, and the relative predictability of when a process will be activated, may lead to very efficient use of secondary memory.

This idea can be extended to situations where memory is filled with only ready processes, after all blocked processes have been moved to secondary storage. During the solution of very large problems, a PE's memory will overflow with processes and messages. The first step in alleviating the congestion is to move out blocked processes, those corresponding to activity at the top of the AND/OR tree. As the tree continues to grow, a point will be reached where memory will contain only ready processes and their incoming messages. The second stage is to store some subset of these active processes and their messages. Again, the regular structure of the tree of processes will help determine which process/message pairs to move out. Siblings, or processes at the same level in general, do not send messages directly to one another. So, at this stage, processes and messages from the "bottom right" of the tree can be stored, since execution of processes from the "bottom left" will not send them messages. A global view of the expansion of the AND/OR tree of processes can be characterized as mostly breadth first, as processes create descendants in parallel. When PEs become saturated, and active processes moved out of main memory, the expansion will tend toward depth first, as the leftmost parts of the tree are expanded while the rightmost part stays in secondary memory.

The success of this scheme for storing inactive processes will depend heavily on the structure and operation of connections between the PEs and secondary memory. One possibility is to use a second network for this use; have PEs send processes and messages via the primary interconnection network, and use the second for storing processes. This may not be necessary if the migration scheme for process allocation is used. With a migration policy, when the system is overloaded with processes, the communication channels will not be heavily used, since each node is busy. In that case the channels are free for moving blocked processes to nodes connected to secondary memory.

7.5 Summary

The AND/OR Process Model is an abstract model for parallel execution of logic programs. It is an execution model, an abstract interpreter at the Operation layer of the hierarchy of Figure 1.1. In this chapter we moved down a level to the Implementation layer. Techniques used in four different implementations were sketched. Two of the more difficult problems – representation of binding environments and efficient implementation of backward execution – have been solved, and there are promising ideas for the problems of growth control and process allocation. All of the ideas discussed here need to be extensively tested in multiprocessor implementations, and the AND/OR Process Model itself compared with other parallel models, before we consider moving on to the next layer.

Bibliography

[1] Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641.

[2] Bic, L. Data-driven logic: A basic model. In *Proceedings of the International Workshop on High-Level Computer Architecture 84*, (May 21–25), 1984, pp. 1.20–1.25.

[3] Borgwardt, P. Parallel Prolog using stack segments on shared memory multiprocessors. In *Proceedings of the 1984 International Symposium on Logic Programming*, (Atlantic City, NJ, Feb 6–9), 1984, pp. 2–11.

[4] Borgwardt, P. and Rea, D. *Distributed Semi-Intelligent Backtracking for a Stack-Based AND-parallel Prolog*. CRL Tech. Rep. CR86-06, Computer Research Laboratory, Tektronix, Inc., July 1986. (To be presented at the 1986 Symposium on Logic Programming, Salt Lake City, UT.).

[5] Bowen, K.A. Concurrent execution of logic. In *Proceedings of the First International Logic Programming Conference*, (Faculté des Sciences de Luminy, Marseille, France, Sept.), 1982, pp. 26–30.

[6] Bruynooghe, M. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters 12*, 1 (Feb. 1981), 36–39.

[7] Bruynooghe, M. and Pereira, L.M. *Deduction Revision by Intelligent Backtracking*. Report CW 30, Katholieke Universiteit Leuven, Heverlee, Belgium, 1983.

[8] Burton, F.W. and Sleep, M.R. Executing functional programs on a virtual tree of processors. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, (Wentworth-by-the-Sea, NH, Oct. 18–22), ACM, 1981, pp. 187–194.

[9] Chang, J-H and Despain, A.M. Semi-intelligent backtracking of Prolog based on static dependency analysis. In *Proceedings of the 1985 International Symposium on Logic Programming*, (Boston, MA, July 15–18), 1985, pp. 10–21.

[10] Chang, J-H, Despain, A.M., and DeGroot, D. AND-parallelism of logic programs based on static data dependency analysis. In *COMPCON Spring 85*, (Feb.), IEEE, 1985, pp. 218–225.

[11] Ciepielewski, A. and Haridi, S. *Formal Models for Or-Parallel Execution of Logic Programs*. CSALAB Working Paper 821121, Royal Institute of Technology, Stockholm, Sweden, 1982.

[12] Ciepielewski, A. and Haridi, S. *Storage Models for Or-Parallel Execution of Logic Programs*. Tech. Rep. TRITA-CS-8301, Royal Institute of Technology, Stockholm, Sweden, 1983.

[13] Ciepielewski, A. and Haridi, S. Control of activities in the OR-parallel token machine. In *Proceedings of the 1984 International Symposium on Logic Programming*, (Atlantic City, NJ, Feb 6–9), 1984, pp. 49–57.

[14] Clark, K.L. Negation as failure. In Gallaire, H. and Minker, J., editors, *Logic and Databases*, Plenum Press, 1978.

[15] Clark, K.L. and Gregory, S. A relational language for parallel programming. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, (Wentworth-by-the-Sea, NH, Oct. 18–22), ACM, 1981, pp. 171–178.

[16] Clark, K.L. and Gregory, S. Notes on system programming in Parlog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, (Tokyo, Japan), 1984, pp. 299–306.

[17] Clark, K.L. and Gregory, S. PARLOG: Parallel programming in logic. *ACM Trans. Prog. Lang. Syst. 8*, 1 (Jan. 1986), 1–49.

[18] Clark, K.L. and McCabe, F. The control facilities of IC-Prolog. In Michie, D., editor, *Expert Systems in the Microelectronic Age*, Edinburgh University Press, 1979.

[19] Conery, J.S. *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, Univ. of California, Irvine, 1983. (Computer and Information Science Tech. Rep. 204).

[20] Conery, J.S. *Closed Environments: Partitioned Memory Representation for Parallel Logic Programs*. Tech. Rep. 86-02, University of Oregon, 1986.

[21] Conery, J.S. and Kibler, D.F. Parallel interpretation of logic programs. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, (Wentworth-by-the-Sea, NH, Oct. 18–22), ACM, 1981, pp. 163–170.

[22] Crammond, J.A. A comparative study of unification algorithms for or-parallel execution of logic languages. In *Proceedings of the 1985 International Conference on Parallel Processing*, (August 20–23), 1985, pp. 131–138.

[23] Crammond, J.A. and Miller, C.D.F. An architecture for parallel logic languages. In *Proceedings of the Second International Logic Programming Conference*, (Uppsala, Sweden, July 2–6), 1984, pp. 183–194.

[24] Dahl, V. On database systems development through logic. *ACM Trans. Database Syst. 7*, 1 (March 1982), 102–123.

[25] DeGroot, D. Restricted AND-parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, (Tokyo, Japan), 1984, pp. 471–478.

[26] Deliyanni, A. and Kowalski, R.A. Logic and semantic networks. *Commun. ACM 22*, 3 (March 1979), 184–192.

[27] Dembinski, P. and Maluszynski, J. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the 1985 International Symposium on Logic Programming*, (Boston, MA, July 15–18), 1985, pp. 29–38.

[28] Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[29] Dobry, T., Despain, A.M., and Patt, Y. Performance studies of a Prolog machine architecture. In *Proceedings of the 12th International Symposium on Computer Architecture*, (Boston, MA, June 17–19), 1985, pp. 180–190.

[30] Dwork, C., Kanellakis, P.C., and Mitchell, J.C. On the sequential nature of unification. *J. Logic Programming 1*, 1 (June 1984), 35–50.

[31] Eisinger, N., Kasif, S., and Minker, J. Logic programming: A parallel approach. In *Proceedings of the First International Logic Programming Conference*, (Faculté des Sciences de Luminy, Marseille, France, Sept.), 1982, pp. 1–8.

[32] van Emden, M.H. and Kowalski, R.A. The semantics of predicate logic as a programming language. *J. ACM 23*, 4 (Oct. 1976), 773–742.

[33] van Emden, M.H. and de Lucena Filho, G.J. Predicate logic as a language for parallel programming. In Clark, K.L. and Tärnlund, S-Å., editors, *Logic Programming*, Academic Press, New York, NY, 1982, pp. 189–198.

[34] Furukawa, K., Nitta, K., and Matsumoto, Y. Prolog interpreter based on concurrent programming. In *Proceedings of the First International Logic Programming Conference*, (Faculté des Sciences de Luminy, Marseille, France, Sept.), 1982, pp. 38–44.

[35] Gallaire, H. and Minker, J., editors. *Logic and Data Bases*. Plenum Press, New York, NY, 1978.

[36] Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[37] Gostelow, K.P. and Thomas, R. Performance of a simulated dataflow computer. *IEEE Trans. Comput. C-29*, 10 (Oct. 1980), 905–919.

[38] Goto, A., Tanaka, H., and Moto-Oka, T. Highly parallel inference engine PIE – goal rewriting model and machine architecture. *New Generation Computing 2*, (1984), 37–58.

[39] Halim, Z. and Watson, I. An OR-parallel data-driven model for logic programs. In *Proceedings of the International Workshop on High-Level Computer Architecture 84*, (May 21–25), 1984, pp. 1.26–1.36.

[40] Haridi, S. *Logic Programming Based on a Natural Deduction System*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 1981. Report TRITA-CS-8104.

[41] Hewitt, C.E., Attardi, G., and Lieberman, H. Specifying and proving properties of guardians for distributed systems. In Kahn, G., editor, *Semantics of Concurrent Computation*, Springer-Verlag, New York, NY, 1979, pp. 316–336.

[42] Hoare, C.A.R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–667.

[43] Hogger, C.J. Concurrent logic programming. In Clark, K.L. and Tärnlund, S-Å., editors, *Logic Programming*, Academic Press, New York, NY, 1982, pp. 199–211.

[44] Hopcroft, J.E. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.

[45] Ito, N. and Masuda, K. Parallel inference machine based on the data flow model. In *Proceedings of the International Workshop on High-Level Computer Architecture 84*, (May 21–25), 1984, pp. 4.31–4.40.

[46] Ito, N., Shimizu, H., Kishi, M., Kuno, E., and Rokusawa, K. Data-flow based execution mechanisms of parallel and Concurrent Prolog. *New Generation Computing 3*, 1 (1985), 15–41.

[47] Kacsuk, P. A highly parallel Prolog interpreter based on the generalized data flow model. In *Proceedings of the Second International Logic Programming Conference*, (Uppsala, Sweden, July 2–6), 1984, pp. 195–205.

[48] Kalé, L.V. *Parallel Architectures for Problem Solving*. PhD thesis, SUNY Stony Brook, Dec. 1985. (Univ. of Illinois at Urbana-Champaign Tech. Rep. UIUCDCS-R-85-1237).

[49] Kaneda, Y., Tamura, N., Wada, K., and Matsuda, H. Sequential Prolog machine PEK architecture and software system. In *Proceedings of the International Workshop on High-Level Computer Architecture 84*, (May 21–25), 1984, pp. 4.1–4.6.

[50] Kasif, S., Kohli, M., and Minker, J. PRISM: A parallel inference system for problem solving. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, (Karlsruhe, Germany, Aug. 8–12), 1983, pp. 544–546.

[51] Katz, E.P. *A Realization of Relational Semantics in an Automatic Programming System*. PhD thesis, Univ. of Southwest Louisiana, 1978.

[52] Keller, R.M., Lin, F.C.H., and Tanaka, J. Rediflow multiprocessing. In *COMPCON Spring 84*, IEEE, Feb. 1984, pp. 410–417.

[53] Kibler, D.F. and Porter, B. Episodic learning. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, (Karlsruhe, Germany, Aug. 8–12), 1983.

[54] Kowalski, R.A. Predicate logic as a programming language. In *Information Processing 74*, IFIPS, 1974, pp. 569–574.

[55] Kumon, K., Masuzawa, H., Itashiki, A., Satoh, K., and Sohma, Y. KABU-WAKE: A new parallel inference method and its evaluation. In *COMPCON Spring 86*, IEEE, 1986.

[56] Li, G. and Wah, B.W. MANIP-2: A multicomputer architecture for evaluating logic programs. In *Proceedings of the 1985 International Conference on Parallel Processing*, (August 20–23), 1985, pp. 123–130.

[57] Li, P. and Martin, A.J. *The Sync Model for Parallel Execution of Logic Programming*. Tech. Rep., California Inst. of Technology, July 1986. (To be presented at the 1986 Symposium on Logic Programming, Salt Lake City, UT.).

[58] Lindstrom, G. OR parallelism on applicative architectures. In *Proceedings of the Second International Logic Programming Conference*, (Uppsala, Sweden, July 2–6), 1984, pp. 159–170.

[59] Lindstrom, G. and Panangaden, P. Stream based execution of logic programs. In *Proceedings of the 1984 International Symposium on Logic Programming*, (Atlantic City, NJ, Feb 6–9), 1984, pp. 168–176.

[60] Lipovski, G.J. and Hermenegildo, M.V. B-Log: A branch and bound methodology for the parallel execution of logic programs. In *Proceedings of the 1985 International Conference on Parallel Processing*, (August 20–23), 1985, pp. 560–567.

[61] MacLennan, B.J. Introduction to relational programming. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, (Wentworth-by-the-Sea, NH, Oct. 18–22), ACM, 1981, pp. 213–220.

[62] Magó, G. The FFP machine – A progress report. In *Proceedings of the International Workshop on High-Level Computer Architecture 84*, (May 21–25), 1984, pp. 5.13 –5.25.

[63] Martelli, A. and Montanari, U. An efficient unification algorithm. *ACM Trans. Prog. Lang. Syst. 4*, 2 (Apr. 1982), 258–282.

[64] Monteiro, L. A Horn clause-like logic for specifying concurrency. In *Proceedings of the First International Logic Programming Conference*, (Faculté des Sciences de Luminy, Marseille, France, Sept.), 1982, pp. 1–8.

[65] Nakagawa, H. AND parallel Prolog with divided assertion set. In *Proceedings of the 1984 International Symposium on Logic Programming*, (Atlantic City, NJ, Feb 6–9), 1984, pp. 22–28.

[66] Nakamura, K. Associative concurrent evaluation of logic programs. In *Proceedings of the Second International Logic Programming Conference*, (Uppsala, Sweden, July 2–6), 1984, pp. 321–331.

[67] Nakazaki, R., *et al.* Design of a high-speed Prolog machine (HPM). In *Proceedings of the 12th International Symposium on Computer Architecture*, (Boston, MA, June 17–19), 1985, pp. 191–197.

[68] Nilsson, N.J. *Problem Solving Methods in Artificial Intelligence*. McGraw-Hill, New York, NY, 1971.

[69] Page, R.L., Conant, M.G., and Grit, D.H. If-then-else as a concurrency inhibitor in eager beaver evaluation of recursive programs. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, (Wentworth-by-the-Sea, NH, Oct. 18–22), ACM, 1981, pp. 179–186.

[70] Pereira, F.C.N. and Warren, D.H.D. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence 13*, (1980), 231–278.

[71] Pereira, L.M. *Logic Control with Logic*. Report 2/82, Dept. de Informatica, Univ. Nova de Lisboa, Feb. 1982.

[72] Pereira, L.M., Pereira, F.C.N., and Warren, D.H.D. *Users Guide to DECsystem-10 Prolog*. Tech. Rep., Dept. of Artificial Intelligence, Univ. of Edinburgh, Sep. 1978.

[73] Pereira, L.M. and Porto, A. *Intelligent Backtracking and Sidetracking in Horn Clause Programs – the Theory*. Report 2/79, Dept. de Informatica, Univ. Nova de Lisboa, Oct. 1979.

[74] Pereira, L.M. and Porto, A. *An Interpreter of Logic Programs Using Selective Backtracking*. Report 3/80, Dept. de Informatica, Univ. Nova de Lisboa, July 1980.

[75] Robinson, I. *A Prolog Processor Based on a Pattern Matching Memory Device*. Tech. Rep., Schlumberger Palo Alto Research AI Lab, Palo Alto, CA, 1986. (To be presented at the 1986 International Conference on Logic Programming, London, England.).

[76] Robinson, J.A. A machine oriented logic based on the resolution principle. *J. ACM 12*, 1 (Jan. 1965), 23–41.

[77] Shapiro, E.Y. and Takeuchi, A. Object oriented programming in Concurrent Prolog. *New Generation Computing 1*, (1983), 25–48.

[78] Shapiro, E.Y. *A Subset of Concurrent Prolog and its Interpreter*. Tech. Rep. TR-003, Institute for New Generation Computer Technology, Tokyo, Japan, Jan. 1983.

[79] Singh, V. and Genesereth, M.R. *PM: A Parallel Execution Model for Backward-Chaining Deductions*. KSL Report KSL-85-18, Knowledge Systems Laboratory, Computer Science Department, Stanford Univ., May 1985.

[80] Subrahmanyam, P.A. The "software engineering" of expert systems: Is Prolog appropriate? *IEEE Trans. Softw. Eng. SE-11*, 11 (Nov. 1985), 1391–1400.

[81] Taki, K., Yokota, M., Yamamoto, A., Nishikawa, H., Uchida, S., Nakashima, H., and Mitsuishi, A. Hardware design and implementation of the personal sequential inference machine (PSI). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, (Tokyo, Japan), 1984, pp. 398–409.

[82] Tamura, N. and Kaneda, Y. Implementing parallel Prolog on a multiprocessor machine. In *Proceedings of the 1984 International Symposium on Logic Programming*, (Atlantic City, NJ, Feb 6–9), 1984, pp. 42–48.

[83] Taylor, S., Lowry, A., Maguire, G.Q., and Stolfo, S.J. Logic programming using parallel associative operations. In *Proceedings of the 1984 International Symposium on Logic Programming*, (Atlantic City, NJ, Feb 6–9), 1984, pp. 58–68.

[84] Tick, E. Towards a multiple pipelined Prolog processor. In *Proceedings of the International Workshop on High-Level Computer Architecture 84*, (May 21–25), 1984, pp. 4.7–4.17.

[85] Tick, E. and Warren, D.H.D. Towards a pipelined Prolog processor. In *Proceedings of the 1984 International Symposium on Logic Programming*, (Atlantic City, NJ, Feb 6–9), 1984, pp. 29–40.

[86] Turner, D.A. A new implementation technique for applicative languages. *Software – Practice and Experience 9*, 1 (Jan. 1979), 31–49.

[87] Ueda, K. *Guarded Horn Clauses*. Tech. Rep. TR-103, Institute for New Generation Computer Technology, Tokyo, Japan, June 1985.

[88] Umeyama, S. and Tamura, K. A parallel execution model of logic programs. In *Proceedings of the 10th International Symposium on Computer Architecture*, (Stockholm, Sweden, June 13–17), 1983, pp. 349–355.

[89] Warren, D.H.D. *WARPLAN: A System for Generating Plans*. Paper 76, Dept. of Artificial Intelligence, Univ. of Edinburgh, June 1974.

[90] Warren, D.H.D. *Applied Logic – Its Use and Implementation as a Programming Tool*. PhD thesis, Univ. of Edinburgh, 1977. (SRI Tech. Note 290, June 1983).

[91] Warren, D.H.D. *Implementing Prolog: Compiling Predicate Logic Programs*. DAI Research Papers 39 and 40, Dept. of Artificial Intelligence, Univ. of Edinburgh, May 1977.

[92] Warren, D.H.D. Logic programming and compiler writing. *Software – Practice and Experience 10*, (1980), 97–125.

[93] Warren, D.H.D. *Efficient Processing of Interactive Relational Database Queries Expressed in Logic*. Paper 156, Dept. of Artificial Intelligence, Univ. of Edinburgh, Sep. 1981.

[94] Warren, D.H.D. *Higher-Order Extensions to Prolog – Are They Needed?* Paper 165, Dept. of Artificial Intelligence, Univ. of Edinburgh, Sep. 1981.

[95] Warren, D.H.D. *An Abstract Prolog Instruction Set*. Tech. Note 309, SRI International, Oct. 1983.

[96] Warren, D.H.D. and Pereira, F.C.N. *An Efficient Easily Adaptable System for Interpreting Natural Language Queries*. Paper 155, Dept. of Artificial Intelligence, Univ. of Edinburgh, Feb. 1981.

[97] Warren, D.H.D., Pereira, L.M., and Pereira, F.C.N. Prolog – the language and its implementation compared with LISP. *ACM SIGPLAN Notices 12*, 8 (1977), 109–115.

[98] Warren, D.S., Ahamad, M., Debray, S.K., and Kalé, L.V. Executing distributed Prolog programs on a broadcast network. In *Proceedings of the 1984 International Symposium on Logic Programming*, (Atlantic City, NJ, Feb 6–9), 1984, pp. 12–21.

[99] Wise, M.J. A parallel Prolog: The construction of a data driven model. In *Conference Record of the Symposium on LISP and Functional Programming*, (Pittsburgh, PA, Aug. 15–18), ACM, 1982, pp. 55–66.

[100] Woo, N.S. and Choe, K-M. *Selecting the Backtrack Literal in the AND Process of the AND/OR Process Model*. Tech. Rep., AT&T Bell Labs, Murray Hill, NJ, Jan. 1986. (To be presented at the 1986 Symposium on Logic Programming, Salt Lake City, UT.).

[101] Wulf, W.A. and Shaw, M. Abstraction and verification in ALPHARD: Defining and specifying iteration and generators. *Commun. ACM 20*, 8 (Aug. 1977), 553–564.

[102] Yasuhara, H. and Nitadori, K. ORBIT: A parallel computing model of Prolog. *New Generation Computing 2*, (1984), 277–288.

Index

m

n

o

p