Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I

Vítor Manuel de Morais Santos Costa

A thesis submitted to the University of Bristol in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science.

August 1993

Abstract

Logic programming languages, such as Prolog, provide a high-level view of programming. These languages allow a form of programming where one declares the logic of the problem, plus the control necessary for efficient execution. Logic programs can take advantage of recent parallel architectures by exploiting implicit parallelism, where and-parallelism results from the parallel execution of goals, and or-parallelism results from the parallel execution of alternatives to the same goal.

Andorra-I is an experimental parallel Prolog system that transparently exploits both dependent and-parallelism and or-parallelism. It implements the Basic Andorra Model, a parallel execution model for logic programs in which determinate goals are executed in parallel. This model not only combines two of the most important forms of implicit parallelism in logic programs, it also allows a form of implicit coroutining. This means that Andorra-I not only supports standard Prolog but also provides the capabilities of flat committed-choice languages.

The main subject of this thesis is the Andorra-I preprocessor, that supports the execution of Prolog programs in the Basic Andorra Model. The preprocessor analyses clauses heads and builtins in the body to generate determinacy routines. These routines are used at run-time to verify when at most a clause will match a goal. To cater for Prolog programs containing builtins such as side-effects or some cuts that depend on a left-to-right order of execution, the preprocessor includes a sequencer that generates sequential annotations to guarantee the correct execution of these builtins. An abstract interpretation module performs global analysis to assist the sequencer in detecting the cuts and meta-predicates which need sequencing. Finally, the preprocessor includes a compiler that generates WAM-style code for each clause.

With the aid of the preprocessor, a number of substantial Prolog applications have already been successfully ported to Andorra-I. We discuss the performance of the different components of the preprocessor and the performance of the Andorra-I system as a whole, and finally we present some possible extensions to the Basic Andorra Model.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or institution of learning.

Vítor Manuel de Morais Santos Costa

Acknowledgements

Working at Bristol has been a very rewarding experience for me. Of all the many people who have made the contributions in thesis possible, I would like to thank my supervisor, David H. D. Warren, first. I have learned much from his vast experience and knowledge in logic programming. His interest and the many discussions we have had throughout my research are a major influence on this work.

I have also been very fortunate to work with Rong Yang. Her contributions to Andorra-I have made my work possible, and her different approach to logic programming has widened my interests in this field. Thank you very much.

I also thank Clive Williams, Gopal Gupta, Inês Dutra, Kish Shen, Péter Szeredi, Raéd Sindaha, Sanjay Raina, Tony Beaumont, and many others at Bristol for our many discussions and for your very good friendship. Inês and Tony have given major contributions to our system. My Sainsbury's walks with Gopal led me to research in other forms of parallelism in logic programs. Talking to Kish and Raéd is always a fruitful and welcome experience. I also thank the Andorra-I users, people such as Steve Gregory, Andrea Domenici, and Dewi Munaf, for their comments and patience.

I am most grateful to the people who have read drafts of this thesis. David D. H. Warren's careful reading and suggestions have much improved this thesis. Ahmed, André, Inês, Kish, and Rong have suggested many needed improvements.

Working at Bristol has allowed to meet and learn from people from logic programming groups throughout the world. I particularly thank the Esprit PEPMA project for this. My research has gained much from the ideas of people such as Reem Bahgat, Seif Haridi, Manuel Hermenegildo, and Lee Naish. I also thank Manuel Hermenegildo for my very fruitful visit to Madrid.

I thank Luís Damas and all the people at my group in Porto for introducing me to Prolog, and for our work in YAP. I say obrigado to Fernando Silva, João Lopes, Nelma Moreira, Rogério Reis, Zé Paulo Leal, and all of you, for your help and friendship throughout these years, and especially through this last year.

My work has been made possible by the support of the Universidade do Porto, Fundação Calouste Gulbenkian, and Esprit Project from the European Community, to whom I am most grateful.

Last, but not the least, I say thank you for everything to my parents.

To Inês

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Developments in computing have been dominated by the rise of ever more powerful hardware. Processing speed and memory capacity have increased dramatically over the last decades. Parallel computers connect together several processing units to obtain even higher performance. Unfortunately, progress in software has been much less impressive. One reason is that most programmers still rely on traditional, imperative languages, and high-level tasks are difficult to express on an imperative language primarily concerned with how memory positions are to be updated. This low-level approach to programming is also cumbersome when programming parallel computers, as the details of control flow can become very complex, and as the best execution strategy can very much depend on a computer's architecture and configuration.

In contrast to the traditional programming languages, *logic programming* provides a high-level view of programming. In this approach, programs are fundamentally seen as a collection of statements that define a model of the intended problem. Questions may be asked against this model, and can be answered by an inference system, with the aid of some user-defined control. The combination was summarised by Kowalski [95]:

$$algorithm \ = \ logic \ + control$$

Traditionally, logic programming systems are based on Horn clauses, a natural and useful subset of First Order Logic. For Horn clauses, a simple proof algorithm, SLD resolution, provides clear operational semantics and can be implemented efficiently. The most popular logic programming language is Prolog [35]. Throughout its history, Prolog has exemplified the use of logic programming for applications such as artificial intelligence, database programming, circuit design, genetic sequencing, expert systems, compilers, simulation and natural language processing. Other logic programming languages have been successfully used in areas such as constraint based resource allocation and optimisation problems, and on operating system design.

Logic programming systems are also a good match for parallel computers. As different execution schemes may be used for the same logic program, forms of program execution can be developed to best exploit the advantages of the parallel architecture used. This means that parallelism in logic programs can be exploited implicitly, and that the programmer can be left free to concentrate on the logic of the program and on the control information necessary to obtain efficient algorithms.

In our work we aim towards a parallel logic programming system that (i) will allow the user to be mainly concerned with the logic of the program and with the sufficient control to obtain an efficient algorithm; (ii) will obtain good performance versus sequential systems; and (iii) will exploit maximum parallelism. Such systems should be be a correct and efficient implementation of a parallel execution model giving the most implicit parallelism possible. In this thesis, we claim that compile-time analysis is fundamental to build such a system:

- Compilation reduces the overheads of a parallel model and is necessary to obtain performance close to traditional Prolog systems.
- Logic programming languages such as Prolog include input/output and other primitives that may depend on a certain execution scheme. Global analysis may be necessary to guarantee correct execution of these primitives on sophisticated parallel execution schemes.

We demonstrate our claims in the context of the Andorra-I system, a system that arguably extracts the most useful forms of implicit and-parallelism. We show that through compile-time analysis Andorra-I can obtain good performance for a wide range of Prolog and other logic programming language applications.

We next motivate this work with a brief discussion of logic programming languages and of implicit parallelism in logic programs. After this, we present the main principles of Andorra-I and explain where compile-time analysis is most important.

1.1 Logic Programming Languages

Logic programs consist of Horn clauses. Horn clauses may be facts, saying that some goal (called the head of the clause) is true, or rules, saying that some goal (again called the head of the clause), is true if some other goals are true. Execution proceeds by trying to satisfy goals, usually starting from an original goal given by the the user, the query. Each goal is tried with a clause. If it matches the head of a fact, the goal succeeds. If it matches the head of a rule, Prolog tries to satisfy the goals in the body. If no head matches the goal, the goal fails. Note that several heads of clauses may match a goal.

Prolog uses depth-first search and always selects the leftmost goal first. Control is mainly given through clause and goal ordering. For instance, if a goal depends on some other goal, it should be placed after that goal in the clause. The Prolog language also includes some control features necessary for effective programming. These include the pruning operator cut, builtins to generate Input/Output interactions, and the meta-predicates to test the state of the computation.

Prolog's execution strategy is simple to understand and efficient to execute. The price to pay is that control is expressed statically through the placement of goals in a rule. Many problems require data to be produced and consumed cooperatively by different entities, that is, through *coroutining*. Although such problems can often be written elegantly as logic programs, Prolog's fixed execution strategy does not (directly) support the necessary coroutining, resulting in inefficient execution. Hence problems with a simple and elegant logic formulation may end up as a complex and involved Prolog program.

Prolog can be extended with extra control features, such as Prolog-II's "geler" and NU-Prolog's "wait" (these two features delay execution of specific goals until certain conditions are fulfilled). This is a compromise solution, and as such the new features sometimes conflict with older features, such as cut.

The committed-choice languages are a different solution. These languages were designed to take advantage of data cooperation between goals. Whereas Prolog selects the leftmost goal, committed-choice languages say that a goal can be selected if it can *commit* to a clause, and that each clause is associated with a set of conditions that make execution of the goal meaningful. Thus, goals are selected, not according to the position, but according to whether the data they need to commit has been received. This can be an important advantage over Prolog. The drawback occurs for problems with several solutions. In Prolog the different solutions can be expressed as alternative clauses. In the committed-choice languages, only a single clause will be chosen, and more complex formulations are needed. This very much restricts the number of Prolog applications that can be easily ported to these languages.

1.2 Parallelism in Logic Programs

Two main forms of implicit parallelism have been recognised in logic programs: or-parallelism and and-parallelism. In *or-parallelism*, the several matching clauses for a goal are tried in parallel. In *and-parallelism* several goals are launched and solved in parallel. Two important cases of and-parallelism can arise. In *independent* and-parallelism, parallel goals must not share (or communicate via) logical variables. In *dependent* and-parallelism, goals can share variables, and different restrictions to and-parallelism apply.

Both forms of parallelism exist in logic programs, and therefore both can be exploited from most logic programs. Still, language design may affect which forms of parallelism are most promising.

One example is the committed-choice languages. In these languages, or-parallelism is limited because only one solution will eventually be considered. On the other hand, it is possible that several goals can be committed simultaneously. Thus, dependent and-parallelism can be exploited quite naturally, simply by running all these goals in parallel.

Prolog programs are in a different situation. Many Prolog programs return several solutions (or at least develop several partial solutions), thus or-parallelism is frequently found. Or-parallelism can be extracted quite naturally, simply by adapting the depth-first search rule of Prolog. Independent and-parallelism is also quite natural. One simply recognises (usually at compile-time) two or more goals in a clause such that they do not share variables. Prolog would execute them independently, and independent and-parallel system can run them in parallel. On the other hand, the dependent and-parallelism, which is so important in committed-choice systems, is not easily implementable with a leftmost goal selection function.

Quite a few systems that successfully exploit a single form of parallelism are available. They include the or-parallel systems Aurora [98] and Muse [2], the independent and-parallel system &-Prolog [74], and dependent and-parallel systems implementing committed-choice languages such as KL1 [139] and PARLOG [40].

These systems show that implicit parallelism in logic programs is practical and useful. Still, they are limited in that they explore only one form of parallelism. Programs that have many solutions will do well with Aurora or Muse, divide-and-conquer programs will do well with &-Prolog, concurrent programs will do well with Parallel PARLOG or GHC. But the user still has to choose the most suitable system. And if a program exhibits several forms of parallelism, not all the parallelism will be used. We would therefore prefer a system which would be able to extract several forms of parallelism. Andorra-I is such a system that exploits the two arguably most important forms of parallelism in logic programs, or-parallelism and and-parallelism between determinate goals.

1.3 The Basic Andorra Model

Andorra-I is based on the Basic Andorra Model, a model for the execution of logic programs. The model classifies goals as *determinate*, if at most one clause matches the goal, or nondeterminate, otherwise. The Basic Andorra Model says that:

- If determinate goals exist, one may execute them first and in parallel;
- If no determinate goals are available, select a nondeterminate goal.

The model has important advantages. First, it naturally supports implicit coroutining, as goals can make other goals immediately executable by making them determinate. This coroutining is in the style of the flat committed-choice languages. However, the Basic Andorra Model still supports goals with many matching clauses, and thus goals with several solutions. Indeed, the model naturally yields two forms of parallelism:

- Dependent And-Parallelism, by running *determinate* goals in parallel;
- Or-parallelism, by trying the different alternative clauses to a goal in parallel.

Note that dependent and-parallelism in the Basic Andorra Model arises in much the same way as in the committed-choice style of execution, and that or-parallelism arises in much the same way as in or-parallel Prolog systems.

1.3.1 Language Issues in the Basic Andorra Model

One of the most important considerations in designing a logic programming system such as Andorra-I is which language to support. One can take an existing language, such as Prolog or the committed-choice languages, or one can propose a new language.

If one wants to support an existing language, the main alternatives are Prolog and the committed-choice languages. Flat committed-choice languages could be supported, but they do have the important disadvantage that they do not exploit multiple solutions. Supporting Prolog has several advantages. First, pure Prolog programs will give the same solutions either executed by traditional left-to-right selection function or by selecting determinate goals first. Hence, an important class of Prolog programs can be run very simply in Andorra-I. Second, Prolog is still by far the most popular logic programming language, hence giving Andorra-I easy access to mainstream logic programming. Moreover, some Prolog applications should benefit from implicit coroutining.

We therefore support Prolog in the Andorra-I system. We do so by running the Prolog programs according to the Basic Andorra Model. Note that if the program includes primitives that depend on left-to-right selection of goals, then execution in Andorra-I is more complex, as the system may need to restrict early execution of some goals in order to generate the correct execution.

Supporting Prolog does not prevent us from obtaining the full benefits of the coroutining. Not only can Andorra-I take advantage of the coroutining to speed-up existing Prolog applications, it can also obtain good performance for programs where Prolog's left-to-right selection function would be much too inefficient. As an important example, consider programs written for the flat committed-choice languages. Some of these programs are Horn clause programs that depend on coroutining to perform well. Such programs are inefficient Prolog programs but can be ported directly to Andorra-I, where they will run as well as in their original environment. Other committed-choice programs need special control primitives. These programs can still be ported to Andorra-I, as long as equivalent control primitives are provided by Andorra-I.

We can now define the precise language that we support in Andorra-I, which we call *Andorra-I Prolog.* It is a logic programming language that extends Prolog with the implicit coroutining of the Basic Andorra Model. Therefore all Prolog programs are Andorra-I Prolog programs. In addition, Andorra-I Prolog includes new primitives designed to support other programming models, such as the committed-choice languages. Note that programs that include these features may not be valid Prolog programs.

One could be more ambitious and propose a completely new language based on this model (and indeed several authors have done so). The main advantage of a completely new language is that one can obtain a cleaner design. On the other hand, designing a new, hopefully better, programming language is not easy and has some drawbacks:

- One loses the advantages of supporting Prolog, and could in fact reduce the appeal of the parallel system. Note that one can extend a new language to support Prolog, but in this case one will have to deal with issues that Andorra-I tackles from the beginning.
- Research on how Prolog's builtins can be supported in Andorra-I's environment can

be used to design builtins and control operators that will be useful in any logic programming environment. On the other hand, special primitives designed to take best advantage of Andorra-I may be outdated by new execution models.

We believe that supporting Prolog in Andorra-I is not only a worthwhile goal in itself, but also one whose results should be considered by future work in logic programming language design.

1.4 The Preprocessor

The goal of this work was to design compile-time tools to obtain correct and efficient execution of Prolog programs in Andorra-I. To allow the correct execution of Prolog programs, we researched the operation of Prolog programs with traditional left-to-right selection function, and investigated which features allow early execution of goals, and for which features left-to-right needs to be enforced. To obtain efficient execution of programs in Andorra-I, we designed a compiler for Andorra-I. The compiler had to address the new characteristics of Andorra-I's selection function, and particularly the problem of determinacy detection.

The output of this work was the Andorra-I preprocessor. As we explained, the main innovation of the preprocessor over previous compile-time analysis tools results from addressing two new problems in Andorra-I: which goals *can* be executed early, and which goals *must not* be executed early.

A goal must not be executed early if it interferes with the correct operation of some builtin, such as a side-effect predicate or cut. This is rather hard to detect at run-time. We use the principle that some calls in the program are "sensitive", i.e., may behave incorrectly if goals executed later in the left-to-right execution are executed early. Compile-time analysis detects such calls, and detects for which goals it should restrict early execution.

A goal can be executed early if it is determinate. Determinacy can be verified at run-time by testing every clause for the procedure. Andorra-I uses a more efficient solution: at compile-time it generates some code for each procedure. This code must detect, very quickly at run-time, whether a call is determinate.

The tools we described are integrated in Andorra-I's *preprocessor*. The preprocessor thus includes a *sequencer*, that generates code to prevent early execution of goals, and a *determinacy code analyser*, that compiles code to detect determinate goals. The code generated by the preprocessor is executed by the Andorra-I *engine*.

The sequencer benefits from global information generated by an *abstract interpreter*. The information is mainly useful in detecting uses of cut and of meta-predicates that are not sensitive, and thus prevents some unnecessary sequencing.

The determinacy code analyser was originally designed to compile only determinacy code, but was since extended to fully compile the (Andorra-I) Prolog source program. A *clause compiler* generates code for individual clauses. The determinacy compiler integrates this code with the determinacy code to obtain the full procedure code. The end-code can then be run by the Andorra-I engine.

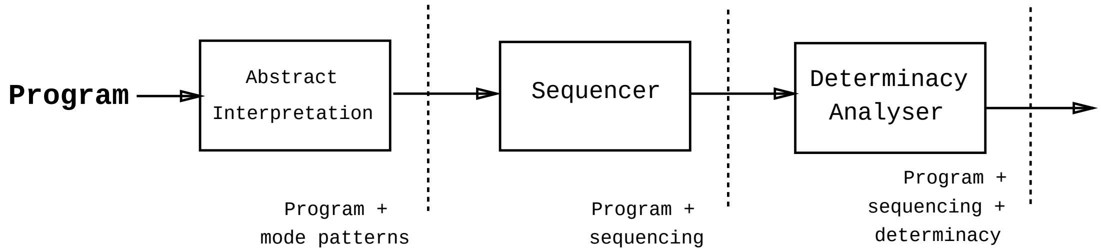

Figure 1.1: The structure of the preprocessor

Figure 1.1 presents the structure of the preprocessor is presented. Note that the first two components are mainly concerned with restricting the early execution of goals in Prolog programs. They could well apply to any Prolog system that allows early execution of goals. The determinacy compiler is mainly concerned with Andorra-I style execution, and it can (at least in part) be used for other programming languages to be supported on the Basic Andorra Model.

The compile-time analysis we have designed is part of the full Andorra-I system. The Andorra-I system started from the original goal of using David Warren's Basic Andorra Model to obtain both and-parallelism and or-parallelism, in logic programs. Development has been a group effort of the parallel logic programming group at Bristol, supervised by David Warren. We can discriminate as individual components the engine that actually executes the programs, and that was developed by Yang, the preprocessor we already presented, the and-scheduler that manages and-parallelism, and was also developed by Yang, the or-scheduler, that manages or-parallelism, mainly developed by Beaumont, and the top-scheduler, developed by Dutra.

1.5 Thesis Outline

This thesis describes the Andorra-I preprocessor as part of the parallel logic programming system Andorra-I, and presents the main components in of the preprocessor in detail.

Chapter 1 is the present chapter.

Chapter 2 discusses logic programming in more detail. The Prolog language and its implementation are discussed. Alternative logic programming languages, such as the committed-choice languages, are also presented in more detail, and the main issues in their implementation are discussed.

Chapter 3 discusses the issues involved in the parallel execution of logic programs, and gives a brief survey of the research in parallel logic programming systems. Emphasis is given to work influential in or related to the development of Andorra-I.

Chapter 4 introduces the Andorra-I system. We describe how the Basic Andorra Model can be used to obtain both or- and and-parallelism. Andorra-I supports Andorra-I Prolog and is designed to obtain correct and efficient execution of Prolog programs.

Chapter 5 describes in detail the problems we can found when supporting Prolog in Andorra-I. Most important are the problems that arise due to uses of builtins or of pruning operators that are sensitive. We explain in detail which uses of builtins are sensitive, and present a scheme for the execution of logic programs in Andorra-I.

Chapter 6 describes the sequencer, the part of the preprocessor which generates the annotations necessary for the correct execution of the extra-logical features of Prolog. The algorithms of the sequencer are described in detail and its performance is analysed.

Chapter 7 concentrates on the use of abstract interpretation by the preprocessor. The abstract interpretation system studies the execution of programs with the left-to-right selection function to obtain mode information that is used in the sequencer. Its design is discussed in detail in this chapter, and its performance analysed.

Chapter 8 presents in more detail Andorra-I Prolog. The chapter presents how the coroutining in the Basic Andorra Model can be used to run committed-choice applications or to reduce the search-space in search problems. It also discusses support to Prolog extensions such as delaying of goals and constraints.

Chapter 9 describes the determinacy analyser. This system generates the code which verifies when a goal is determinate. To generate this code, the analyser must analyse the heads of clauses and builtins in the clauses. To verify all cases of determinacy is a hard problem. Some simplifications are necessary. Such simplifications and their motivation are also discussed in this chapter.

Chapter 10 describes the Andorra-I engine and its performance. The chapter gives an overview of the main data-structures designed by Yang to support parallelism in Andorra-I and of the schedulers designed by Yang, Beaumont and Dutra to support parallel execution. Next, the chapter presents an overview of how the Andorra-I abstract machine and compiler was designed, and gives a short study of Andorra-I performance.

Chapter 11 discusses some more powerful models that can improve on the Basic Andorra Model. IDIOM expands the Basic Andorra Model to support independent and-parallelism. The EAM and AKL provide powerful execution models which support non-determinate dependent and-parallelism. Such models lift some of the limitations of the Andorra-I system.

Chapter 12 presents the main conclusions of this work and suggests lines of research for future work.

The thesis includes an appendix containing a more detailed presentation of the operations on the abstract domain presented in chapter 7.

Chapter 2

Logic Programming Languages

The logic programming paradigm uses logical inference on the clausal form of logic to provide a problem-solving mechanism. This mechanism stems from the investigations into the mechanical proof of theorems, such as Robinson's resolution rule [131]. It was found that for an important subset of first order languages, Horn clauses, there is a simple proof procedure, that can be easily implemented as a computer program. As expressed by Kowalski [95], logic programming is thus about expressing problems as logic and using a proof procedure to obtain answers from the logic. Logic programs include two components, the *logic*, usually expressed as Horn clauses, and the *control*, used to guarantee efficient algorithms.

The language Prolog is due to Colmerauer and others [35]. Prolog uses a particularly simple proof procedure, which is very well suited to standard computer architectures. Prolog has been quite successful as a general-purpose programming language. Still, it has some limitations, one of the most serious being that Prolog's execution rule can be too constraining. In fact, it is sometimes the case that a problem can be easily expressed as Horn clauses, but that using Prolog's proof procedure on this program will give a very bad algorithm. The belief that more flexible proof procedures would expand the number of problems that can be solved in a declarative fashion has resulted in several extensions to Prolog and on new logic programming languages having been proposed.

In this chapter, we first give a brief overview of the fundamental concepts in logic programs. We present Prolog and its implementation. We also discuss some of the many alternatives to Prolog, including the committed-choice languages.

2.1 Logic Programs

Logic programs manipulate *terms*. A term is either a logical variable, or a constant, or a compound term. Constants are elementary objects, and include symbols and numbers. Logical variables are terms that can be attributed values or *bindings*. This process is known as *instantiation* or *binding*. Logical variables can be seen as referring to an initially unspecified object. Hence, variables can be given a definite value (or *bound*) only once. Several variables can also be made to share the same value, that is a variable may be instantiated to another variable.

Compound terms are structured data objects. Compound terms comprise a functor (called the *principal functor* of the term) and a sequence of one or more terms, the *arguments*. A functor is characterised by its *name* and by its *arity*, or number of arguments. The form *name/arity* is used to refer to a functor. In the Edinburgh syntax [28], terms are written as $f(T_1, \ldots, T_n)$, where f is the name of the principal functor and the T_i the arguments. A very common term is the compound term $.(Head, Tail)$, written as $[Head|Tail]$, usually called the pair or the list constructor. The Edinburgh syntax also allows some functors to be written as *operators*. For instance, the term `'+'(1,2)` can also be written as `1+2`.

A term is said to be *ground*, or *fully instantiated*, if it does not contain any variables.

We define *size* of a term to be one if the term is a constant or variable, and one plus the size of the arguments if the term is a compound term.

We can now define Horn clauses. Horn clauses are terms of the form:

$$H :- G_1, \ldots, G_n.$$

H is the *head* of the clause and $G_1, \ldots, G_n$ is the *body* of the clause. The body of a clause is a conjunction of *goals*. If the body is *empty*, a clause is named a *unit clause*. Otherwise, a clause is called a *non-unit clause*. The *head* consists of a single goal or is empty. If the head of a clause is empty, the clause is called a *query*.

Goals are terms, either compound terms or constants. Goals are distinguished from other terms only by the context in which they appear in the programs.

Logic programs consist of clauses. A sequence of clauses whose head goals have the same functor, forms a *procedure*. Procedures are formed with unit clauses, or *facts*, and non-unit clauses, or *rules*.

Declarative reading We give a brief definition of declarative semantics of a logic program (see Lloyd [97] for a thorough description). In the declarative reading, each clause in a program specifies a logic relation, where the symbol `:-` represents the logical implication, and the symbol ',' represents the conjunction. Thus, a clause of the form:

$$H :- G_1, \ldots, G_n.$$

can be read as: if there is a set of assignments for the variables appearing in G_1 to G_n such that G_1 to G_n hold, H will hold.

The declarative semantics of a program give the set of all truths that can be deduced from that program. This can be formalised in terms of standard model theoretic semantics of first-order logic. These models try to obtain a minimal model, M_P, that includes all the goals that are a logical consequence of a program P.

Traditionally, Herbrand interpretations [97] are used for this purpose. Given a first order language L, the Herbrand Universe U_L is the set of all ground terms that can be formed out of the constants and functions symbols appearing in L, and the Herbrand base B_L is the set of all ground terms that can be formed by using predicate symbols from L (i.e., the set of ground goals). An Herbrand interpretation I_L is a subset of B_L, such that the goals in the subset give the goals that are true with respect to the interpretation. An Herbrand model for a set of formulas S is an Herbrand interpretation which is also a model for S, that is an interpretation such that all formulae in S are true.

A least Herbrand model for a Horn clause program P can be obtained by intersecting all Herbrand models for P. Van Emden and Kowalski [170] showed that the goals in the resulting least model M_P are *the goals that are the logical consequences of the program.*

The least Herbrand model can also be given in terms of fixed point semantics. First, we define complete lattices. A *complete lattice* is a set S with a partial order relation $\leq$ such that there is a least upper bound, *lub*, and a greatest lower bound, *glb*, for every subset of S.

To use fixed point semantics, we need to associate a complete lattice with any definite (Horn clause) program P. The set of all Herbrand interpretations of P, 2^B_P, is such a complete lattice under the partial order of set inclusion $\in$. The top element of this lattice is B_P and the bottom element is $\perp$.

We next define a mapping, the immediate consequence operator T_P, on Herbrand Interpretations of definite programs. The operator T_P is a mapping $I_P \longrightarrow I_P$ such that $T_P(I) =$

$\{A \in B_p \mid A :- A_1, \ldots, A_n$ is a ground instance of a clause in P, and $\{A_1, \ldots, A_n\} \subseteq I\}$.

The operator T_P is clearly monotonic. Moreover, it can be proven that it is continuous, that is that $T_P(lub(X)) = lub(T_P(X))$. If it is continuous, it has a least fixed point $lfp(T_P)$. This fixed point is obtained by using the theorem that the fixed point of a continuous function T on a complete lattice is the infinite ordinal power of T, $T \uparrow \omega$, where the ordinal of power 0, $T \uparrow 0$ is $\perp$ (or empty), the ordinal of power α, $T \uparrow \alpha$, is $T(T \uparrow (\alpha - 1))$, and ω is the infinite ordinal $\{0, 1, 2, \ldots\}$. Van Emden and Kowalski showed that the fixed point of T_P is equivalent to M_P and therefore that $M_P = lfp(T_P) = T_P \uparrow \omega$ (intuitively this says that the minimal model can be obtained by applying T_P iteratively from the unit clauses until converging).

Operational Reading One advantage of Horn clauses is that several complete, and easy to implement, proof mechanisms are available. Traditionally, the *resolution* rule is used by these mechanisms. Given two clauses, resolution creates a new clause that is obtained by matching a negated goal of a clause to a non-negated goal of another clause. Consider the clauses:

```
G'' :- A,B.

:- G', C.
```

The resolution rule will use unification to match the goal `G` in both clauses to obtain a new clause, in this case `:- A,B,C.`. If variables appear in the clauses, then the resolution process will obtain the most general unifier, *mgu*, for the goals that are being matched. For logic programs, the *mgu* is unique if it exists (if it does not exist, the resolution rule fails).

The resolution rule can be used in a top-down or bottom-up fashion. Top-down systems start from an initial query. This query is matched against a clause for the corresponding predicate, and a new goal is launched according to some *selection function*. For Horn clauses, one useful top-down form of resolution is SLD-resolution (or LUSH resolution [79]). In this method, a query is matched against a clause, and generates a new query (or *resolvent*) built from the remainder of the initial query and the body of the matching goal. This process goes on recursively until either some goal has no matching clause, or until an empty query is generated.

There is a simple and intuitive reading to SLD-resolution. Referring to the previous clauses, `G' :- A, B` can be interpreted as part of the definition of a procedure, and the query `:- G'', C` as a set of goals to execute, or satisfy. SLD-resolution operates by selecting one goal of the query and calling a corresponding procedure. To satisfy this goal, some new goals need to be satisfied, hence the new goals are added to the query. The process is repeated until all the goals have been executed.

SLD-resolution does not specify which goal in the query should be selected. This is the province of the *selection function*. Moreover, several clauses may match a goal, hence there might be several ways to search for a solution. For a particular selection function, an SLD-tree represents all the possible ways to solve a query from a program, that is, the *search-space* of the program. It is important to remark that by changing the selection function, one can change the search space. Consider the small program shown in figure 2.1. The figure shows the search trees corresponding to two different selection functions applied in the execution of the query `:- a(X), b(X)`. The first selects `a(X)` first, and needs to consider the two clauses for `a/1`. The second selects `b(X)` first and hence only has a single matching clause for `a(X)`.

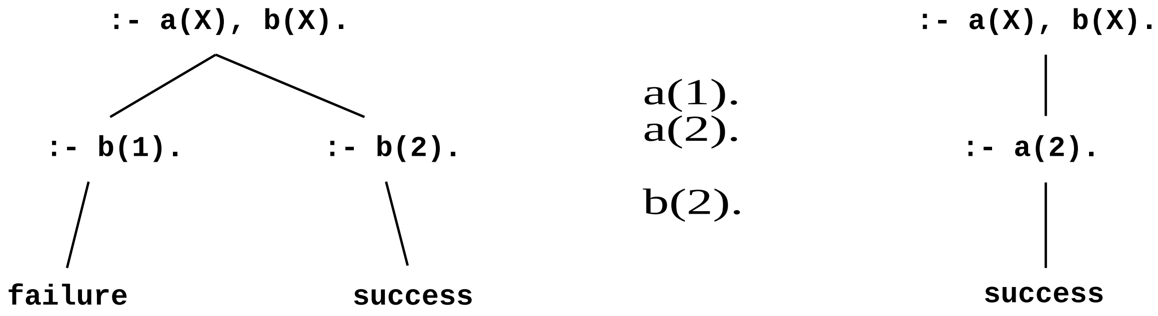

Figure 2.1: Different Search-Trees for the Same Query

There may be several strategies for exploring a search-space. A *search rule* describes which alternative branches should be selected first. Search rules do not affect the search space, but they can affect how quickly one will reach the first solution (if at all).

2.2 The Prolog Language

Prolog was invented in Marseille by Colmerauer and his team [35] (a detailed account of the origins of Prolog is given by Kluźniak and Szpakowicz [93]). Prolog systems apply SLD-resolution, but with some simplifications. Prolog uses a fixed selection function: the leftmost goal is always selected first. The search rule of Prolog is also quite simple: Prolog simply explores the tree in a depth-first left-to-right manner. Whenever several alternatives for a goal are available, Prolog simply tries the first alternative, following the textual order in the program. When an alternative fails, Prolog *backtracks* to the last place with unexplored alternative, (that is, it restores the state of the computation as before that point) and tries the first remaining alternative.

Language Features: In Prolog, programs automatically give control information through the ordering of goals in the body of a clause and of the clauses in the definition of a procedure. The ordering of body goals gives control information for the selection function, whereas the ordering of clauses gives control information for the search rule. To this, Prolog adds extra control operators, and several builtin predicates. Some features may vary slightly for different Prolog systems.

Cut is the control operator most used in Prolog programs. Cut appears as a goal in a clause. When activated, it simply discards all alternatives created since the procedure has been entered.

Prolog includes several other builtins. We next give a brief description (we refer to a Prolog manual or textbook [28, 19] for detailed descriptions).

(i) *Input/Output* builtins allow a Prolog system to interact with its environment. They include the read-in program builtins, such as `consult` and `reconsult`, the term Input and Output builtins, such as `read` and `write`, the character Input and Output builtins, such as `get` and `put`, the stream Input Output builtins, such as `open` and `close`, and the Dec-10 Prolog File operations, such as `tell` and `see`.

(ii) *Internal Database* builtins allow the user to dynamically change the Prolog database. The most important are `assert`, that adds a clause to the Prolog database, and `retract`, that deletes a clause from the Prolog database. The `record` family of predicates add or retract terms from a separate database.

In recent Prolog implementations, `assert` and `retract` only is valid for some predicates, termed the *dynamic predicates.*

(iii) *Arithmetic* builtins perform the arithmetic operations. They include the expression evaluator `is/2` and the arithmetic comparisons.

(iv) *Term Comparison* builtins compare any two Prolog terms. Mostly, they are based on the `compare/3` builtin. The results of comparing two variable depends on the Prolog implementation.

(v) *Control* builtins perform some control operations. They include the if-then-else, `->`, that is a restricted version of cut; not-provable, `\+`, that succeeds if a goal fails; and simple control builtins such as `true`, `false` and `repeat`.

(vi) *Meta-Logical* builtins can test and build new terms. The most well-known are the `var/1` and `atom/1` family of builtins, that test the state of their arguments. The `functor/3`, `arg/3` and `=../2` builtins can be used to create new functors, or to select different arguments from a compound term. The `name/2` family of builtins establishes a relationship between a list of character codes and a symbol.

(vi) *Set* predicates give the several solutions for a goal. Some Prolog systems include `findall/3`, which simply gives the solutions as computed by Prolog, but the most important are the `bagof` and `setof` builtins, that have better semantics [115].

(vii) *Other* builtins are available. Programs need builtins that give program and execution status, or debugging are important. Most Prolog systems also support some interface to other languages, and give a notation for the definite clause grammar formalism.

Note that the use of some of these builtins relies on knowledge of Prolog execution. For instance, in the case of Input/Output goals one will place the goals that write the result after the goals that perform the necessary computation.

The implementation of Prolog: Prolog adapts well to conventional computer architectures. The selection function and search rule are simple operations, and the fact that Prolog only uses terms means that the state of the computation can be coded quite efficiently.

The original Marseille Prolog [7] system was an interpreter. Since then more efficient execution has been possible through refinements in the data structures used to implement the language and through compilation. The DEC-10 Prolog system [177] was the first compiled Prolog system and showed good performance, comparable to the existing Lisp systems [185]. But the basis for most of the current implementations of logic programming languages is the Warren Abstract Machine [178], or *WAM*, an "abstract machine" useful as a target for the compilation of Prolog because it can be implemented very efficiently in most conventional architectures. We give next a brief description of the WAM (we refer the reader to the several tutorials, e.g., [1, 51], for more detailed information).

The main problems that the WAM addresses are the efficient representation of program clauses, control information and representation of data (terms).

As regards data representation, the WAM represents Prolog terms as groups of *cells*, where a cell can be either a value, such as a constant, or a pointer. Variables are represented as a single cells. Free variables are represented as null pointers or as pointers pointing to themselves. Bound variables can simply receive the value they are assigned to, if the value fits the cell size, or made to point to the term they are bound to. The WAM uses a *copy* representation for compound terms. In this representation a compound term is represented as a set of cells, where the first cell represents the main functor, and the other cells represent the arguments. Unification proceeds by first comparing the two main functors and then by being called recursively for every argument. An alternative representation to compound terms is *structure sharing* (structure sharing was used in the original DEC-10 compiler). In structure-sharing terms are represented as pairs, one containing the fixed structure of the term, the *skeleton*, the other containing the free variables of the term. Unification proceeds by comparing the skeletons and assigning variables in the environments. Note that in structure sharing sharing different terms can share the same skeletons whereas in copying each term is independent. Both representations have advantages and disadvantages, but the WAM uses copying for several reasons that include easier compilation and better locality [178].

As regards control, it concerns clause execution, clause selection and backtracking, with clause execution including the two steps of head unification and goal launching.

Clause execution consists of head unification, followed by execution of the goals in the body. Execution of the goals in the body is similar to the execution of a series of subroutine calls in a procedural language, where each goal corresponds to a function call and when the last goal is executed, control returns to the caller as in a procedure return. Thus, subgoals in a clause can be represented as an *environment*. Environments will include control information such as the point where to return to, and the variables that are shared between goals in the clause. An alternative representation, goal-stacking, is to represent each goal individually, by creating a *goal frame* for each goal (goal-stacking was initially considered for Prolog by Warren [179]). Environment representation can be constructed more quickly and more incrementally, and thus is the one used in the WAM.

The environments created during execution form a stack, the *environment stack*, with the last environment to have been created on top of the stack. Note that terms created during the execution of a goal may be passed by the caller. If such terms were stored in the environment, it would not be easy to pop an environment when execution of its clause is completed. The solution is to store compound terms and some variables in a separate data structure, the *heap* (also known as *global stack*). By using the heap it becomes possible to minimise environment size at compile-time.

When a goal fails, Prolog backtracks to the last point where it had alternative clauses. The WAM represents alternative clauses for a goal through the *choice-point* data-structure. This structure stores the states of the stacks and of the caller goal (that is, its arguments). Prolog uses depth-first search, thus the choice-points can also be accommodated in a stack. (The original WAM compacts the choice-point stack and environment stack into a single stack, the *local* stack. For ease of understanding we keep both stacks separate in this presentation).

When executing a goal, Prolog may bind variables of a caller goal. If the variable was created before the last choice-point bindings may need to be undone during backtracking (such bindings are known as *conditional*). Prolog keeps a separate stack, the *trail*, that registers all conditional bindings. During backtracking this stack is consulted and all such modifications undone. Note that a variable is only bound once in Prolog. Thus, the trail needs only to store a pointer to the variable that is bound. When backtracking, this pointer is used to reset the value of a variable to unbound.

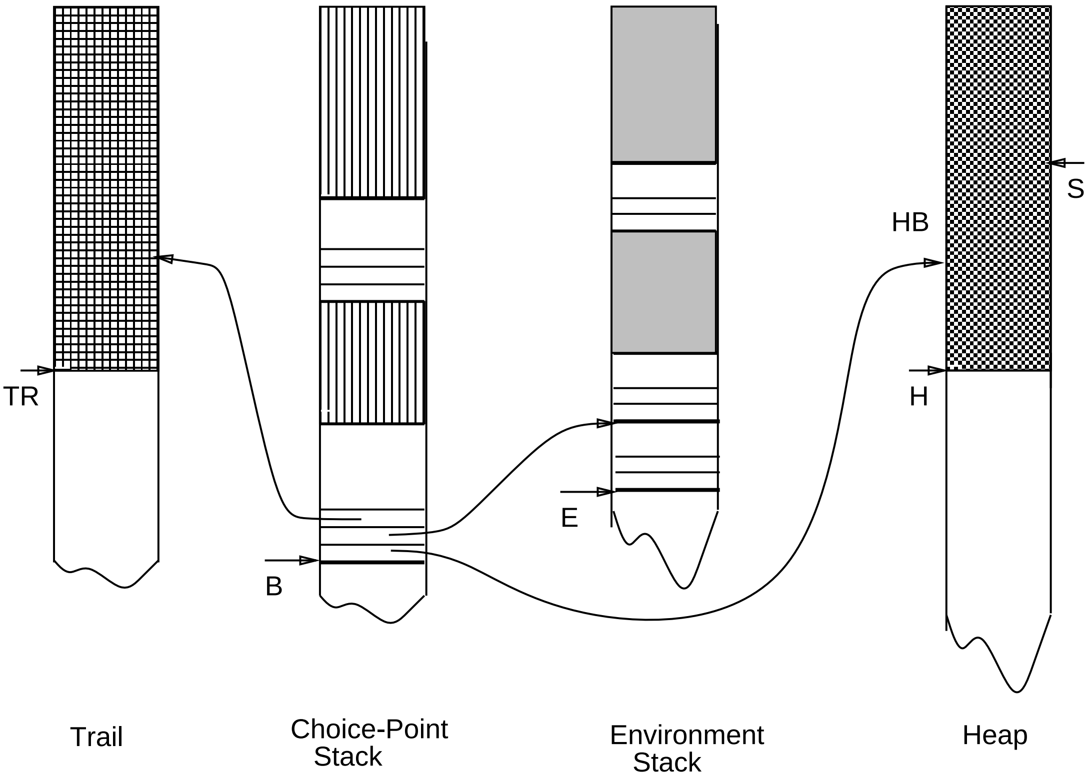

Figure 2.2: WAM Stacks

Figure 2.2 gives an overview of the stacks used by the WAM, with separate local and choice-point stacks. The WAM maintains a set of registers with the current arguments, the `A` registers, the current program pointer, `P`, and the current choice-point, current environment, top of trail and top of heap, `B, E, TR` and `H` respectively. The `HB` and `EB` registers are used when a variable is bound, in order to tell if the variable is older or not than the current choice-point. The `S` register is used when the system is writing or reading a compound term from the heap.

Finally, and as regards efficient program representation, in the WAM programs are coded as instructions of the abstract machine. We give a very brief overview of the WAM instruction set. Instructions can be divided into three groups, the procedural or control instructions, the unification instructions and the indexing instructions. The *control* instructions implement invocation of a new goal (`execute` and `call`) the return to the caller goal (`proceed`) and the management of environments (`allocate` and `deallocate`). Note that failure in the WAM is implicit. The *unification* instructions corresponds to a specialisation of unification for a particular argument or sub-argument. Head unification is specialised into instructions of the type `get`, for arguments, and `unify`, for sub-arguments. Setting up an argument for a goal is specialised into `put` instructions. The *indexing* instructions test arguments to constrain the set of clauses to try. The choice-point manipulation instructions (`try`, `retry` and `trust`) create and manage choice-points.

A WAM-based compiler will generate a set of WAM instructions that are interpreted by an *emulator*. Performance of a WAM-based Prolog thus depends both on the quality of the compiler and on the relative performance of the emulator. Recent efforts on the implementation of Prolog have tried to improve further performance by using direct compilation to native code and by using techniques of global analysis [172, 157]. Native code systems gain performance by by-passing the emulator. They can also perform machine-level optimisations. Global analysis provides information on how arguments are actually used during execution. Its most common uses are in the further specialisation of unification and in more sophisticated indexing.

Although these optimisations generate consistent improvements in performance, they still use the same underlying data-structures and algorithms. Improvements of an order of magnitude have been obtained for some test programs [172]. We would expect this to be an upper bound for such improvements.

2.3 Other Logic Programming Languages

One of the most serious criticisms of Prolog is that the selection function used by Prolog is too restrictive. From the beginning, authors such as Kowalski [95] remarked the effect on the size of the search space of solving different goals in different sequences. For example, several authors showed that often the search space is reduced by selecting the goals for which fewer clauses apply first. They include Kowalski [95], Warren and Pereira in the Chat system [184], and Porto and Pereira in the sidetracking execution principle [127].

A more flexible execution than the one used by Prolog can be obtained through coroutining. *Coroutines* cooperatively produce and consume data [95]. Goals can also cooperate in the same way, as the following example shows.

```
generate_and_test :-
        value( Solution ),
        validate( Solution ).
```

In Prolog, `validate` will wait until `value` fully creates a solution, hence all the alternative values to `Solution` must be tried. If the goals act as coroutines, `validate` can cooperate with `value`. It can fail as soon as a partial value to `Solution` is known never to allow for a solution. Even better, `validate` can itself partially instantiate `Solution`, and further help to reduce the search space.

Since quite early, coroutining has been seen as an alternative to the Prolog execution rule. Languages such as Prolog-II [33], IC-Prolog [26], Epilog [128], MU-Prolog [117] and NU-Prolog [159] were designed to give explicit coroutining.

IC-Prolog was designed by Clark and others [26]. One goal of IC-Prolog was to provide a rich set of control facilities. Two goals may be executed in pseudo-parallel by time-slicing. The pseudo-parallel evaluation may be constrained by only allowing one the goals to bind shared variables. These special annotations allow for data triggered coroutining, where a producer can wait until the consumer suspends. IC-Prolog also provides suspension on variables, that is, a goal can delay until a variable is instantiated. Finally, IC-Prolog introduced *guards* in logic programming. In this language, guards mean that the computation of the first goal cannot be interrupted (thus, even if a guard succeeds, other clauses for the procedure may still be tried).

Epilog [128] allows the user to specify partial ordering of goals. Epilog introduces a "coroutining execution", where reductions for two goals alternate, and "delayed coroutining", where execution proceeds in cycles which should produce and completely consume some instantiation.

The Prolog-II language [33] was designed by Colmerauer and his group. Coroutining is provided in Prolog-II mainly through the `geler` predicate that delays a goal until some variable is instantiated. Also, the `dif` builtin provides a form of inequality that delays until either when the terms are known never to unify, or if the terms are ground. Prolog-II also adds systems of equations and inequations to the Prolog language. These equations are delayed until they can be solved in the most efficient way. Prolog-II is a predecessor of the constraint languages that will be discussed later.

MU-Prolog [117] and its successor, NU-Prolog [159], also attempted to give better control than what is supported by the Prolog language. Both languages use `when` declarations to delay goals until conditions on their arguments are fulfilled. These conditions are either tests on the arguments becoming instantiated, or tests on the arguments becoming ground. The latter are useful to implement negation as failure. Mu-Prolog also introduced several new builtins that, in the style of `dif`, should be less dependent on execution order.

Notice that there has been an evolution in the way coroutining is provided in logic programming languages. This evolution has been from providing coroutining more explicitly to languages where coroutining is expressed more implicitly, that is through data-driven control. The advantages of a more data-driven approach to coroutining are fundamental to the committed-choice languages and in the constraint logic programming languages that are discussed next.

Implementation of Coroutining: coroutining languages such as Epilog or IC-Prolog are quite complex, and thus rather more difficult to implement than Prolog. Pereira suggests the use of Prolog interpreters to implement these languages [125], but in this case the languages can hardly become an alternative to Prolog. Languages such as Prolog-II or NU-Prolog are much closer to Prolog, as they are basically Prolog plus suspension and resumption of goals. These features can be implemented over a conventional Prolog engine in the following way [10, 117, 16]:

- Information on a suspended goal can be stored as a *goal frame*, which will contain the arguments of the goal and control information. Goal frames can be stored either on the heap [16] or in a separate stack [10].

- In both languages a goal suspends until some variables are bound. When these variables are bound, the implementation must quickly find which goals were suspended on them. The simple solution is for these variables to point at the goals frames of the goal suspended on them. This is usually implemented by associating *suspension records* to such variables, that also include some bookkeeping information.

  Bookkeeping information is necessary when a goal suspends on a variable that already has at least one goal suspended on it. Most implementations will create a new suspension record (which will point to the other suspension record) and make the variable point to this new record. Note that when doing so variables lose their single assignment property. Trailing such variables is also more complex, as they may now be assigned several different values during execution.

2.3.1 The Committed-Choice Languages

The committed-choice, or *concurrent*, logic programming languages [144] are a family of logic programming languages that use the process reading [169] of logic programs, instead of Prolog's operational semantics. In this reading, each goal is viewed as a process and computation as a whole as a network of concurrent processes, with interconnections specified by the shared logical variables. The process reading of programs is most useful to build *reactive* systems, which contrast with *transformational* systems in that their purpose is to interact with their environment in some way, and not necessarily to obtain an answer to a problem. Examples of reactive systems are operating systems and database management systems.

Prolog queries can have several solutions, corresponding to selecting different clauses. This has been called *don't-know nondeterminism* to separate it from *don't-care nondeterminism.* In don't-care nondeterminism only a *single* solution is chosen, although this solution can be selected from a set of possible solutions.

The most important characteristics of the committed-choice languages are (a) that they use the process reading of logic programs, (b) that they implement don't-care nondeterminism, but not don't-know nondeterminism, and (c) that goals can only choose a clause, or *commit* to a clause, when certain conditions on the goal are fulfilled. In all the committed-choice languages one can associate these conditions to the state of instantiation the goal, hence these languages naturally offer a form of *data-driven coroutining* where a goal executes only when other goals have instantiated the variables the goal needs to commit.

The conditions for goals to commit to a clause are expressed through *guarded Horn clauses.* These clauses are of the following form:

$$Head :- \ Guard \mid Body.$$

Both the *guard* and the *body* of a clause consist of a (possibly empty) conjunction of goals. If a goal "satisfies" the head and guard of a clause (where the meaning of satisfied varies with each committed-choice language), the goal can *commit* to that clause.

Execution of committed-choice languages starts from an initial query. At each point, the goals in the query test the clauses for the corresponding procedure, and try to commit to a clause. When the goal commits to a clause, the other clauses are discarded, and the goals in the clause's body are added to the query. If the goal cannot commit to a clause, it will wait until its arguments become more instantiated. It is quite reasonable for a committed-choice program to never terminate (an example would be an operating system). Committed-choice programs can *deadlock*, when no clauses can commit, or *fail*, if no clauses match the goal. These two situations are usually treated as errors.

The committed-choice languages evolved from the original efforts towards introducing coroutining and and-parallelism to Prolog, and especially from IC-Prolog. The first proposal of a committed-choice language was Clark and Gregory's "Relational Language" [23], which introduced the commit operator to control don't-care non determinism, but the most well-known committed-choice languages are Concurrent Prolog [141], PARLOG [24] and GHC [165]. Note that all these languages have evolved into several dialects.

Concurrent Prolog was designed by Shapiro [141]. Concurrent Prolog uses *read-only* variables: a goal must suspend until its read-only variables are instantiated. The next example shows a procedure to append the list in the first argument to the list in the second argument in Concurrent Prolog:

```
append([X|Xs], Ys, [X|Zs]) :- |,
       append(Xs?, Ys, Zs).
append([], Ys, Ys) :- |.
```

Notice the read-only annotation for the variable `Xs` in the recursive call. Concurrent Prolog evolved into the CP family of languages [144]. One of the problems these languages attempted to solve is that in Concurrent Prolog several clauses may instantiate variables created by the caller goal before committing, which is quite an expensive operation. Basic CP languages include CP(|) that allows only matching (i.e., no bindings of the caller's variable in the head and the guard), and CP(:) that allows *atomic* unification (atomic because if the unification fails it must be as if it had not been tried). Several combinations of these dialects are possible.

Clark and Gregory's PARLOG [24] language relies on the notion of *safe* guards, where the programmer guarantees that no external variables will be bound. PARLOG classifies the arguments of a procedure as input or output, according to mode declarations provided by the programmer. The execution of a clause will wait until the *input* arguments are sufficiently instantiated, and is not allowed to instantiate its *output* arguments until commitment. The append procedure for PARLOG can be written in the following form (the commits are implicit):

```
mode append(?,?,^).

append([X|Xs], Ys, [X|Zs]) <-
       append(Xs, Ys, Zs).
```

```
append([], Ys, Ys).
```

Ueda's *GHC* [165] (Guarded Horn Clauses) is a language where there are no explicit annotations, but a process will suspend if attempting to bind *any* of the caller's variables during head unification or execution of the guard. GHC's version of `append` is of the form:

```
append([X|Xs], Ys, XZs) :-
        XZs = [X|Zs],
        append(Xs, Ys, Zs).
append([], Ys, Zs) :- Zs = Ys.
```

All the main committed-choice languages have *flat* versions. In flat languages guards consist only of a conjunction of some builtins such as equality, inequality and arithmetic predicates. Flat languages are simpler to implement and understand, but there is some cost in expressiveness and convenience. One example of a flat language is KL1 [154], the language used in the Japanese Fifth Generation Project. The core of KL1 is FGHC, the flat subset of GHC. To this, KL1 adds meta-call predicates to control computations, a pragma language to control parallel execution and some other extensions.

The decision to support a single solution simplifies the design and implementation of these languages. Arguably, don't-care nondeterminism is sufficient for most reactive systems, and indeed the committed-choice languages have been used successfully to implement complex applications such as operating systems kernels or compilers [142]. On the other hand, these languages lack the advantages of don't-know non-determinism. For instance, search programs that can be coded easily and naturally in Prolog are much more awkward to write in these languages [161]. To address this problem, authors have proposed simulating Prolog in the committed-choice language [143], automatically translating Prolog programs to the committed-choice languages [164, 30, 6], or interfacing Prolog to the committed-choice languages [25]. Although useful, such proposals are limited and not as elegant as having a language which would give the advantages of both the committed choice languages and of don't-know nondeterminism. We next mention some proposals: CP[$\downarrow$,|,&,;], P-Prolog, and ANDOR-II.

CP[$\downarrow$,|,&,;]

CP[$\downarrow$,|,&,;] [136] was introduced by Saraswat and is one of the first proposals that introduces don't-know nondeterminacy in the committed choice languages. Saraswat later [137] describes this language as part of his `cc` formalism, where it is seen as a minor syntactic variant of `cc`($\downarrow$,`w`,$\leftarrow$,$\Rightarrow$) over the Herbrand domain, i.e., as **`Herbrand`**($\downarrow$,`w`,$\leftarrow$,$\Rightarrow$) (the arguments to the language represent the control structures of the language). **CP**[↓,|,&,;] includes the control structures Blocking Ask, ↓, that applies to a variable and waits until the variable is instantiated, Atomic Tell (from Concurrent Prolog), don't-care commitment and *don't-know commitment*, &. If a clause with a don't-know commit & is selected for a goal, the system must behave as if the current resolvent has been replaced by two (disjunctive) copies. Both copies can execute in parallel but cannot influence each other. All solutions are accepted.

In order to implement this language, Saraswat designed a translation algorithm from **CP**[↓,|,&,;] to Prolog with `freeze` [137]. Or-fairness (that is a breadth-first like search rule) is not necessary in this language, thus Prolog's sequential search is a sound implementation. **CP**[↓,|,&,;] is a very rich language, and its full parallel implementation will likely be quite complex.

P-Prolog

Yang's P-Prolog [191] extends the committed-choice languages with two fundamental concepts: an *exclusive relation* for clauses and *don't-know* non-determinism.

Guarded Horn clauses are said to be *exclusive* if only one can match the goal. If a set of clauses is not exclusive, all the alternative clauses can be tried.

P-Prolog programmers specify which clauses are mutually exclusive by dividing clauses into subsets, where clauses in different subsets are expected to be exclusive, and clauses in the same subset need not to be exclusive. P-Prolog originally used versions of the `:-` operator to indicate the possible subsets. The `:-` operator indicated a subset with a single element, the `:--` indicated a subset with two elements, and so on. In practice, P-Prolog offers the `:-` and the `:--` operators, where the `:-` operator is used for sets of consecutive exclusive clauses, and the `:--` operators is used for sets of non-exclusive clauses.

The ":" operator is used to separate the guard and the body of a clause. Note that in practice this operator works as a sequential conjunction.

P-Prolog also includes the `other` primitive, useful to obtain don't-care nondeterminism. Merge in P-Prolog can be implemented through this primitive:

```
merge([X|Xs], Y, [X|Zs]) :- merge(Y, Xs, Zs).
merge([], Y, Y).
merge(X, [Y|Ys], [Y|Zs]) :- other: merge(Ys, X, Zs).
merge(X, [], X).
```

The builtin `other` separates clauses into two groups. The second group includes the two last clauses and is only executed if the first group suspends. The `merge` procedure commits when only one clause of a group is a candidate.

P-Prolog allows non-exclusive clauses to be tried as soon as the corresponding procedure is reached. If the procedure is called before all arguments have received their final instantiation, clauses that would have otherwise failed immediately may be tried and generate unnecessary computation. A solution is to use exclusive clauses to wait for some conditions on the arguments, as the next example shows:

```
append([],X,X) :-- X \== [] : true.
append([X|Xs], Y, [Z|Zs]) :-- append(Xs, Y, Zs).
append([],[],[]).
```

The third clause is exclusive with the first two clauses. Non determinate execution of the goal must therefore wait until either the first or the third argument are instantiated.

Yang presented a scheme for the implementation of P-Prolog [189], based on or-trees which are implemented through hash-tables, but in practice only a sequential interpreter for exclusive clauses was implemented. The notion of exclusive clauses in P-Prolog was an important influence in the design of the Andorra model.

ANDOR-II

Takeuchi and others at ICOT proposed an AND- OR- computation model and the ANDOR-II language [152]. As in the programming language Pandora [4] (presented in section 11.1.3), predicates are divided into AND-predicates and OR-predicates. An AND-predicate is defined by a set of guarded clauses and behaves like a normal FGHC predicate. OR-predicates are defined by non-guarded clauses, but a reduction which instantiates the goal during head unification will be suspended until the goal is sufficiently instantiated. AND- and OR-parallel computation starts with a conjunction of atoms, also referred to as a *world*. When a nondeterministic atom is invoked, the world proliferates into several worlds.

The ANDOR-II language uses mode declarations in the style of PARLOG. Mode declarations are needed for both AND- and OR-predicates. In practice, modes are used in the *single-producer constraint* that specifies that a variable is allowed to occur at most once in positions with write mode in its right-hand side. This restriction allows translation from ANDOR-II to GHC. Note that only allowing read-only unification in OR-predicates prevents goals from being called with different modes, which can be quite useful (e.g., the guessing phase typical of Pandora applications).

The AND- OR- parallel computation allows for nondeterminate goals to be reduced eagerly, that is even when determinate goals are available, and is therefore close to **CP**(↓,|,&,;). Although the read-only declarations mean that the system will not be as eager as P-Prolog, eager evaluation of non-determinate goals still means that the search space may be larger than the one generated by selection functions that delay non-determinate goals as much as possible, such as Andorra.

2.3.2 Constraint Languages

Constraint logic programming languages include Prolog-III [34], the CLP framework of languages [83] and CHIP [47]. In these languages, clauses are extended to include a special class of goals, *constraints*. Whereas Prolog variables belong to a single domain, the variables appearing in constraints have values in several sets of domains. These domains include real and rational numbers, intervals, booleans, lists, etc.

Standard goals are executed in the Prolog fashion, but constraints are given to a separate *constraint solver*. The solver maintains a set of active constraints and applies some simplification rules to verify how the constraint can be *satisfied* (a constraint is satisfied if it is true). If a constraint is fully instantiated, a constraint can be simply evaluated and a yes/no answer obtained. Otherwise, the constraint solver can wait until more variables become bound, or it can go ahead and obtain the values of the non-instantiated variables which satisfy the constraints. In this way, a constraint solver can give the effect of coroutining mentioned previously.

Several constraint programming languages have been proposed and implemented. Prolog-III [34] improves Prolog-II (arguably the first constraint logic-programming language). It allows for constraints on lists, rational numbers, and booleans. CLP provides a theoretical framework for constraint logic programming [82]. The framework is quite general and Prolog can be considered as particular case, where the domain are infinite trees, and constraints are of the form $A = B$ (that is, unifications). CLP($\mathcal{R}$) [83] is another instance of CLP, where constraints can be used over the reals (floating-point numbers). CHIP [47] supports finite domains, booleans and linear rationals. CHIP allows user-defined constraints. To introduce a constraint, the programmer must specify when it will be executed (that is, how to coroutine them). To do so, CHIP uses *delay* declarations, similar to NU-Prolog's, *demons*, similar to committed-choice guards, and *conditional propagation*, an if-then-else that delays until the argument to the condition is fully instantiated.

We next show a very simple CLP($\mathcal{R}$) program and query [83]:

```
ohm(V, I, R) :- V = I * R.

?- ohm(V1, I, R1), ohm(V2, I, R2), V = V1 + V2, R1 = 15, R2 = 5.
```

The solution given by CLP($\mathcal{R}$) is:

```
V1 = 0.75*V, I = 0.05*V, V2 = 0.25*V.
```

A weakness of constraint languages is that, for most domains, it is only possible to design a good constraint solver for a limited class of queries. For instance, there is a good constraint solver for linear equations on the reals, but not for all types of equations.

It has been observed that Prolog can be seen as a constraint logic programming language. But the reverse is also true, that is, the domains that are used in constraint-logic programming can be formalised as logic programs. For instance, the *finite domains* of CHIP can be formalised as a finite set of facts defining the possible values for a variable. For such simple domains, the main advantage of constraint languages is that they give an elegant way to specify control.

The implementation of the constraint languages is quite similar to implementations of languages such as NU-Prolog and Prolog-II. The main novelty is the introduction of specialised constraint solvers.

ALPS

ALPS is a class of flat committed-choice languages proposed by Maher [99]. ALPS applies the concepts of constraint logic programming framework to the committed-choice style of execution. ALPS also uses new *commit laws* which have a close relationship to P-Prolog's exclusive relation. A goal A in the presence of a set of constraints C can commit to a rule if:

- that rule is the only one satisfied, or,
- the constraints in the head and guard of the rule are validated (in the case of unification constraints, if the guard does not instantiate any variable in the caller goal)

Maher observes that the two rules are *consistent*: if a rule is validated it is also satisfied, and no other rule can be the only rule satisfied.

2.4 Summary

This chapter presented a survey of work in logic programming languages. We gave a brief description of logic programs, and presented Prolog and its implementation. Traditional Prolog systems always select the leftmost goal for immediate execution. Other languages, such as NU-Prolog, Prolog-II, the committed-choice languages, and the constraint logic programming languages exploit coroutining.

Throughout the chapter we discussed the implementation issues involved with the different logic programming languages. We gave main emphasis to the implementation of the original Prolog language and to the implementation of the committed-choice languages.

Chapter 3

Parallelism in Logic Programs

Parallel computers include several processors that can work in parallel to solve problems. Parallel computers can be organised in very different ways. In *shared-memory machines* the processing units share the same memory space. An example is the basic shared-memory machine, as displayed in figure 3.1(a). In this example the processing units are connected through a bus that allows them to share a memory. This simple scheme is quite efficient, but it has the disadvantage that the bandwidth of the bus limits the number of processors that can work in parallel. Figure 3.1(b) shows an alternative architecture, where processing unit and memory unit pairs communicate in a network. Such architectures can scale to a larger number of processors, but they are unsuitable to the shared-memory model, and are usually programmed in terms of a *distributed-memory* model, where each processor has its own memory and sends messages to communicate with other processors.

Notice that recent work in virtual shared memory machines, such as the DDM [183] or the KSR [9], show the promise of being able to connect a larger number of processors to a *virtually shared memory* in a scalable way.

Forms of Parallelism in Logic Programs Parallelism in logic programs can be exploited implicitly or explicitly. In explicit systems such as Delta Prolog [126] special types of goals (events and splits in Delta Prolog) are available to control parallelism. Unfortunately, these languages do not preserve the declarative view of programs as Horn clauses, and thus lose one of the most important advantages of logic programming.

Implicit parallelism can be obtained through the parallel execution of several resolvents arising from the same query, *or-parallelism*, or through the parallel resolution of several goals, *and-parallelism*. All these forms of parallelism can be explored according to very

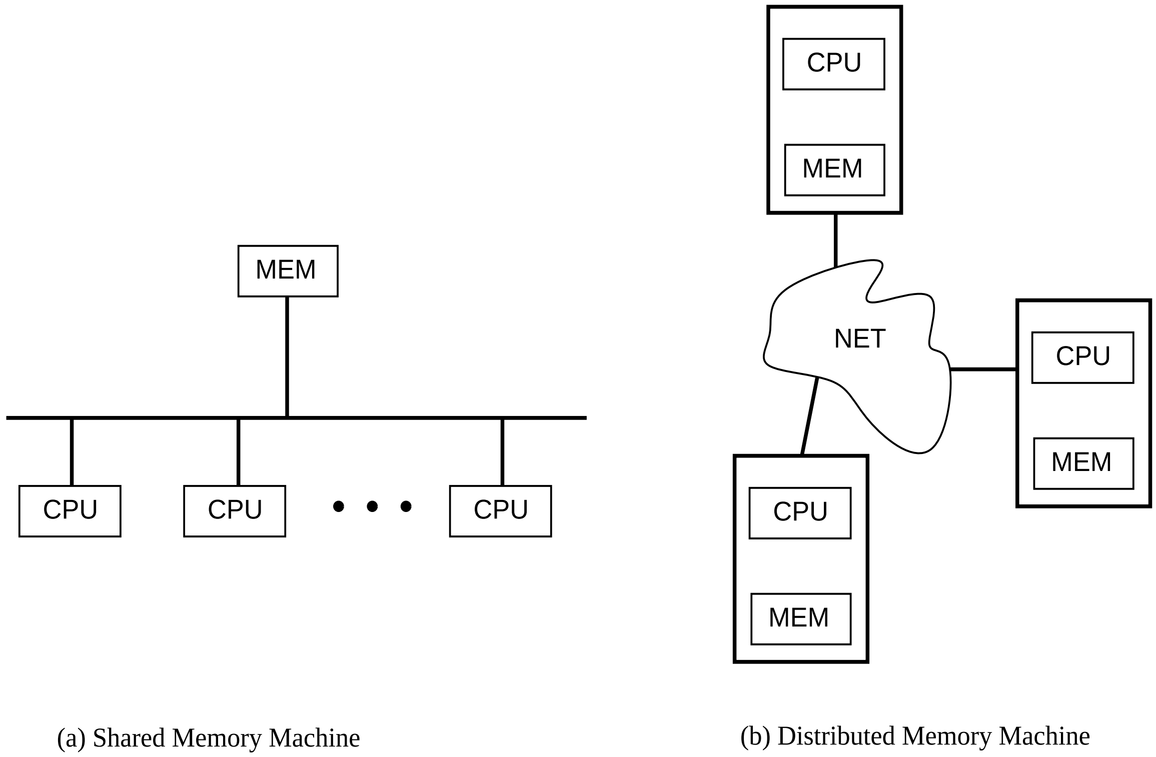

Figure 3.1: Examples of Parallel Machines

different strategies. A full review of the very many models, languages and systems that have been proposed for the exploitation of parallelism in logic programs is a very large task, outside the scope of this thesis. In this thesis we give a brief overview of some systems that we believe to be more important for Andorra-I.

3.1 Or-Parallelism

In or-parallel models of execution, several alternative search branches in a logic program's search tree *can* be tried simultaneously. So far, quite a few models have been proposed. In this section we concentrate on multi-sequential models, where processing agents (workers in the Aurora [98] notation) select or-parallel work and then proceed to execute as normal Prolog engines.

A fundamental problem in the implementation of or-parallel systems is that different or-branches may attribute different bindings to the same variable. In an or-parallel system, and differently from a sequential execution, these bindings must be simultaneously available. The problem is exemplified in Figure 3.2, where choice-points are represented by black circles and branches that are being explored by some worker are represented by arrows. The two branches corresponding to workers W_1 and W_2 see different bindings for the variable X.

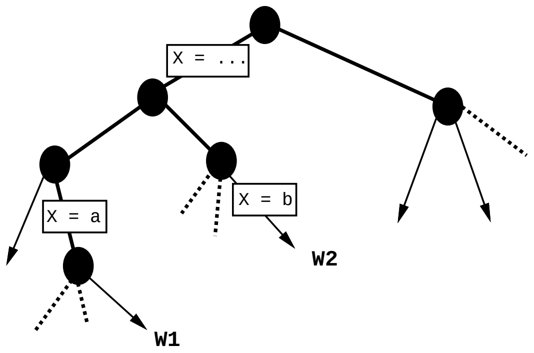

Figure 3.2: The Binding Problem in Or-parallelism

A large number of or-parallel models, including different solutions to these problems have been proposed (the reader is referred to Gupta [63] for a survey of several or-parallel models). The models vary according to the way they address the binding problem. Next, there follows a brief description of some or-parallel models.

Independent Prolog Engines

The binding problem can be avoided by having each worker to operate on its part of the or-tree as independently from other workers as possible. One extreme is represented by the Delphi model [27], each worker receives a set of pre-determined paths in the search term, attributed by *oracles* allocated by a central controller. Whenever a worker must move to an alternative in a different point of the search tree, the worker recomputes all state information for that alternative. Delphi allows for good parallelism, low communication and efficient performance in coarse-grained problems. Delphi can have problems in fine-grained tasks as full recomputation of work may become very expensive. Moreover, Delphi still needs some measure of centralised control to distribute work.

A different alternative, *copying*, is used in other systems such as Ali's Muse system [2]. In copy scheme based implementations, whenever a worker W_1 needs work from a worker W_2, it copies the entire stacks of W_2. Worker W_1 will then work in its tasks independently from other workers until it needs to request more work. To minimise the number of occasions at which copying is needed, scheduling in Muse favours selecting work at the bottom of tree, a strategy which seems to be quite successful at improving task granularity.

Full copy systems basically use the same data-structures as a sequential Prolog engine during ordinary execution. Thus they do not suffer any special overheads during ordinary execution. On the other hand, task switching becomes more expensive. Also, suspension of work in an or-branch, necessary for side-effects and for scheduling when work can be pruned [146], is made more difficult. In general support for builtins that depend on Prolog's left-to-right control, such as `cut`, can be more difficult in systems such as Muse, (and even more for Delphi) as it is necessary to look at the global or-tree to find which worker is leftmost [71].

Shared Space

Instead of each worker having its own stacks, all the workers may share the stacks. In this case they will need to represent the different bindings for the or-branches in some specific way. To do so, some changes must be made to the data structures used to represent Prolog. Whereas sequential implementations of Prolog store bindings in the value-cell representing a variable variable, these systems need to use some intermediate data structure to store bindings to variables that are shared between or-branches. We next discuss two examples of shared space models, the Hash Tables used in PEPSys, and the SRI model used in Aurora.

Hash Tables: The main characteristic of hash-table models [21] is that whenever a worker conditionally bindings a variable, the binding is stored in a data structure associated with the current or-branch (these data structures are implemented as hash-tables for speedy access). Whenever a worker needs to consult the value of a variable, instead of consulting the variable's cell immediately it will look-up the hash-tables first. Figure 3.3 (a) shows the use of hash-tables: note the links between hash windows and the fact that only some hash windows will have bindings. Note also that whenever the value for a variable is consulted, we need only to consult the hash-tables younger than the variable, thus look-up is not necessary for variables created after the last hash-table. PEPSys reduces the overheads in looking up ancestor nodes by adding the binding of a variable to the current hash table whenever that variable is accessed. Analysis of the PEPSys showed that a maximum of 7% of the execution time is being spent in dereferencing through the hash tables.

Binding arrays: In binding arrays each Prolog variable has associated to it a cell in an auxiliary data structure, the *binding array* [186, 180]. In the SRI model, a binding array is associated with each active worker, and every variable is initialised to point to its offset in the corresponding binding-array location. Conditional bindings are stored in the binding arrays and in the trail, but **not** in the environment stack or heap. Unconditional bindings are still stored in the stacks. In this way the model guarantees that if a variable can be bound differently by several or-branches, it must be accessed through the binding array. Moreover, binding arrays have the important property that a variable has the same binding

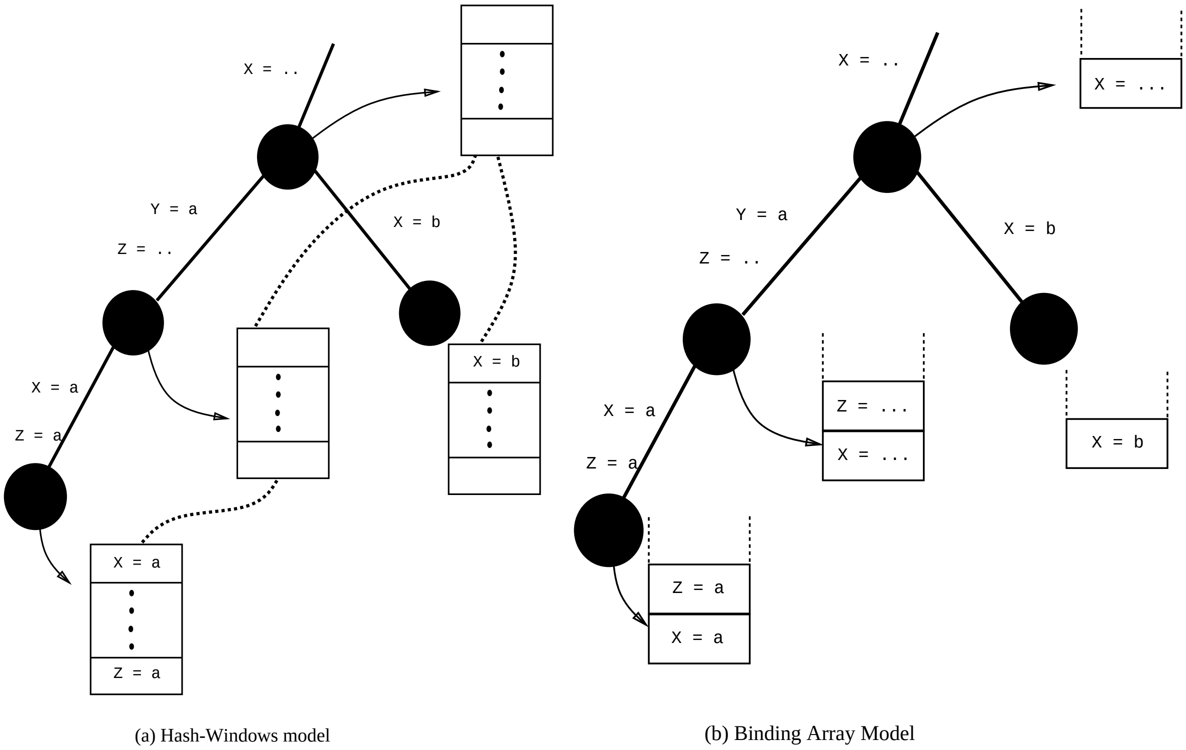

Figure 3.3: Shared Bindings in Or-parallel Models

array offset irrespective of or-branch. Figure 3.3 (b) shows the use of binding arrays: notice that binding arrays always grow when you go down the search tree.

In the SRI model the stacks are completely shared, but binding arrays are *private* to every processor. When a worker W_1 wants to work in a choice-point created by another worker W_2, it backtracks until a choice-point it is sharing with W_2 and then moves down in the or-tree until it finds W_2's choice-point. Backtracking is implemented by inspecting the trail and resetting all entries in the binding array altered since the last choice-point. Moving down the tree is done by setting its pointers to the ones in the choice-point and by inspecting W_2's trail (which is shared) in order to place all the corresponding bindings in the worker's own binding array.

Aurora [98] implements the SRI model. Aurora is a very important influence in the development of the Andorra-I system, and we give a more detailed description of the Aurora implementation next.

Aurora Implementation The Aurora system [98] is based on SICStus Prolog [19]. The SICStus engine incorporates several optimisations to the WAM, particularly several new instructions and the shallow backtracking scheme (discussed in section 9.4.3). SICStus also supports an environment and a choice-point stack. Aurora changes the SICStus engine in several ways [18], we just discuss the most important ones:

- each worker has two binding arrays, one for variables in the environments, the other for variables in the heap;
- choice points are expanded to point to the binding arrays, and to include fields relevant to the management of work in the search-tree;
- memory allocation in Aurora is changed; each worker now represents its stacks as sets of blocks. This allows the stacks to grow without the need for relocation of pointers.

A fundamental problem for or-parallel systems is how to *schedule* or-work. Aurora uses a demand driven approach to scheduling. Basically, a worker executes a task as a Prolog *engine*. When its current task is finished, the worker calls the scheduler which tries to find work somewhere in the search-tree. The interface between the two components has been designed to be as independent as possible of the underlying engine and scheduler [151]. Initial schedulers, such as the Manchester scheduler [15], favoured distribution of work from topmost in the tree. Such strategies do not necessarily obtain the best results, particularly when the or-tree may be pruned by cuts or commits or when most work is fine-grained. The Bristol scheduler [8] was initially implemented to support bottommost dispatching but has been adapted to support several strategies, including selection of leftmost work. The Dharma scheduler [146] favours work which is not likely to be pruned away and tries to avoid *speculative* work. Both the Dharma and Bristol schedulers can use voluntary suspension, i.e., a worker abandoning its unfinished task, to move workers from speculative to non-speculative areas in the or-tree.

Results for Aurora (with the latest schedulers) show good all-solution and first-solution speedups on diverse applications. The static overheads caused by the parallel implementation are on the average of 15%–30%, basically due to supporting the binding arrays and to overheads in the implementation of choice-points. Instrumentation [150] shows that fixed overheads are more substantial than the distance-dependent overheads from moving around the search-tree.

Aurora has a higher sequential execution overhead than a copy based implementation such as Muse. On the other hand, sharing the stacks gives Aurora greater flexibility. For instance, voluntary suspension is harder to implement in a copying based system.

3.2 And-Parallelism

In and-parallel models of logic programming execution, several goals in the query can be active simultaneously. And-parallelism can be exploited in several different ways in logic programs. When exploring and-parallelism for multiprocessor machines it is convenient to separate and-parallelism into *independent* and *dependent* and-parallelism.

Whereas or-parallel computations attempt to obtain different solutions to the same query, and-parallel computations collaborate in obtaining a solutions to a query. Each and-parallel task will contribute to the solution by binding variables to values. Problems can arise if the parallel goals have several different alternative solutions, or if several parallel goals want to give attribute different values to the same value (the *binding conflict* problem).

There are several solutions to these problems. Traditional *independent and-parallel* systems run goals that do not share variables in parallel ((*Non-Strict Independent And-Parallelism* [76] gives a more general definition of independency between goals that allows some variable sharing)). Independent and-goals solutions need to be merged only at the very end of their computation. Independent and-parallel goals may be executed by Prolog engines (one example is the &-Prolog system), or independent and-parallelism may be combined with or-parallelism in some fashion (as, say, the AND/OR process model or the ROLOG system demonstrate).

In contrast, *dependent and-parallel* systems allow goals that share variables to proceed in parallel, while (usually) enforcing some other restrictions. One example of such systems are the parallel implementations of the committed-choice languages. Parallelism in these languages can be exploited quite simply by allowing all goals that can commit to do so simultaneously. By their very nature, committed-choice systems do not have the multiple-solution problem, as they disallow don't-know nondeterminism. In this case the binding conflict disappears, since if two goals in the current query give different values for the same variable, then the query is inconsistent and the entire computation should fail.

We next discuss some examples of and-parallelism in logic programming systems. We discuss and-parallelism in the committed-choice languages, with emphasis on the implementation issues. We mention some independent and-parallel models and systems, and we briefly refer to some proposals to exploit dependent and-parallelism between nondeterminate goals. Exploiting and-parallelism between determinate goals, as performed in the Basic Andorra Model [181] and PNU-Prolog [118] is explained in more detail outside this chapter.

3.2.1 And-Parallelism in the Committed-Choice Languages

In the committed-choice languages, all goals that can commit may run in parallel and generate new goals in parallel. Parallelism in these languages is thus at the *goal-level*. An ideal execution model for such languages could be based on a pool of goals. Whenever a worker is free it looks in the pool of goals and fetches a goal. If the goal does commit to a clause, the worker should add the goals in the body to the pool. Otherwise, the worker would look for another goal.

Consider now a very simple FGHC program:

```
append([X|Xs], Ys, XZs) :-
       XZs = [X|Zs],
       append(Xs, Ys, Zs).
append([], Ys, Zs) :- Zs = Ys.

append4(L1, L2, L3, L4, NL) :-
        append(L1, L2, I1),
        append(L3, L4, I2),
        append(I1, I2, NL).
```

The `append/3` procedure was presented in the previous chapter. The procedure append4/5 appends four lists, by first appending the lists by pairs, and then appending the result.

Consider now the query `append4([1,2],[],[3],[4],L)`. One and-parallel execution with three workers is shown in figure 3.4. The workers executing the left two calls to `append/3` execute independently. The leftmost and rightmost call to `append/3` execute in "pipeline", that is, the leftmost call generates bindings for `L1` which allow the rightmost call to commit.

This very simple example shows the flexibility of the committed-choice languages. Parallelism between independent goals can be exploited naturally. More interestingly, logical variables can be used in quite a natural way to give both sharing and synchronisation between goals one wants to execute in parallel.

Note that verifying if a goal should commit is a very simple process in the flat languages: just performing head unification and some builtins. Thus the parallelism that is exploited in flat committed-choice languages is quite fine-grained.

We next discuss two implementations of the committed-choice languages that have been quite successful for shared-memory machines. Both systems use abstract machines sim-

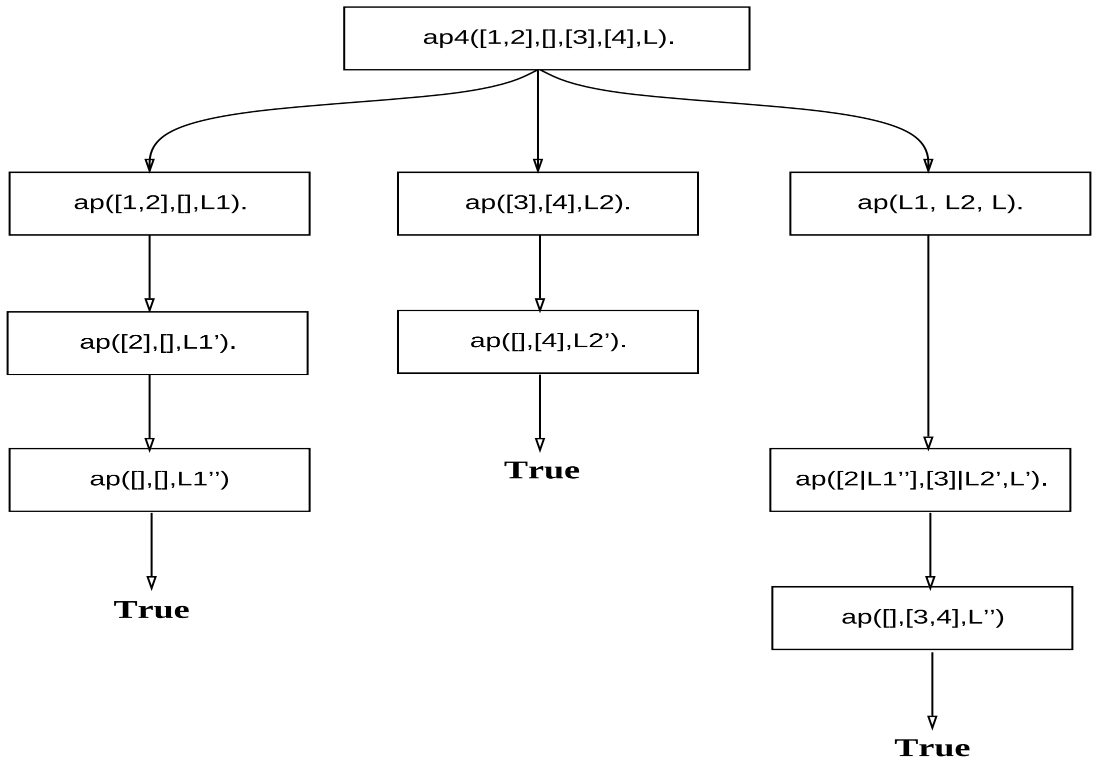

Figure 3.4: Parallel Execution of Multiple Concatenation

ilar to the WAM, but with strong differences on how goals are manipulated. Kimura and Chikayama's *KL1-B* abstract machine [90, 139, 138] implements KL1. Crammond's JAM [41] is an abstract machine for the implementation of parallel PARLOG, including deep guards (although there is no or-parallelism between deep guards) and the sequential conjunction. JAM is based on a light weight process execution model [40].

Both systems use a goal–stacking implementation where for each goal, goal records store all the arguments plus some bookkeeping information, instead of the WAM's environment representation. Goal stacking was first proposed to represent and-parallelism in the RAP-WAM [72], described later. Goal records can be quite heavy, in the JAM they have a total of eleven fields. In the KL1-B goals are stored in a separate heap. In the JAM goal records are divided in a set of arguments, stored in the *argument stack*, and the goal structure, stored in the *process stack*. Both systems store all variables in the heap. Parallel Parlog supports the sequential conjunction that are implemented with a special data structure, *environments*.

We briefly mention some of the more important alterations to the WAM:

- Manipulation of goals: Whereas Prolog can always immediately pick the leftmost goal, in committed choice languages goals can be in several states. The KL1-B classifies these states as *ready*, or available for execution, *suspended*, or waiting for some variable to be instantiated, and *current*, or being executed. The JAM follows similar principles.

- Suspension on variables: Committed-choice languages allow multiple waiting, so a goal may suspend on several variables. The opposite is also true and several goals may suspend on the same variable. Both languages associates to each variable a linked list of *suspension records*, or suspension notes. In the KL1-B each suspension record contains a pointers to the *suspension flag record*, itself consisting of a pointer to the goal record plus the number of variables the goal is suspending. The JAM also uses indirect addressing to guarantee synchronisation between several variables whilst accessing goal records. One useful optimisation of JAM is that goals suspended on a single variable are treated in a simpler way.

- Organisation of clauses: Clauses are divided into a guard, where unification is read-only (*passive* in KL1-B notation) unification and can suspend, and the body where (as in Prolog) unification can bind external variables (*active* in KL1-B's terminology) or be used for *argument preparation*. New instructions are need to support passive unification instructions (these instructions need to consider suspension). Both abstract machines use a special *suspension* instruction that is called when the goal cannot commit.

- Backtracking: In committed-choice languages, there is no true backtracking. Therefore, the trail and choice points can be dispensed with. (In JAM backtracking may occur in the guard but as the goals in the guard cannot bind external variables it is not necessary to implement a trail.) Both abstract machines still include `try` instructions, but they do not manipulate choice-points. The disadvantage of not having backtracking, is that it is impossible to recover space and hence there is a strong need for dynamic memory recovery, such as the recovery of unreferenced data structures through the MRB bit [22] and garbage collection [39, 122].

Shared-memory parallel implementations of both languages have to perform *locking* whenever writing variables because other processors may want to write on the variables simultaneously. To reduce locking, structures are first created and unified to temporary variables; only then they are unified with the actual arguments. Finally scheduling of and-work in these languages is dominated by locality considerations. Each processor has its own work queue, and manages its own parts of the data areas (again, similar ideas were proposed for the RAP-WAM [72]). Depth first scheduling is favoured for efficiency: evaluating the leftmost call first allows better reusage of the goal frames. JAM supports better scheduling [42] by allowing the local run queues to be in part private to each worker. On the

Sequent Symmetry multiprocessor, JAM performs 20% to 40% faster than the corresponding implementation of the KL1-B [162].

3.2.2 Independent And-Parallelism

The literature includes quite a few models that support independent and-parallelism, (often also supporting or-parallelism). Conery's AND/OR process model [36] was influencial in the development of these models. In Conery's model, or-processes are created to solve the different alternative clauses, and and-processes are created to solve the body of a goal. The and-processes start or-processes for the execution of the goals, and join the solutions from the or-processes for the different goals. The model restricts or-parallelism by only starting or-processes for the remaining clauses or if no more or-processes in the current clauses remain to output the solutions. The structure of the model is shown in Figure 3.5, based on Figure 3.7 of [36].

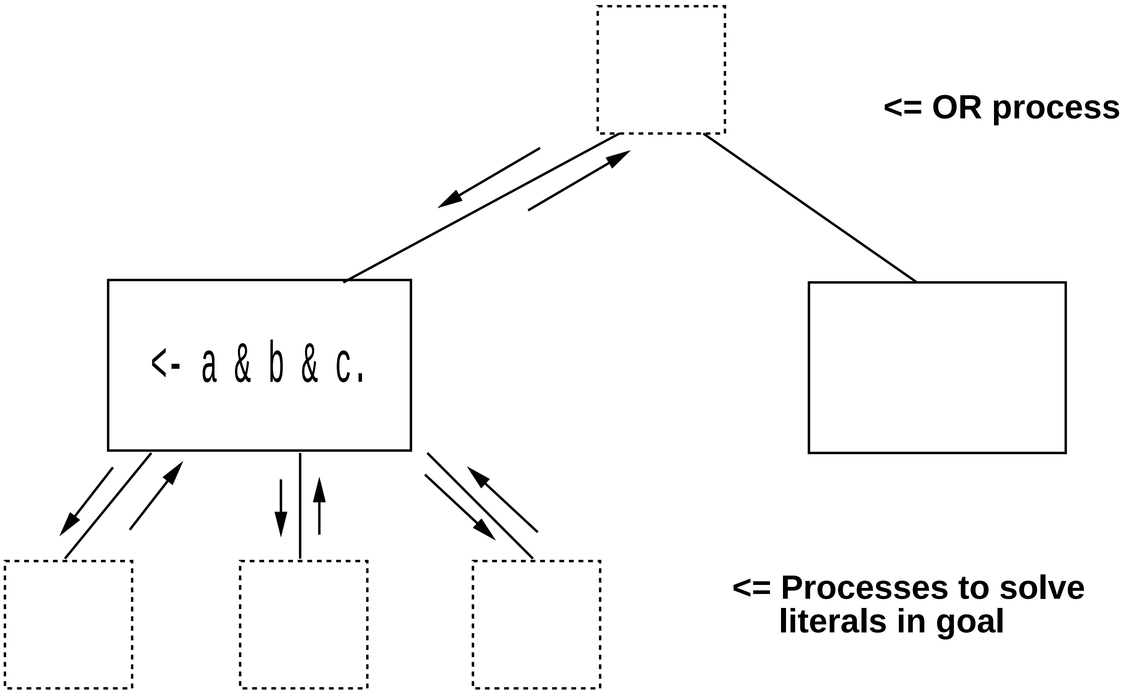

Figure 3.5: Computation in the AND/OR Process Model

In Conery's model a *dependency graph* between goals indicates which goals depend on which goal. The cost of process creation and to maintain the dependency graph means that this model has severe overheads in relation to a sequential Prolog system.

The *REDUCE/OR Process* Model was designed by Kalé [88]. The REDUCE-OR tree is used instead of the AND-OR tree to represent computation. OR nodes correspond to goals, REDUCE nodes correspond to clauses in the program with special notation for generators of variables and several or-nodes for the same goal, corresponding to different bindings of their arguments. REDUCE nodes maintain *partial solution sets, PSS*, initially empty, that are used to avoid recomputation. Sequencing between goals is given by data join graphs in the style of Conery's dependency graphs. Or-parallelism is explored both when a goal is first executed, or when solutions from a goal generate several instances (the latter case is not explored in Conery's model). The ROPM model has been implemented on multiprocessors using structure-sharing to implement a binding environment

that prevents references from a child to its parent node [129]. Benchmark results suggest significant overheads in the implementation of the model, but almost linear speedups in suitable benchmarks due in some cases to AND- and in some cases to OR-parallelism.

The overheads in implementing dependency (or join) graphs may be quite substantial. DeGroot [46] proposed a scheme where only goals which do not have any run-time common variables are allowed to execute in parallel. To verify these conditions, DeGroot suggested the use of expressions that are added to the original clause and at run-time test the arguments of goals to verify independence. DeGroot's work was refined by Hermenegildo into the *CGEs* [72] and &-Prolog's parallel expressions [111]. We next give an example of a linear parallel expression in the &-Prolog language:

```
( ground(X), indep(Z, W) ->
        a(X,Z) & b(X, W) ;
        a(X,Z), b(X, W) )
```

If the first two conditions hold, the two goals `a(X,Y)` and `b(X,W)` are independent and can execute in parallel, otherwise they are to be evaluated sequentially. The `ground` condition guarantees that the shared variable will not contain unbound variables at run-time, and the `indep` variable guarantees that `Z` and `W` do not share run-time variables.

(conditional graph expressions), which are usually expressed in the following form:

$$(\langle condition \rangle \implies goal_1 \ \& \ goal_2 \ \& \ \ldots \ \& \ goal_n)$$

and have the meaning that if $\langle condition \rangle$ is true, goals $goal_1$, …, $goal_n$ are to be evaluated in parallel, otherwise they are to be evaluated sequentially. The $\langle condition \rangle$ is a conjunction of the form $ground([v_1, \ldots, v_n])$, which checks whether the variables $v_1, \ldots, v_n$ are bound to ground terms, or $indep(v_i, v_j)$, which checks whether the set of variables reachable from v_i and v_j are disjoint.

3.2.3 &-Prolog

The &-Prolog system implements independent and-parallelism for Prolog. It is based on an execution scheme proposed by Hermenegildo and Nasr [75] which extends backtracking to cope with independent and-parallel goals. As in Prolog, the scheme recomputes the solutions to independent goals if previous independent goals are nondeterminate.

The &-Prolog language extends Prolog with the parallel conjunction and goal delaying. One objective of corresponding system was to have sequential execution as close to Prolog as possible. To do so, &-Prolog maintains much of the Prolog data structures. &-Prolog programs are executed by a number of PWAMs running in parallel [72]. The instruction set of a PWAM is the SICStus Prolog instruction set, plus instructions that include the CGE tests and instructions for parallel goal execution. Synchronisation between goals is implemented through the *parcall-frames*, data-structures that represent the CGEs and that are used to manage sibling and-goals. The resulting system has very low overheads in relation to the corresponding sequential system, SICStus, and shows good speedups for the selected benchmark programs, including examples of linear speedups.

An essential component of &-Prolog is the &-Prolog compiler. This compiler can use global analysis to generate CGEs. (Notice that for some programs this may still be difficult, and &-Prolog allows hand-written annotations). Abstract interpretation is used to verify conditions such as groundness or independence between variables are always satisfied. If they are, the CGE generator can much simplify the resulting CGEs, and avoid the overheads inherent in performing the CGE tests.

The &-Prolog system was designed to support full Prolog. Similar to or-parallel systems, parallel executions of goals may break the Prolog sequence of side-effects. Several solutions have been proposed for this problem, the one actually used in &-Prolog is simply to sequence computation around side-effects.

Independent and-parallel systems were initially designed to only support and-parallelism between goals that do not share variables. This restriction can be lifted for non-strict and-parallelism, where although several goals can share a variable only one goal can bind it [76].

3.2.4 Independent And/Or models

The PEPSys [21] model and Gupta's Extended And-Or Tree model [62] are two models designed to implement independent and- and or-parallelism in a shared-memory framework. Both use CGEs to implement and-parallelism, and combine or-parallelism with and-parallelism by extending respectively hash tables and binding arrays. The several solutions from the independent and-parallel computations are implemented through a special cross-product node, which can be quite complex to implement. In fact, the PEPSys system only truly implements deterministic and-parallel computations [20].

New proposals for combined and-or parallelism use backtracking to obtain the several and-parallel solutions (thus, and as in &-Prolog, some recomputation is performed) citengc93. Such systems should be easier to implement than PEPSys or the Extend And-Or Tree Model, as they can exploit the technology of &-Prolog and the or-parallel systems, and as they do not need to calculate cross-products. They include Gupta and Hermenegildo's ACE [61], a model that combines Muse and &-Prolog, and Gupta's PBA model [64], a model that combines Aurora and &-Prolog. One other important advantage of these models is that they are quite suitable to the implementation of Full Prolog [65].

3.2.5 Some Dependent And-Parallel Models

Several models that allow dependent and-parallelism between non-determinate goals have been proposed in the literature. In order to solve the binding problems all these models impose some ordering between goals.

In Tebra's optimistic and-parallel model [158], the standard Prolog ordering is used. During normal execution all goals are allowed to go ahead and bind any variables. When binding conflicts arise between two goals, the goal that would have been executed first by Prolog has priority, and the other goal is discarded. In the worst case quite a lot of work can be discarded, and the parallelism can become very speculative. Other optimistic models reduce the amount of discarded work by using other orderings of goals, such as time-stamps [121].

Goals can also be classified according to producer-consumer relationships. In these models, producer goals are allowed to bind variables, and consumer goals wait for these variables. Goals can be classified as producers for a variable statically [31, 147], or dynamically [145]. In Somogyi's system [147], extended mode declarations statically determine producer-consumer relationships. In the Codognets' IBISA scheme [31], it is suggested a system where variables in a clause are marked with read-only annotations in the style of Concurrent Prolog. In Shen's DDAS [145] model, dynamic relationships between producers and consumers are obtained by extending the CGE notation. The extended CGEs now mark some variables as *dependent*, and the system dynamically follows these variables to verify which goals are leftmost for them. If a goal has the leftmost occurrence of a dependent variable it is allowed to bind the variable, but otherwise it delays.

The producer-consumer models become very complex when the producer or consumer have to backtrack. Both Somogyi's scheme and especially IBISA apply ideas of "intelligent backtracking" to reduce the search space. DDAS uses a recomputation scheme that is based on Hermenegildo's recomputation scheme for backtracking [75]. Shen claims that recomputation is simpler to implement and results in an execution closer to sequential Prolog, the target language for DDAS.

The consumer-producer models support independent and-parallelism as a subset. In addition, dependent and-parallelism between deterministic computations (as long as the producer-consumer relations between goals are fixed) can be exploited naturally. Finally, the models offer dependent and-parallelism between non-determinate dependent goals. This is a very interesting advantage of these systems (maybe the main advantage), but not without problems. Models using "intelligent backtracking" may be too complex and may need too much communication between goals. Models using recomputation may do too much speculative computation that may be simply discarded when the producer backtracks. Moreover, fixed consumer-producer relations may not be the best way to extract non-determinate and-parallelism. Results from a parallel implementation of these models should help in clarifying these issues.

3.3 Summary

Parallelism is an important way to speed up execution of logic programs. Two main forms of parallelism appear in logic programs, or-parallelism and and-parallelism. This chapter discussed these two different forms of parallelism, and presented several different models that exploit them, either separately or in combination.

Throughout the chapter we gave emphasis the implementation issues involved with the implementation of parallelism, and particularly to the or-parallel system Aurora, and to the Parallel PARLOG system. These implementations were very influential in the development of the Andorra-I system.

Chapter 4

An Introduction to the Basic Andorra Model and to Andorra-I

The previous chapter described several parallel logic programming models and systems. Particularly successful are the parallel logic programming systems that exploit or-parallelism. These systems have one-processor performance performance close to sequential Prolog systems, obtain good speedups for quite a few applications, and support the full Prolog language.

We also showed that and-parallel systems have evolved quite differently. Committed-choice languages use an elegant and reasonably efficient model of computation, that gives and-parallelism quite naturally, but that sacrifice the ability to search for multiple solutions that is such an important characteristic of Prolog systems. Systems that exploit independent and-parallelism, such as &-Prolog, do obtain good performance and are still compatible with Prolog, but arguably they have less appeal than the flexible and general way in which parallelism is exploited in the committed-choice languages.

In this chapter we present a solution to the problem of supporting and- and or-parallelism for general Horn clauses programs, the Basic Andorra Model, and then present Andorra-I, a parallel logic programming system that implements the Basic Andorra Model and that supports Prolog programs.

4.1 Executing Horn Clause programs in Parallel

Ultimately, the reason for the success of logic programming rests in the elegance and simplicity of SLD-resolution. We would like a parallel system to inherit these two important attributes of SLD-resolution.

At each moment SLD-resolution works on a set of alternatives, or more formally, on a disjunction of resolvents. Each resolvent is a conjunction of goals to solve. We will write a resolvent as $(A_1, \ldots, A_n)$ (we assume every variable to be quantified existentially). Prolog uses depth-first search and a left-to-right selection function, that is, it always selects the leftmost goal from the leftmost resolvent.

We have seen that or-parallel Prolog systems use a very simple generalisation of Prolog's strategy. Instead of working on a single alternative at a time, we can work on several alternatives simultaneously. Note that within each alternative we can still select the leftmost goal, thus many or-parallel systems can be seen as several Prolog systems, each one finding work from a pool of untried alternatives.

We would like the same simplicity for and-parallel systems. That is, we would like to select some goals in parallel, unify them with their matching clause(s), and return a new resolvent, succeed, or fail. Committed-choice systems do show that such and-parallelism exists in logic programs, and that it is a very powerful and flexible forms of parallelism.

Committed-choice goals can execute in parallel, and can communicate in quite sophisticated ways, because *goals always agree on the values of variables.* In other words, goals must commit to a single clause, hence if a goal binds a variable to some value, then this binding is **the** binding for the variable and all other goals can immediately use it to commit. On the other hand, in Prolog programs the same goal may have several alternative bindings to a variable. If goals are not sure on what value they want to give to variables, how can they communicate through them?

Independent and-parallel systems give a simple answer to this dilemma. Only goals that do not communicate can execute in parallel. For instance, an independent and-parallel system would allow the two first calls to `append/3` to execute in parallel, because they do not share variables, but would force the third call to wait for the first two. Such systems can obtain good performance and can support Prolog, but lose important forms of parallelism.

There is an alternative that allows one to exploit dependent and-parallelism in full logic programs. The key observation is that at some time some goals in the resolvent may be *determinate*, that is, may have at most one single matching clause. Any solution must agree with the clause chosen by any such determinate goal, or in other words, it must agree with the bindings we make when selecting a determinate goal. Therefore, while we are executing determinate goals only, we have the same properties that allow us to exploit dependent and-parallelism in the committed-choice languages.

The *Basic Andorra Model* uses this principle to allow both or-parallelism, and and-parallelism between determinate goals. In this model:

- Goals can execute in and-parallel, provided they are *determinate*;
- If no (selectable) goals are determinate, we can select one nondeterminate goal, and explore its alternatives, possibly in or-parallel.

Note that the model clearly separates and-parallelism and or-parallelism. In fact, this separation allows a simple description of the model in terms of rewrite rules on the usual disjunction of alternative resolvents. We call each resolvent R, where each resolvent is a conjunction of goals A (we also use the name D for a determinate goal).

- (Selectable) determinate goals of a goal may be reduced in and-parallel:

$$(A_1, \ldots, D_1, \ldots, D_i, \ldots, A_n) \Rightarrow (A_1, \ldots, B_1, \ldots, B_j, \ldots, B'_1, \ldots, B'_k, \ldots, A_n)$$

- Otherwise, some goal is chosen and the resolvent may fork into a set of resolvents:

$$(A_1, \ldots, D_1, \ldots, A_n) \Rightarrow (B_1, \ldots, B_j, \ldots, A_n) \vee \ldots \vee (B'_1, \ldots, B'_k, \ldots, A_n)$$

  where we chose to select the leftmost goal.

- The previous two rules may apply in parallel (or-parallelism):
  if $R_a \Rightarrow R'_{a1} \vee \ldots \vee R'_{aj}$ and $R_b \Rightarrow R'_{b1} \vee \ldots \vee R'_{bk}$ then

$$\ldots \vee R_a \vee \ldots \vee R_b \vee \ldots \Rightarrow \ldots \vee R'_{a1} \vee \ldots \vee R'_{aj} \vee \ldots \vee R'_{b1} \vee R'_{bk} \vee \ldots$$

Note that the first rule provides for parallelism, in the style of the committed-choice languages, as these have a single resolvent. The second rule provides for search, as in Prolog. The second and the third together are the execution model for an or-parallel Prolog system. By joining the three, we can say that the Basic Andorra Model integrates and-parallelism, in the style of the committed choice languages, with or-parallelism, in the style of Prolog or-parallel systems.

4.1.1 The Structure of the Andorra Computation

The Basic Andorra Model describes where parallelism can be exploited. To actually exploit parallelism, several workers must be simultaneously active in the search tree. Consider any two such workers, W_1 and W_2:

- If W_1 and W_2 are working in and-parallel, that is, if they are reducing determinate goals from the same resolvent, then any binding from W_1 must be observable by W_2, and vice-versa.
- If W_1 and W_2 are working in or-parallel, that is, if they are reducing goals from two different resolvents, then any bindings from W_1 must not be observable by W_2, and vice-versa.

To address this problem, one solution (proposed in the Andorra-I engine) is to structure the workers into *teams*. Workers within a team reduce goals in and-parallel. Teams of workers place themselves at different resolvents, and therefore work in or-parallel.

The organisation into teams closely reflects the nature of the Basic Andorra Model, where several and-parallel reductions proceed in or-parallel. It also simplifies the implementation, first because each worker does not need to scan the computation for a resolvent with available work whenever it reduces a goal, thus increasing granularity; second because workers in the team can easily find out with which other workers they need to communicate, thus simplifying synchronisation and reducing contention; third, because teams behave as purely or-parallel workers, and because workers within a team behave as pure and-parallel workers, thus enabling easy adaption of techniques from or-parallel systems and from parallel committed-choice languages, respectively.

Note that the number and size of teams should reflect both the resources available to the computation, and the amount and kind of parallelism available. In general these variables will vary during execution, and therefore the number and size of teams should also vary during execution.

4.1.2 Selection Functions for the Basic Andorra Model

In the Basic Andorra Model and-parallelism is obtained by running determinate goals in parallel. Which and-parallel goals can be run in parallel depends on which parallel goals we can select, that is, it depends on which selection function we use. In general, one will choose a selection function according to the language one wants to support, but it should be clear that the selection function that best fits the model, and particularly that gives the most and-parallelism, is the one where we allow all determinate goals to be immediately selectable. We call the selection function that follows these principles, the Andorra selection function.

- *Andorra selection function:* select a determinate goal, if one is available.

The Andorra selection function is the preferred selection function for a system implementing the Basic Andorra Model. This is partly because we are allowed to run all determinate goals in parallel, and therefore can obtain maximum and-parallelism, but also because this selection function gives a natural form of *implicit coroutining*, where bindings may passed from right to left during the execution of two goals, as well as left to right as in standard Prolog. Often the search space is reduced, relative to a left-to-right selection function, because execution of determinate goals may make other goals determinate, and so on.

Note that the advantages of selecting determinate goals first have been well known since the early days of logic programming. Ultimately, the idea of selecting determinate goals first is an instance of a more general principle, mentioned in section 2.3, the idea that one can obtain better search by selecting the goals with the fewest alternatives first.

Although we have described the Andorra selection function as a single selection function, in fact we still have a choice on which nondeterminate goal to select first. We can say that the Andorra selection function is not a single solution function, but a family of selection functions. We will use Andorra selection function to refer to any member of this family.

Although selecting determinate goals first has the very important advantages we mentioned, it may create difficulties when we want to support full Prolog. The problem arises in Prolog programs which use builtins that rely on Prolog's traditional left-to-right selection function. Such programs may not work correctly when determinate goals execute before the leftmost goal. A system supporting Prolog must therefore restrict the Andorra selection function somewhat, as we shall see later.

4.2 Andorra-I

We have so far presented the Basic Andorra Model. We will now present an actual implementation of these principles, the Andorra-I system:

- The Andorra-I system is a parallel logic programming system that implements the Basic Andorra Model and that fully supports Prolog.

Andorra-I was designed to take full advantage of the Basic Andorra Model. This means both exploiting parallelism, and exploiting implicit coroutining as much as possible. Through these principles, Andorra-I should fulfil two goals:

- Better performance in current Prolog applications.
- Good performance in problems that have a clear and elegant logic formulation, but where a left-to-right selection function will either result in infinite search, or in a very large search space.

As regards the first goal, we can see Andorra-I as optimising traditional Prolog systems, or or-parallel systems. As regards the second goal, we can see Andorra-I as also extending the scope of traditional Prolog systems. We summarise these two goals by saying that Andorra-I supports an extension of Prolog:

- *Andorra-I Prolog*, the user language for Andorra-I, mainly extends Prolog with the implicit coroutining that results from selecting determinate goals first.

From the point of the view of the user, builtins such as side-effects, meta-predicates, and cuts in Andorra-I Prolog will give exactly the same results as in a traditional Prolog system. The main difference will arise when we consider a conjunction of goals, say `(A,B)`. Whereas traditional Prolog systems would always execute `A` first, and then `B`, Andorra-I may select `B` first, if found to be determinate.

4.2.1 The Basic Andorra Model in Andorra-I

We have presented the main goals of the system. Now, we will discuss how the system can take the best advantage of the model to improve execution of logic programs, and particularly, of Prolog.

The first question is how should one implement the Basic Andorra Model. We first note that the model gives implementations a large degree of freedom:

- the model does not give a "strict" definition of determinate goal;

- the model does not specify which determinate goals are selectable;
- the model does not specify which goal to choose when no selectable determinate goals are available.

Different implementations may take different advantage of this flexibility. For instance, we may concentrate on having minimal overheads by only selecting determinate goals if they can be found very quickly. Or we can concentrate on reducing the search space by always selecting the goal with the least number of alternatives first.

In our case, the goal is to support Prolog correctly, and efficiently. We can make the task of supporting Prolog correctly easier, by selecting the same goals as Prolog whenever we can. For instance, we can select the leftmost goal when no non-determinate goals are available. In practice, this is not sufficient. As we shall see later, we also need a mechanism to prevent early execution of some determinate goals.

For Prolog compatibility to be useful, Andorra-I must also be able to do better than conventional Prolog systems. To understand the problem, first notice that the time T a logic programming system needs to execute a program can be described by the equation:

$$T = NR/(SLIPS * PAR)$$

where NR is the number of reductions, $SLIPS$ are the number of logical inferences per second each worker would perform, and PAR is average amount of parallelism in the program, or parallel speedup. For Andorra-I to perform better than, say Prolog, we need for Andorra-I to have a lower number of inferences, an higher amount of parallelism, or an higher LIPS rate per worker. Clearly, all these vary from program to program, but we can give some general comments on each factor:

- PAR

  The main new advantage, particularly over proven or-parallel systems, is and-parallelism. We would therefore like to run as many determinate goals in parallel as possible.

- NR

  We expect to benefit here, as executing determinate goals first will often, but not always, reduce the search space, and therefore NR. The main example is search programs, where early determinate execution of goals can prune unnecessary alternatives.

- $SLIPS$

> We expect some overheads over the very simple left-to-right selection function. These overheads are partly due to determinacy detection, and partly to extra machinery for parallelism. The overheads of parallelism should be offset by the speedups from parallelism. The overheads in determinacy detection should be less if we only consider the cases of determinacy that can be detected efficiently.

Andorra-I follows these considerations by restricting the Basic Andorra Model in the following way:

- By default, Andorra-I selects the *leftmost* goal if no determinate goals are available.
- A determinate goal is *selectable* provided its early execution is not disallowed by the effect of a sequential conjunction, or by a declaration that the goal can never be determinate.
- Andorra-I only detects *flat determinacy*, that is, the form of determinacy where only head unification and builtins in the body are tested to verify whether a clause matches.

The first constraint makes Andorra-I execution as close as possible to Prolog, without losing and-parallelism. The second constraint is mainly necessary to guarantee correct execution of some Prolog builtins, and it is explained in more detail next. The third constraint follows from our consideration in performance. The actual constructs for disallowing execution of determinate goal and the concept of flat determinacy are discussed in detail next.

Disallowing Early Execution of Goals

There are several ways to disallow early execution of goals. Probably the simplest one is to declare that calls to some procedure may never be considered determinate. In practice, Andorra-I mainly uses a more flexible construct, the sequential conjunction, to disallow early execution of goals:

- A conjunction of goals is declared *sequential*, if the right-side of the conjunction can only execute after the left-side of conjunction completes.

Maximum parallelism and coroutining is obtained in Andorra-I by *only* declaring as sequential the conjunctions that need to execute in this way. The system therefore tries to generate the minimal number of sequential conjunctions that guarantee correct execution of Prolog programs.

Note also that users are allowed to introduce sequential conjunctions. This is justified partly because the user may not want any coroutining for some parts of the program, and partly because sequential conjunctions can sometimes increase the granularity of and-parallel tasks [58], which will reduce overheads in exploiting and-parallelism.

Determinacy in Andorra-I

The question of which goals are determinate is a very fundamental question in the design of Andorra-I. In practice, goals can commit to a clause because of several reasons. Consider a short example:

```
a([], _, _, []).
a([X|Y], P, Ls, [X|NY] ) :- X > P, a(Y, P, NY).
a([X|Y], P, Ls, [X|NY] ) :- X =< P, in(X, Ls), a(Y, P, NY).
a([X|Y], P, Ls, NY ) :- X =< P, notin(X, Ls), a(Y, P, NY).
```

The first argument receives a list of values. The fourth argument returns a list of values, such that they are either larger than the second argument, or that they are members of a list of values given by the third argument. The example is interesting because it gives increasingly more complex examples of determinacy:

- `a([], 5, [2], L).`

  The goal is determinate, and we can find that out just by noticing that only head unification of the first clause can succeed. Determinacy through head unification is the more basic form of determinacy, and clearly should be always supported.

- `a([7], 5, [2], L).`

  The goal is determinate. Although head unification can succeed for three clauses, the call `7 =< 5` should always fail, hence only the second clause can succeed. It is reasonable to expect that a determinacy system will recognise determinacy through such builtins, as they are precisely used quite often to distinguish between different clauses.

- `a([7], 3, [2], L).`

  We expect the goal to be determinate, as only the fourth clause will match, but we at least need to execute `in(7, [2])` to verify this. If executing `in(7, [2])` is fast this would be useful. On the other hand, if `in/2` has a more complex definition we can spend quite some time executing this goal, maybe only to discover that the goal was not determinate.

The two first cases correspond to *flat determinacy*. They are clearly the easiest to recognise, and also the cases where detecting determinacy will incur less overheads. As these cases are also the most common, Andorra-I has been designed to recognise only flat determinacy.

We should remark that flat determinacy in Andorra-I is quite similar to the flat guards in the committed-choice languages. In fact, the same reasons that have made the flat committed-choice languages so important also argue for flat determinacy.

There is one disadvantage to flat determinacy. Sometimes Prolog programmers may write goals that are determinate because of some user defined tests, which may be quite simple. Flat determinacy cannot directly recognise these goals as determinate. This means that in order to extract maximum and-parallelism and coroutining in the current Andorra-I, programmers may need to be aware that Andorra-I only recognises flat determinacy.

4.2.2 The Architecture of Andorra-I

We can now present the architecture of Andorra-I. As most other logic programming systems, Andorra-I includes two main components, the compile-time subsystem, the preprocessor, and the run-time subsystem, the engine and its schedulers.

The structure of the system is shown in figure 4.1. The preprocessor extends traditional compilation techniques to address the new problems of determinacy detection, and of specifying goals that cannot execute early at compile-time. The engine and schedulers run compiled or interpreted programs, and address the complex issues of parallel execution of goals.

The preprocessor and the engine are the main subject of the next few chapters. (An overview of the system is also given in [133, 135, 134]). Meanwhile, we will give a brief description of their operation:

- The sequencer and the abstract interpreter detect at compile-time which Prolog goals cannot be executed early, even when determinate. Their output is the original Andorra-I Prolog program, plus some sequential conjunctions.
- The determinacy analyser and compiler generate abstract code suitable for execution by the emulator in the engine. Their output is therefore Andorra-I abstract machine code.

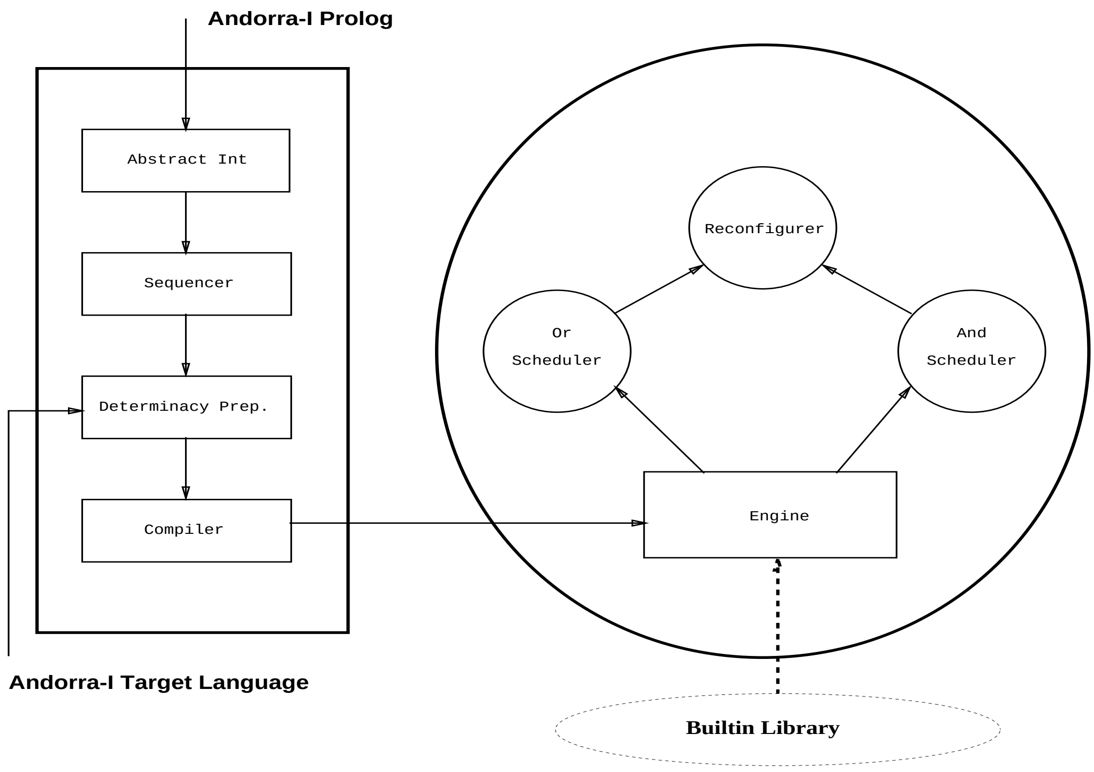

Figure 4.1: The Andorra-I Architecture

- The engine actually executes the program. Workers are grouped onto teams, as discussed before. Each worker can request or-parallel work, and-parallel work, or it can reorganise teams, via the corresponding schedulers.

System components such as the sequencer and abstract interpreter are designed for to support sequential Prolog semantics. Prolog features such as cuts and other builtins also demand support from the compiler, engine, and even scheduler.

Still, one should note that the main effort in the compiler, determinacy analyser, engine and schedulers is concerned with efficient support for coroutining and parallelism. This means that in practice, although directed at Prolog, the Andorra-I system can be easily reconfigured to support other languages that could take advantage of the Basic Andorra Model, such as the committed-choice languages, or even languages designed only to exploit the model. Andorra-I currently provides limited support for such languages by allowing direct access to the compiler and determinacy analyser, through the *Andorra-I Target Language*.

4.3 Summary

The chapter described the Basic Andorra Model, a parallel execution model for logic programs that combines and-parallelism with or-parallelism by executing determinate goals in and-parallel and by extracting or-parallelism from nondeterminate goals.

The chapter also described the first implementation of the model, Andorra-I, and explained the main constraints and characteristics of its design, as a logic programming system that can support Prolog.

Chapter 5

Executing Prolog Programs under the Basic Andorra Model

In this chapter we analyse in detail the problems that arise when executing Prolog programs in Andorra-I. Andorra-I differs from a sequential Prolog system in that it allows or-parallel execution of alternatives, and early execution of determinate goals, possibly in parallel. We present the actual problems in some detail, and show that although some of them can be efficiently solved at run-time, others need support from global analysis of the program at compile-time.

Compile-time analysis is necessary because early execution of goals may prevent correct execution of some builtins. We derive a scheme that uses both compile-time and run-time analysis to guarantee correct and efficient execution of Prolog programs. First we specify the run-time conditions the engine needs to support Prolog. Second, we identify for which calls to builtins we need to disallow early execution of goals. We show that to recognise these goals we need global analysis. Third, we give the conditions that allow early execution of any Prolog goal, including builtins.

We can then present our scheme for the execution of Prolog programs in Andorra-I. The scheme restricts some possible parallelism, but still gives scope for or-parallelism, coroutining and and-parallelism in many Prolog applications.

5.1 Prolog and The Basic Andorra Model

Can we simply feed a Prolog program to a system implementing the Basic Andorra Model? Both the left-to-right and the Andorra selection functions are sound, and should give the same answers to a query, except for infinite loops. But as we have seen, there are important differences in how they generate these answers. The number of steps needed to compute the solutions may be different, and if one is not careful, the interactions and the order between interactions may also be different.

We have seen that Andorra-I reduces these differences by selecting the leftmost goal whenever it cannot select determinate goals. This means that alternatives will be tried in the same order as for Prolog, thus guaranteeing that solutions will come out in the same order. But we still have an important difference. Determinate goals that would only be executed very late by Prolog can be executed immediately by the Basic Andorra Model.

Early evaluation of goals may interfere with builtins. It may affect side-effect builtins, such as `write/1`, or meta-predicates, such as `var/1`. To guarantee correct execution, Andorra-I prevents early execution of some goals (we discuss which next in the chapter). Note that parts of the program that do not call builtins, and which we hope will form the meat of most programs, should still be run according to the model (in general, we call Prolog programs or procedures that do not call builtins *pure*).

This approach should therefore give the most coroutining, and the most parallelism, that the model can correctly exploit from the original Prolog program. The parallelism should be always beneficial, and we claim that in most cases, the coroutining is beneficial. To explain why, consider a simple example:

```
g :- a(X), b(X).

a :- a(1).   b(1).
a :- a(2).
```

Besides parallelism, executing `b/1` first gives two advantages:

- We only have to execute the determinate goal `b/1` once. In contrast, Prolog would select the goal twice.
- We constrain the search space for `a/1`. Prolog would need to try the two clauses for `a/1`, whereas by executing the determinate goal first we can execute `a/1` once.

The two advantages can be very important. They are particularly impressive if the search space grows exponentially, as in these cases we may able to reduce the number of branches to be tried quite dramatically, and also we may also be able to reduce the number of goals to try in each branch.

Do these advantages apply to every program and query? If the computation succeeds, executing a determinate goal early is safe because it can never increase the number of reductions, as we would need to execute the goal at least once. On the other hand, if the computation fails executing determinate goals early may actually delay us from recognising failure. Consider the following example:

```
g :- a, b.

a :- fail1.   b.
a :- fail2.
```

Executing the determinate goal `b/0` early is not worthwhile, because `a/0` will fail regardless. Note that there are examples where although the computation fails and Prolog would never select a determinate goal, we can still benefit by selecting determinate goals first. Consider the next example:

```
g(X) :- a(X), b(X), c(X).

a(1).   b(3).  c(4).
a(2).   b(4).
```

Prolog would fail before calling `c/1`. But calling `c/1` first is still useful, because we can show determinately that we can never find a common value for `X`.

In a perfect world we would have a perfect solution, one which would guarantee that our Andorra execution would never need more steps, or reductions, than Prolog, but that would still allow us to take maximum advantage of the model. Such a solution would only prevent execution of "selfish" determinate goals in branches that fail, that is, it would only disallow early execution of determinate goals that do not help the search space in branches that fail. The problem is that to find out which goals cannot execute early we need first to know which branches will eventually fail, and if we knew which branches fail beforehand, there would be no point in trying such branches at all!

In practice, we have two choices:

- We can try to guarantee better execution than Prolog, by reducing further the amount of coroutining we allow. For instance, we could execute N determinate goals in parallel if, and only if, these were the N leftmost goals.

  Such solutions must be more restrictive than necessary, because otherwise we would know which branches fail. They must also be more complex than simply running the program in the Basic Andorra Model. Hence, for the sake of possibly a few programs, we need to make the model more complex, and less useful.

- We can simply accept the fact that, as most other optimisations, Andorra style execution will work better most of the time, but not always.

The second solution is the most reasonable, and it fits well with our other goal of extracting the most parallelism. Therefore, we will apply it, even if that means taking the risk of performing a larger number of reductions for some programs.

An exception where we would not like to take this risk, is the worst case where the computation should fail, but Andorra-I enters an infinite loop of determinate goals. Haridi and Brand [69] give an example of such a program:

```
?- p(X), loop.

p(X) :- fail1.          loop :- loop.
p(X) :- fail2.
```

Note that the problem of detecting whether the determinate phase will never terminate is simpler than the problem of detecting whether a determinate goal would ever be executed by Prolog. Therefore, in this case, we can obtain some more practical solutions:

1. Haridi and Brand propose a counter that would limit the number of reductions in the determinate phase. This is not a very elegant solution, and the number of acceptable reductions would have to vary from program to program.

2. One can eagerly force nondeterminate reductions as in **CP**[↓,|,&,;] [136], P-Prolog [191] or ANDOR-II [152]. By "eager" execution of nondeterminate reductions, one means forcing a nondeterminate reduction when determinate reductions are still possible. The nondeterminate reduction can be activated as soon as some conditions (such as mode declarations) are fulfilled (the counter method of **(i)** can be seen as a way to force nondeterminate reductions). Associating conditions to nondeterminate goals has several problems. First, it will result in a larger search space when there is a solution. Second, it is difficult to find out which nondeterminate reductions should be allowed to execute eagerly.

3. The most elegant solution is to prevent determinate evaluation of the goals which might lead to the infinite computation (similar principles were used in MU-Prolog [117]). Although it is impossible to determine whether a goal will result in an infinite computation, goals that loop in Andorra but not in Prolog must follow some special conditions:

(i). Firstly, the infinite computation must be caused by a recursive cycle of determinate goals.

(ii). Secondly, the determinate goals in the loop cannot consume recursive data structures created by some goals external to the loop (that is, they cannot consume data structures that are built by some other goals). If they did so, either the data-structure they consume is finite and would eventually be exhausted, or it is infinite, and must be generated by some infinite computation. In the latter case, the infinite computation would in fact be caused by a cycle including the determinate goals that were creating the infinite data structure.

(iii). Thirdly, there needs to be a nondeterminate goal outside the cycle (say, to the left if the system is emulating Prolog).

Global analysis at compile-time can be used to detect goals that can fulfil these conditions. The algorithm would be to (a) detect which goals can be determinate; (b) build the possible determinate recursive cycles; (c) detect which ones could proceed without consuming external data-structures; (d) verify if there are nondeterminate goals to the left of the cycle; and (d) break the cycle by disallowing determinate execution of at least one procedure.

Infinite loops are quite rare and in fact have not been encountered in any of the many Prolog programs run in Andorra-I. Furthermore, detecting possible infinite loops is expensive. A precise algorithm would be very difficult to implement for all cases, whereas a conservative algorithm could lose valuable coroutining and and-parallelism. Instead of devoting resources to a problem that has never been met in practice, we decided not to include analysis of infinite loops in our system.

5.2 Execution of Prolog Builtins

We previously said that correct execution near builtins can be obtained mainly by executing some parts of the program as Prolog would. We have not yet said which parts of the program, nor have we specified exactly which goals cause the problems. We next explain the actual problems in some detail.

5.2.1 Problems With the Early Execution of Builtins

We first consider the problems arising from early execution of some builtins. Clearly, early execution of side-effects in Andorra-I should be disallowed. Consider a procedure `test/2`.

```
test(X, Sol) :- write('Y? '), read(Y), alg1(X, Y, Sol).
```

If we allow early execution of `read(Y)` we can ask for input before we write the prompt. Even worse is the next example:

```
test(X, Sol) :- alg1(X, Sol), write('Sol: '), write(Sol).
```

The builtins assume left-to-right ordering, and executing them before the goal `alg1(X, Sol)` produces incorrect output.

There is a simple solution to this problem. Andorra-I associates each builtin to some specialised determinacy code that tells when the the builtin to execute early. In the case of side-effect builtins this code simply says the builtin can *never* be executed early, that is, that it must wait until becoming leftmost.

It is not always the case that a builtin should wait until leftmost. Consider the following clause, from `fibonacci/2`.

```
fibonacci(N, F) :-
        N > 1,
        N1 is N - 1, N2 is N - 2,
        fibonacci(N1, F1),
        fibonacci(N2, F2),
        F is F1+F2.
```

We expect `N` to be bound when we execute `fibonacci/2`. Clearly, there is no reason why the two first calls to `is/2` should not execute in parallel immediately. Moreover, if we would wait until all calls to `is/2` become leftmost, we would unnecessarily eliminate most and-parallelism from `fibonacci/2`.

5.2.2 Problems With the Early Execution of Goals Near a Builtin

The novel difference between the Andorra-I execution and a traditional sequential Prolog system corresponds to the early execution of determinate goals. Determinate goals can affect builtins if (i) the determinate goal fails before the builtin could execute, or if (ii) the determinate goal binds arguments to a builtin.

We first consider the case where a determinate goal fails. We use as example a procedure that writes all the elements of a list, up to and including the first element that appears only once.

```
write_until_single([G|List]) :-
        write(G),
        in(G, List),
        write_until_single(List).

in(G, [G|L]).
in(G, [OG|L]) :- G \== OG, in(G, L).
```

Suppose the query `write_until_single([1,2,1])`. Its execution is shown in figure 5.1, where oval boxes represent calls to side-effects, and square boxes represent other goals. In a left-to-right execution, two elements of the list are written, and then the goal fails. If determinate goals are allowed to go ahead, the determinate calls `in/2` and `write_until_single` will execute before any calls to `write/1`. But the call `in(2, [1])` will fail before the query calls `write/1`! Hence no side-effects will be generated in an execution that selects determinate goals first, whereas Prolog would execute `write` twice.

In the example, the side-effects pending to the left should have gone ahead even when the goal determinately fails. To address this immediate problem, one could verify if a goal to the left is a side-effect before allowing a determinate goal to fail. This simple solution does not address other programs, as the next example shows:

```
write_pretty_until_single([G|List]) :- write_pretty(G),
        in(G, List),
        write_pretty_until_single(List).
```

The procedure `write_pretty/1` calls `write/1` and is non-determinate. But, in this case the side-effects in `write_pretty` should be finished before `in(2,[1])` is allowed to fail. We

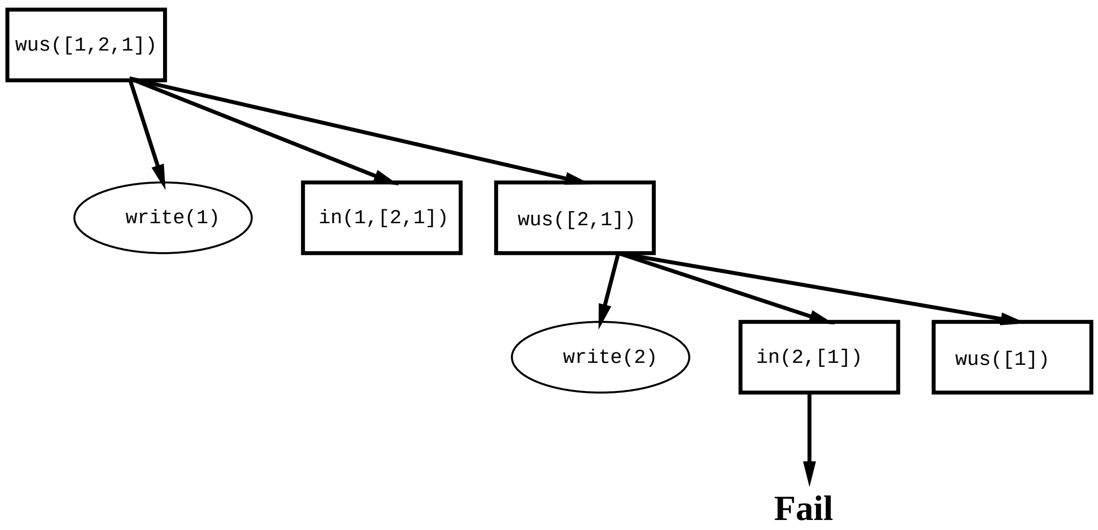

Figure 5.1: And-Execution With Side-Effects

therefore need to recognise whether there are any goals to the left *that call side-effects*. This information can only be obtained by global analysis.

Consider now the problem where a determinate goal binds arguments to a builtin such as a meta-predicate. The following simple procedure is an example:

```
reg_for_perm_var(Reg, Next, Regs) :-
        permanent_var(Reg),
        new_reg(Reg, Next, Regs).

permanent_var(Reg) :-
        var(Reg), !,
        fail.
permanent_var(y(_)).

new_reg(y(R), Next, Regs) :-
        add_to_list(R, Next, Regs).
```

The procedure `reg_for_perm_var` could be part of a Prolog compiler. Variables in a clause are classified as either permanent or temporary. Permanent variables have been found by a previous step and bound to the term `y(_)`. Temporary variables are still unbound. The procedure must find a new register position for permanent variables. It first calls `permanent_var` to verify the variable is permanent. If it is, the procedure `new_reg` obtains a new register for the permanent variable.

Imagine a query `reg_for_perm_var(Reg, 0, [_])`. If executed left-to-right, the query will fail. If the determinate goal `new_reg(Reg, 0, [_])` is executed first, it will bind the variable `Reg` to `y(_)`. Eventually, `permanent_var/1` is called as `permanent_var(y(_))` and succeeds!

The problem here is the builtin `var/1`. The builtin succeeds in the left-to-right call, because the variable has been unbound, but fails if the variable has been bound by early execution of goals to the right. Note that some side-effects also have this problem, for instance in:

```
reg_for_perm_var(Reg, Next, Regs) :-
        write(Reg),
        permanent_var(Reg),
        new_reg(Reg, Next, Regs).
```

The `write/1` goal prints out different values in a left-to-right and determinate-goals-first execution.

Programs that use cut to implement the functionality of `var/1`, may also return different results if some goals are executed early. As a simple example, consider:

```
noisy :- test(X), valid(X).

test(hello) :- !.
test(_).

valid(bye).
```

A left-to-right execution binds `X` to `hello`, prunes the next alternative, and fails when calling `valid/1`. On the other hand, if the determinate goal `valid` is executed first, the first clause for `test` fails, no pruning is performed, the second clause can be taken, and the computation succeeds.

This problem arises because in the left-to-right execution `test` forces a value for `X`. Clearly, if `X` was bound by some other goal, say G, before `test` is called, either the determinate goals and G agree, and `test` can be executed as soon as `X` is bound, or the determinate goals and G disagree, and the computation would fail regardless of which alternative `test` chooses.

We have seen that early execution of goals can interfere with builtins in one of two fails:

- Early failure can prevent side-effects.
- Early bindings affects cuts, meta-predicates and side-effects that assume their arguments to be unbound.

A drastic solution to this problem would be to simply disallow early failure and early bindings in Prolog programs. We believe that such an approach is too restrictive, and propose instead to restrict early execution only on the points of the program where it is really necessary.

The key observation is that only some calls are *sensitive* to early failure or early bindings, namely the calls to procedures with cut that force bindings, calls to side-effect predicates, and calls to certain meta-predicates (such as `var/1`). Our approach is to prevent any execution of goals to the right of such a point until all goals to the left have been completely executed. This is arguably somewhat more than we need. It would be sufficient to allow execution to the right provided early failure and early bindings are either prevented or encapsulated so that they will not affect the execution to the left. However this is more difficult to achieve and arguably not worth the cost.

5.2.3 Problems With the Early Execution of Alternatives Calling Builtins

Finally, we consider the problems arising from the interaction between builtin execution and or-parallelism. We demonstrate the problems through a very simple program:

```
program(X, Sol) :- write('Y? '), read(Y), alg1(X, Y, Sol).
program(X, Sol) :- write('Z? '), read(Z), alg2(X, Z, Sol).
```

The corresponding search-tree is shown in figure 5.2. Imagine now that there are two teams T_1 and T_2, executing the two alternatives at the same speed. If we do not require synchronisation between the branches of the search tree, there are several possible executions, one of them shown in figure 5.3. In this case, the user would be very confused about to which team it was answering!

The problem is a well-known problem in or-parallel systems. The solution is to delay a side-effect goal until all side-effects to the left in the tree have been executed. As in general it is hard to know which goals can call side-effects, most or-parallel systems simply wait until all branches to the left have been finished.

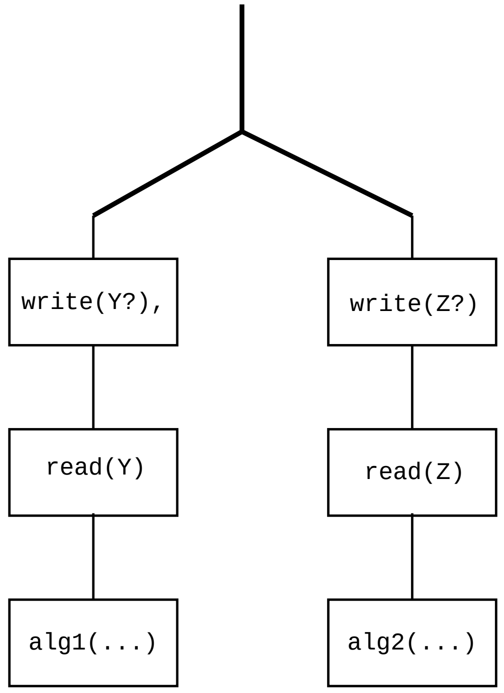

Figure 5.2: A Search Tree With Side-Effects

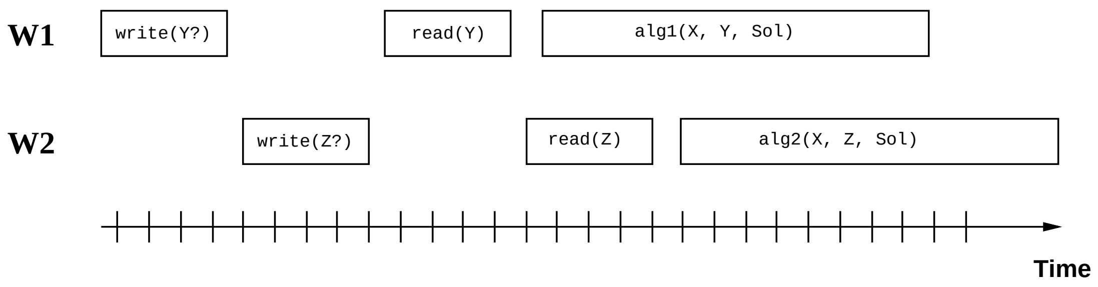

Figure 5.3: Execution of Side-Effects without Synchronisation in the Search Tree

This solution is implemented by Andorra-I without any special compile-time support. Whenever trying to execute a side-effect goal, the Andorra-I engine must first call the or-scheduler to verify if the branch is currently leftmost. If it is, the side-effect can go ahead, but otherwise it will have to wait.

5.3 Engine Support for Builtins

From the previous discussion, it is clear that Andorra-I should include for each builtin:

1. The code that actually implements the builtin.
2. Determinacy code, that says when the builtin can be executed. For builtins such as side-effects, the determinacy code says the builtin can be executed only when leftmost. For other builtins we would like to allow early execution.
3. Code for synchronous execution. This code is necessary for side-effects to verify whether the side-effect is being called in the leftmost branch of the search-tree, and is not necessary otherwise.

This code solves the problems with early execution of side-effects, and the problems with or-parallelism. The problem of sensitive calls needs compile-time analysis.

5.3.1 Andorra-I Execution of Cut and Commit

Cut is a very special builtin. We would expect execution of cut by the Andorra-I to be similar to execution of cut by Prolog. Thus, given a goal A calling the clause $H :- G, !, B$, the execution of the corresponding cut proceeds in two steps:

- after head unification between A and H the current choice-point is stored;
- when all goals in G have executed, all choice-points created after the stored choice-point are removed.

Andorra-I must introduce extra restrictions due to and- and or-parallelism.

- In an or-parallel execution, other teams may be trying alternatives to the left when we reach a cut. A cut can only execute if all cuts that may themselves prune it have been executed. Moreover, the cut should prune all alternatives to the right, but never alternatives to the left [71].

- Andorra-I prevents execution of any goals in B, that is of any goals to the right of the cut, until the cut has been performed.

The first restriction follows the conditions for Aurora. The second condition prevents problems with programs such as:

```
t(X, L) :- member(X, L), !, fail.
t(_, _).
```

and in general follows the idea that it is only worth executing the goals in B after knowing which path will be selected.

Commit is a pruning operator that assumes no ordering in the execution of clauses. Commit is executed in a similar way to cut, but there is a main difference:

- In an or-parallel execution, commit will prune branches that are to its left.

Extra coroutining and and-parallelism can be obtained by using cut and commit to make goals determinate. We discuss the necessary conditions later.

5.4 Sensitive Goals

We have shown that execution of goals to the right of a sensitive call may be dangerous. A goal is sensitive if early execution of goals to its right can result in different results. No goals that call pure Prolog routines can be sensitive. Thus, only the following goals can be sensitive:

- calls to side-effects;
- calls to other builtins, such as meta-predicates;
- calls to procedures that include cuts or commits.

Calls to side-effects are in general sensitive to failure of goals to the left (we call them *fail-sensitive*). As it is in general quite difficult to know if a goal will fail or not, we will always assume they are sensitive. Some calls to side-effects, meta-predicates and goals that use cut are sensitive to early bindings of variables (we call them *var-sensitive*).

If a call to a builtin is *fail-sensitive*, as any call to a side-effects goal is, we cannot allow early execution of goals. We next discuss whether we can do better for var-sensitive only calls to builtins.

5.4.1 Var-Sensitive Calls to Builtins

A call to a builtin is var-sensitive if the builtin does not allow early binding to its arguments. In general, calls to builtin other than side-effects will be either (a) *invalid*, if the builtin cannot be correctly executed, (b) *var-sensitive*, or (c) *var-safe*.

We first give a precise definition of var-safe call. A call for a builtin, B, is var-safe iff executing B first and then binding one of its arguments, say X_i, gives the same solution as binding the argument X_i first and then calling B. In other words:

$$\forall X_i, T \quad B(X_1, \ldots, X_i, \ldots, X_n) \;\; , \;\; X_i = T \quad \rightsquigarrow \quad X_i = T \;\; , \;\; B(X_1, \ldots, X_i, \ldots, X_n)$$

where the operator $\rightsquigarrow$ holds if any solution to its left-side is also a solution to the right-side, up to renaming of variables.

Clearly, var-safe calls allow early bindings to their arguments, and therefore early execution of goals. Therefore, we can allow early execution of determinate goals without any restrictions, as long as we guarantee that in the original execution builtins were only called when var-safe.

Note that pure Prolog goals are always var-safe. They also obey the converse property:

$$\forall X_i, T \quad X_i = T \;\; , \;\; B(X_1, \ldots, X_i, \ldots, X_n) \quad \rightsquigarrow \quad B(X_1, \ldots, X_i, \ldots, X_n) \;\; , \;\; X_i = T$$

Together, these two properties guarantee that we can switch execution of any two pure goals, and therefore that any selection function will return the same results.

In contrast, var-safe calls to builtins do not necessarily obey the second property. In other words, if we remove bindings to a var-safe call, the call may not be var-safe. For instance, the call to `atom/1` in `X = f(_), atom(X)` is var-safe, but the call to `atom/1` in `atom(X), X = f(_)` is not necessarily var-safe. In general, we can switch execution execution of a var-safe call with some other goal only as long as the call is guaranteed var-safe.

For each builtin, we can find a range of calls that will be invalid, var-sensitive, or var-safe depending on whether the value of the call's arguments belong to a certain range. We correspondingly define three domains for the arguments to a builtin. Figure 5.4 shows one example.

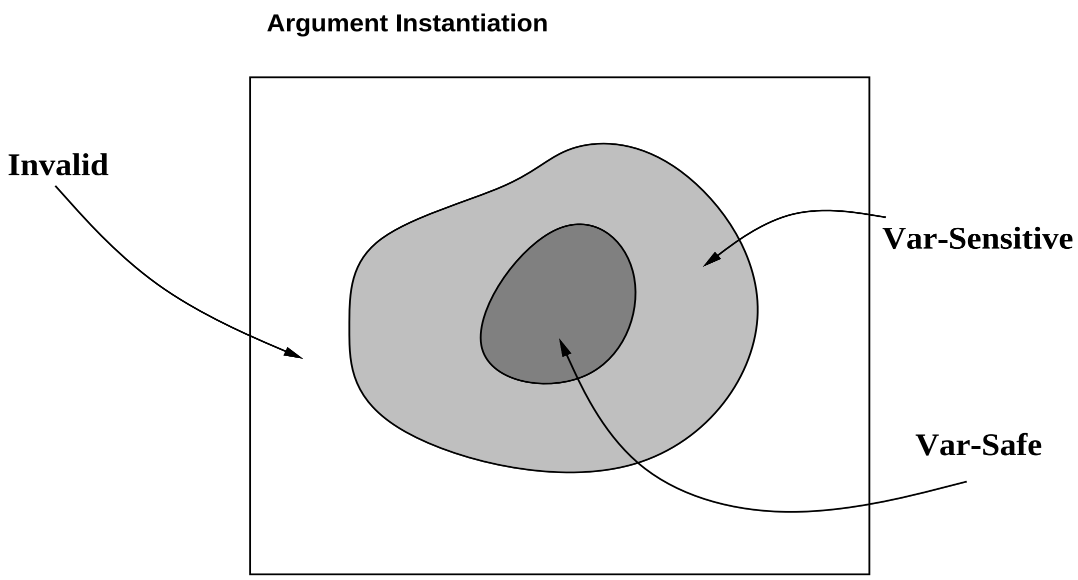

Figure 5.4: Calls to a Builtin

Usually, invalid calls correspond to the builtin not being sufficiently instantiated to execute correctly (for instance, a call `(X is _)` is invalid). More instantiated calls may be executable, but var-sensitive. Finally, if sufficiently instantiated the call is var-safe. Note that calls to a goal may never be invalid. For instance, calls to `var/1` are always executable. Calls to a goal may also never be sensitive. For instance, calls to `is/2` can be invalid, or var-safe. Note also that if all arguments to a builtin are ground, the call could be invalid or var-safe, but never var-sensitive.

We next establish these domains for the most important builtins:

(i) *The Unification*, `X = Y`, is always var-safe.

(ii) *The Arithmetic Builtins* must have the arguments they use to be ground before they can execute. Therefore, calls to these builtins can be either invalid or var-safe.

(iii) *The Term Comparison Builtins* are most often var-sensitive. E.g., `X == a` will fail whereas the more instantiated call `a == a` will succeed. We can be sure a call is var-safe if its arguments are ground, or if the result to the comparison is known by comparing the instantiated part of the arguments. For instance, the call `a(_) == b(_)` is var-safe because we only compare the the main functors of the first argument, but the call `a(X) == a(Y)` is clearly sensitive to any bindings for `X` and `Y`.

(iv) *The Meta-Predicates* such as `var/1` and `atom/1` will be var-safe if their arguments are instantiated. For instance, `atom(a(_))` will give the same results as `atom(a(b(_)))`.

Builtins such as `functor/3, arg/3, =../2` can only be used correctly when they are sufficiently instantiated, and calls to them must be either invalid or var-safe.

(v) *The Meta-Call* is invalid before its argument is bound. As soon as the main functor is known, further instantiation will not affect the execution of the meta-call proper.

(v) *The Set* predicates such as `setof/3` and `bagof/3` are designed to be var-safe. Still, we can have some problems if we allow for existentially quantified variables to be bound. To explain these problems, consider the builtin `findall/3`, used in some Prolog systems. With this builtin, all variables that are unbound at the time of the call are treated as existentially quantified. As a result, `findall` can be used to implement the `var(X)` meta-predicate, as shown by Naish [115]:

```
var(X) :- findall(_, (X = 1 ; X = 2), [_,_]).
```

The problem arises because some existentially quantified variables may or not be unbound when the goal is called. If such variables exist, `findall/3` will be sensitive to them.

Note that the problem does not arise with the free variables of `setof/3` and `bagof/3`. Different bindings to these variables correspond to different solutions, as one would expect for pure goals.

5.4.2 Cut

The problems we face for cut are similar to the problems we face for other builtins. Consider a goal G that calls a cut. We would like for G to obey the property obeyed by a var-safe call to a builtin:

$$\forall X_i, T \quad G(X_1, \ldots, X_i, \ldots, X_n) \ , \ X_i = T \quad \rightsquigarrow \quad X_i = T \ , \ G(X_1, \ldots, X_i, \ldots, X_n)$$

We claim that a goal G is var-safe if its cuts are quiet. We define a cut to be *quiet* if it cannot nondeterminately restrict external variables, that is, if any bindings to external variables that take place before executing the cut are *determinate*. Otherwise, we say the cut is *noisy*.

We next show that goals with quiet cuts are indeed var-safe. First, we show the simpler case where all goals called from G are either pure, or var-safe calls to builtins:

The goal G was pruned by a quiet cut. This cut must force some determinate bindings, say $G\delta$ and a certain path in the search tree for G, say $\mathcal{P}$. By definition of determinate binding, any other more instantiated solution for G, say $G\theta$, must either unify with $G\delta$ or fail. If $G\theta$ and $G\delta$ unify, and all goals called from G are var-safe, the path $\mathcal{P}$ can still be taken for $G\theta$. Further any previous path that failed for G will also fail for $G\theta$, hence the cut must still force $\mathcal{P}$. Therefore, if $G\theta$ and $G\delta$ unify, $G\theta$ must commit to the same path as G. If $G\theta$ and $G\delta$ do not unify, $G\theta$ must always fail.

There are two cases when we move `X = T` to before the goal. First, `X = T` fails when executed after the cut. If we execute `X = T` before G and it also fails, we clearly obtain the same result. If we execute `X = T` before G and `X = T` succeeds, we have already shown we must still choose the same path to prune. The remaining computation is pure Prolog plus var-safe goals, and we should therefore obtain the same results whether we execute `X = T` first.

Consider now that `X = T` succeeds if executed before the cut. Therefore, `X` must still unify with any determinate bindings forced by the cut. Again, when we execute `X = T` first we must choose the same path to prune, and therefore obtain the same results. $\Box$

We use induction to show the case where G contains N goals with cuts:

We have already shown the base case with $N = 0$. Consider now the case of a goal G that contains $I - 1$ calls to goals with quiet cuts. If all cuts are quiet, then all goals called from G must be var-safe, hence from the previous argument G must also be var-safe. $\Box$

Note that we have only shown that we can execute early goals that would be called after G. We can allow early execution of goals to the left of the cut, as we are guaranteed to choose the same solutions, and therefore to prune to the same path. We clearly can also allow early execution of goals to the right of cut, if pruning has already taken place. We cannot allow goals to the right of the cut to execute before the cut, as, if we would, we could prune to a different path, or fail to prune (see the case of the combination cut-fail shown in page 70).

We have shown that we can allow early execution of goals before or during quiet cuts. This means that we can allow early execution of goals in Prolog programs where all cuts were quiet. Coroutining is therefore only dangerous in the presence of procedures with noisy cuts, such as the procedure `test/0` shown in page 66.

5.4.3 Commit

We define quiet commit and noisy commit in a fashion similar to quiet and noisy cut. A quiet commit can only restrict external variables determinately, whereas noisy commit can restrict external variables in an arbitrary fashion.

Although both are pruning operators, their operation is quite different, as commit can pick an arbitrary solution. We use the following example to explain the differences:

```
t(X) :- com(X), X = 2.

com(1) :- |.
com(2) :- |.
```

If `t(X)` is called with `X` unbound, the commit is noisy in the left-to-right execution. In the example, commit could choose either the first or second clause, and the computation could either succeed or fail. In contrast, cut would always choose the first clause. On the other hand, if the determinate goal `X = 2` is allowed to execute first, we can only commit to the second clause.

There is a fundamental difference between noisy cut and noisy commit. This difference is made clearer if we consider early execution of determinate goals, that is of goals that introduce bindings which would always need to be made for the computation to succeed. A noisy cut may prevent us from selecting a path in the search tree that would agree with these bindings. A noisy commit may also prevent us from selecting such a path, *but can also commit to the correct path.* In the example, a noisy cut in `com/1` would force us to select the first clause, when the second clause is the one that agrees. With a noisy commit we could select either the first or the second clause: we may fail, as cut would, but we may also be lucky and hit the right solution.

Consider now a commit that is quiet in the left-to-right execution. Early execution of goals will generate extra bindings. If the commit is quiet, either these bindings agree with the (determinate) bindings necessary to commit, or there is no way to commit. Therefore, if left-to-right execution can commit to a set of clauses, Andorra style execution will either fail, or *can only commit to the same paths in the search tree.*

Early execution of goals before noisy commits is clearly less of a problem than before a noisy cut. Early evaluation of determinate bindings mainly blocks commit from choosing paths that lead to failure. Eventually, some programs that might fail in, say, Aurora may never fail in Andorra-I. On the other hand, most programmers would consider a program that sometimes fails and sometimes succeeds buggy! Still, in the Andorra-I system, we have taken a conservative approach and decided to prevent these differences by treating noisy commits in the same way as noisy cuts.

5.5 Early Execution of Builtins

We have previously shown that there are good reasons why side-effects can only be executed when leftmost, but that by allowing early execution of other builtins we can obtain more coroutining and and-parallelism.

In Andorra-I, a builtin can be executed as soon as:

- the builtin has a single answer, and,
- the builtin is var-safe.

The first condition agrees with our principle of only executing early determinate goals. The second condition guarantees that if we execute the builtin now, we will obtain the same solutions than if we execute the builtin when, say, leftmost.

Most builtins are always determinate, and can therefore be executed as soon as they are var-safe. An exception might be calls to `bagof/3` and `setof/3`, where problems might arise if the callee binds free variables nondeterminately. In fact, to do so the callee needs to be itself nondeterminate, hence it will itself delay, and execution of `bagof/3` or `setof/3` needs to wait until solutions for the callee are found.

5.5.1 Early Execution of Goals With Cut

We can execute a cut as soon as it is quiet. The problem arises with what is the meaning of *determinate execution of cut.* Clearly, by pruning alternatives the cut itself makes a goal determinate, and therefore any quiet cut should be immediately executable.

In fact, Andorra-I uses only flat determinacy, that is, a goal is considered to be determinate if only head unification and builtin execution for at most one clause succeeds. We follow the same principles for cut:

A cut is executable early and makes a goal determinate when (i) it is quiet; (ii) head unification or builtin execution has failed for all previous clauses; and (iii) head unification and builtin execution guarantees this cut will be taken.

The last condition can only be fulfilled if all calls to the left of the cut are calls to builtins.

5.5.2 Early Execution of Goals With Commit

We could go ahead and make any goal with commits automatically determinate. The problem, as we discussed earlier, is that execution of a noisy commit may constrain some bindings which will later on disagree with determinate bindings and thus eventually lead to failure. Pruning while commit is still noisy would thus mean that computations which always succeed with a left to right selection function, could now sometimes fail. We therefore only allow commit to make a goal determinate if the commit is quiet.

As for cut, Andorra-I only supports early execution of flat commits. The rules need not be as restrictive as for cut:

A commit is executable early and makes a goal determinate when (i) it is quiet; and (ii) head unification and builtin execution guarantees commit can succeed.

5.6 A Scheme For Correct Execution of Prolog Programs in Andorra-I

We can now give a scheme for the correct execution of Prolog programs in Andorra-I:

(i) *Compile-Time:* We generate annotations that prevent early execution of goals that interfere with sensitive calls. A simple way to do so is by disallowing execution of any goal to the right of a sensitive call until the call is finished.

(ii) *Run-Time Conditions:* Given the compile time-annotations, we execute calls to pure goals as soon as they are determinate, calls to procedures with cut and commit as soon as they are determinate or as soon as cut and commit can force a single clause quietly, calls to side-effects only when leftmost, and calls to other builtins as soon as they are var-safe and have a single solution.

The scheme assumes that we can detect at compile-time all sensitive calls. This is easy for side-effects, but harder for noisy cuts or noisy builtins, and in general, we may not be sure if a call to a cut or meta-predicate is sensitive or not. Even if we are, the same program call may sometimes be sensitive and sometimes not be sensitive. The following rules apply:

- Compile-Time: if a call may be noisy, then we must assume it is.
- Run-Time: if a supposedly noisy call is found to be var-safe at run-time, we can execute it as any other var-safe call. The argument is that we prevented at compile-time early execution of goals from binding arguments to this goal. Therefore, the call was actually var-safe in a left-to-right execution. In practice, Andorra-I has specialised determinacy code for each builtin, which also verifies a call to a builtin is var-safe.

We should note that the key to our ability to extract coroutining and dependent and-parallelism is that we only sequence for sensitive calls. We would like to remark that the notion of sensitive call is more than a convenience for the execution of Prolog programs in Andorra-I. It agrees with, and reinforces, both good Prolog programming practice, and previous research on the flexible execution of logic programs.

(i) The problems with var-sensitive builtins are well-known. To allow coroutining in MU-Prolog, authors such as Naish [117] define new predicates, similar to `atom/1` or the term comparison builtins, and that are only executed when var-safe.

(ii) Cut is probably the single most controversial feature of Prolog. Still, our work agrees with good programming practice for cuts, mainly with the principle that a cut should be executed as soon as it is determined the current path in the proof tree is correct, and not later, and with the principle that (whenever possible) one should avoid meta-predicates of the form:

```
p(Var, ... ) :- var(Var), !, fail.
p(_, ....).
```

By delaying pruning one can make a cut noisy. By using the `var` builtin one makes the cut noisy.

(iii) Commit has been used in several logic programming languages, particularly (but not only) the committed-choice languages. There is a clear relation between quiet commit and say the safe execution of commit in PARLOG [58]. In PARLOG guard is said to be *safe* if it only tests values of variables obtained by input matching; it is not allowed to bind any input variable. PARLOG verifies guard safety at compile-time (GHC implements a similar system, where the guard will suspend if trying to bind external variables). Safe guards are more restrictive than what is necessary, but they guarantee efficient implementation and precise synchronisation.

Note that quiet commits are more expressive than PARLOG's safe guards in that they allow binding of external variables, as long as that binding is unique. In this sense they are very close to ALPS' commit [99].

5.7 Summary

This chapter presented a scheme for the execution of Prolog programs calls in the Basic Andorra Model. For pure logic programs, both Andorra-I and the left-to-right selection function of Prolog will return the same results, except for the case of infinite computations. The problems with or-parallelism can be addressed at run-time in the same fashion as for Aurora.

The problems with early execution of goals need more care. Sensitive goals are goals that prevent others from executing early. We found that side-effects, meta-predicates, and noisy cuts are sensitive. Finally, we showed that the notion of sensitive goal agrees and generalises previous work on good logic-program design.

Chapter 6

The Sequencer

Given the principles described in the previous chapter, the task of the sequencer is to:

- Detect which goals are sensitive in the Prolog execution.
- Generate annotations to prevent early execution of goals that can interfere with sensitive goals.

The preprocessor considers all calls to side-effects sensitive. In order to detect if a call to a procedure using cut is noisy or if a call to some other builtin is sensitive, the preprocessor must detect whether the call's arguments are sufficiently instantiated. To do so, the preprocessor uses mode information, either obtained from user annotations, or from results from abstract interpretation.

To generate annotations the sequencer must first detect which goals, if evaluated early, will interfere with sensitive calls. The sequencer basically:

1. detects all goals that are ancestors of sensitive goals,
2. and disallows executions of goals to the right of an ancestor

For some programs this is more than necessary, and in practice the preprocessor does allow early execution of some goals to the right.

6.1 Mode Based Analysis of Programs

Prolog programs do not include annotations to tell if cuts are quiet or noisy, or if uses of the meta-calls such as `atom/1` are quiet. The preprocessor must obtain this information either from mode declarations introduced by the programmer, or from the results of abstract interpretation. We next discuss how user declarations and the results from abstract interpretation are used by the preprocessor to detect if a cut may be noisy.

6.1.1 Mode Declarations Based Analysis of Cuts

Mode declarations tell if the arguments to an Andorra-I Prolog procedure are input, output, or don't-know. The mode declarations are:

+ represents a read-only, or *input* argument. A user can declare an argument to be read-only if the argument will not be bound or its instantiation tested by execution of the call.

- represents a write-only, or *output*, argument. The value is not intended to affect the execution of the procedure. Unification against this argument may be postponed until after execution of the procedure.

? represents a read-write, or *don't-know*, argument. No information is provided about the state of the argument or how the argument will be used in the call. This is the default mode.

As an example, consider the following definition for `delete/3`:

```
:- delete(+,+,-).
delete(X, [X|L], L).
delete(X, [Hd|L], [Hd|NL]) :- delete(X, L, NL).
```

The Prolog query `delete(a, [a,b,c], L)` is a valid query. On the other hand, the queries `delete(X, [a,b,c], L)` and `delete(a(X), [a(1),a(2)], X)` do not obey the mode declarations because the first argument will be bound during execution of `delete/3`.

As regards output mode declarations, the query `delete(a, [b,c], b)` does not respect the output mode declaration, although calling the procedure with the third argument bound will give the same results. On the contrary, for the procedure:

```
:- max(+,+,-).
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

allowing early bindings for the third argument can result in an incorrect answer, as `max(4,3,3)` shows.

Prolog mode declarations only affect efficiency for pure Prolog procedures, but they can affect correctness for procedures with builtins or cuts. We next discuss how early execution of goals affects mode declarations.

`+` arguments: we can guarantee that an argument will be as instantiated for Andorra-I as it would be for a left-to-right execution if we only execute the procedure when leftmost.

`-` arguments: early instantiation of goals means that arguments that would be unbound in Prolog may be bound in Andorra-I. Still, execution will return the same results, except if we have sensitive calls in the procedure. In this case, early bindings to output arguments may result in incorrect execution. Therefore, the preprocessor delays unification with output arguments in a procedure until after any cut or sensitive builtin in the procedure.

6.1.2 Mode Based Analysis of Cuts

Cuts in a procedure where all arguments are input or output *must* be quiet, as they cannot force bindings to the input or the output arguments. If some arguments are don't-know, we need to check if unification can constrain variables appearing in these arguments.

We need therefore to verify if either head unification or execution of goals in the body can bind variables.

The algorithm in figure 6.1 succeeds if head unification cannot constrain external variables, and fails otherwise. The algorithm uses two sets of variables, VI, describing which variables appear only in input arguments, and VD, describing variables which cannot be bound, and two auxiliary functions: $vars(T)$ gives the set of variables appearing in the term T, and $nonvar(T)$ tells if the term T is instantiated. The algorithm verifies if head unification may instantiate a don't know argument directly, via unification of that argument to a non-variable term, or indirectly, via variable to variable unification with some other argument. Output arguments are ignored. Notice that all these conditions can be simplified to ignoring output unification and verifying if the don't know arguments are singleton variables.

$VI \leftarrow \emptyset$
$VD \leftarrow \emptyset$
for each argument a_i **do**
    $Vs \leftarrow vars(a_i)$;
    if mode(a_i) $=$ $'+'$ **then begin**
        if $VD \cap Vs \neq \emptyset$ FAIL;
        else $VI \leftarrow VI \cup Vs$;
    end;
    if mode(a_i) $=$ $'?'$ **then begin**
        if $nonvar(a_i)$ FAIL;
        else if $Vs \cap (VI \cup VD) \neq \emptyset$ FAIL;
        else $VD \leftarrow VD \cup Vs$;
    end
end;

Figure 6.1: Read-Only Head Unification

The second step is to verify if goals in the guard of the cut can bind external variables. The sets VD and VI from the previous algorithm are used here to refer respectively to variables which cannot be safely instantiated and to variables which may be. The algorithm must verify for every subgoal before the cut if any external non-instantiated variable, which may only come from VD, can be instantiated. Figure 6.2 describes the algorithm.

for each call C **do**
    $Vs \leftarrow vars(C)$;
    if $Vs \cap VD \neq \emptyset$ FAIL;
    else $VI \leftarrow VI \cup Vs$;
end;

Figure 6.2: Read-Only Calls

Note that the algorithm follows the Prolog execution pattern. Consider a general call C. First, the algorithm verifies if any variable from VD appears in C. If so, the variable may be instantiated and the algorithm fails. Otherwise, the algorithm had to consider the case of new variables created in C: they can only share with variables from VI, which we know cannot be bound, so they can themselves be safely added to VI. This algorithm proceeds until no more calls are left in the guard of the cut.

Mode information for a call C can improve the algorithm. In particular, variables in input arguments, even if in VD, cannot receive extra bindings during execution of the call. Note that one must be careful about C's newly created variables, as they can share with one of VD's variable appearing as an input argument, and therefore they must be included in VD.

Determinate Bindings

Note that even if a pruning operator binds external variables, the pruning operator can be quiet as long as all the different alternatives for the goal give the same bindings to the external variables. In general, to verify this case demands even more precise information than for finding whether external variables are to be bound, and therefore cannot be verified by the preprocessor. The main exception is for arguments that take the same value in all heads of a procedure. They will always result in the same bindings and can never make the pruning operator noisy.

Builtins in the Scope of a Cut

We can say that a sensitive call to a builtin makes the cut noisy. For instance, consider the procedure:

```
a(X, Y) :- var(X), !, Y = var.
a(X, X).
```

We can say that the call to `var/1` is sensitive, and that it makes the cut noisy (because the cut prunes on `X` being unbound). To verify if a call to a builtin is noisy we check all variables appearing in its arguments:

- If the variable appears in an input argument, it must be bound and cannot make the call sensitive
- If the variable is always unbound, we can check for which argument it appears and if the argument is tested by the builtin.
- Otherwise, the call may be sensitive.

Builtins can also be used to test the instantiation of variables. For instance, if the builtin is of the form `X is Exp`, we add rules to the effect that the variables in `Exp` must be ground before the call, and that the variables in `X` will be ground after the call. Therefore, all these variables can be safely included in VI.

6.1.3 Sensitive Calls to Builtins

The general rule for calls to builtins correspond to the rules for builtin discussed previously. We therefore need to apply the algorithm in Figure 6.2 until we reach the call to the builtin, and then verify if the call may be noisy, using the algorithm presented above.

As explained before, we can use information on some builtins to obtain better performance.

6.2 Analysis Based on Abstract Interpretation

The sequencer can use both mode declarations from the user and mode patterns from the abstract interpretation together to find sensitive goals. The sequencer first tries analysis based on the results from abstract interpretation. If this analysis fails, the mode declarations are used. We next describe how the mode patterns from abstract interpretation are used to detect quiet cuts and to say if a call to a builtin can be sensitive.

6.2.1 Analysis of Cuts

The analysis of cuts based on abstract information uses the same basic algorithms as for user mode declarations but takes advantage of the more detailed mode patterns that can result from abstract interpretation. Abstract interpretation obtains the different call patterns for the various calls to the same procedure, plus the input and output Prolog call patterns for every goal in the guard of the pruning operator.

The information provided by the abstract domain is described in the next chapter. For this purpose, it is sufficient to say that the instantiation of an argument may be a variable Var, or an instantiated term with a main functor and possibly some instantiated arguments, or a term $\top(L)$ which can take any value, that is, be either instantiated or a free variable.

Output Arguments The first step is to check which arguments will never affect execution. This occurs if unification for an argument always succeeds, and if the argument cannot be tested by any goals in the guard. The latter holds true if in all calls to the procedure the argument is instantiated to a *single occurrence of an unbound variable.* The sequencer verifies this condition, but in some cases it can also do better, as the next examples based in the use of difference lists shows:

```
process_list([], End-End).
process_list([Hd|Tail], Begin-End) :-
        process_hd(Hd, Begin-Middle), !,
        process_tail(Tail, Middle-End).
process_list([_|Tail], Begin-End) :-
        process_tail(Tail, Begin-End).
```

In this case, if it is known that the call patterns for the second argument are either of the form $\top(\_) - \top(\_)$, or of the form Var, unification to the second argument is always known to succeed and the argument is output. The sequencer thus includes a step which, for every argument, basically verifies if unification with that argument is known to always succeed. If so, the argument is declared output.

Head Unification After processing the output modes, each call pattern is compared with the heads of clauses containing cuts in order to verify if the clause can bind external variables. The algorithm is a specialisation of the abstract unification algorithm used by the abstract interpreter and at each point it tests for one of three conditions: (a) the pattern is not unifiable to the head of the clause, (b) the pattern unifies with the head of the clause but it cannot bind external variables, or (c) an external variable may be bound. Only in case (c) should the algorithm fail.

An external variable may be bound either if (i) a term of the form $\top(L)$ in the calling pattern is unified with a non variable term or a non-singleton variable, or (ii) if a term of the form Var in the calling term is unified to a non variable term or a non-singleton variable that does not match Var, i.e., the head variable should only appear in places where Var appears.

6.2.2 Calls to Builtins

For calls to builtins, the sequencer can use the input and output patterns provided by abstract interpretation. The algorithm is basically the one presented for user mode declarations. The main difference is that one may, by comparing input and output patterns, verify if an external variable has been bound during execution of the goal. Thus some of the optimisations that were before used for builtins become unnecessary as they are now provided for free. Finally, information about sharing of variables can be used to restrict the number of variables to add to VD in each step.

6.3 Indirectly Sensitive Calls

After detecting all sensitive calls, the sequencer must find out for which calls early execution of right-siblings may be dangerous, the *indirectly sensitive* calls.

The set of all sensitive and indirectly sensitive calls can be defined as the fixed point of an operator $S_P(I) = \{A \in B_P : A :- A_1, \ldots, A_n \wedge \exists A_i \in I, enabled(A_i)\}$, where *enabled* means that either A_i is an ancestor of a side-effect, or that A_i is an ancestor to a noisy cut or sensitive meta-predicate that shares variables with the head A (either directly or indirectly). The initial value for I is the set of all sensitive calls.

The operator S_P is obviously a monotonic mapping on a finite lattice. Thus it has a least fixpoint, which can be obtained by one of the several fixpoint algorithms that have been proposed in the literature [167, 155, 120].

The sequencer uses a simple algorithm to obtain the fixpoint. As in Ullman's algorithms [167], the sequencer first constructs a *dependency graph*.

```
sibling(X,Y) :- parent(X, Z), parent(Y, Z),
                X \== Y.

cousin(X, Y) :- parent(X, Xp), parent(Y, Yp),
                sibling(Xp, Yp).
cousin(X, Y) :- parent(X, Xp), parent(Y, Yp),
                cousin(Xp, Yp).

related(X, Y) :- sibling(X, Y).
related(X, Y) :- related(X, Z), parent(Y, Z).
related(X, Y) :- related(Z, Y), parent(X, Z).
```

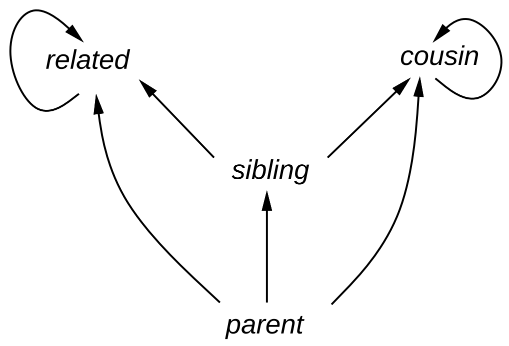

Figure 6.3: A Program and Its Dependency Graph

Basically, the nodes of a dependency graph are either the procedures of a program, or all calls in the program. If the nodes are procedures there is an arc from a node p to a node q if there is a rule with head q and with a subgoal whose procedure is p. If the nodes are calls there is an arc from a node p to a node q if the procedure whose rule contains the call p is called by the subgoal q. An example of a simple dependency graph for a family database is given in Figure 6.3 (based on Figure 3.2 of [167]).

A program is *recursive* if its dependency graph contains one or more cycles. A procedure, or call, is recursive if it is in one or more cycles (`related` and `cousin` in Figure 6.3), *nonrecursive* otherwise (`parent` and `sibling` in Figure 6.3). The main problem in fixpoint computation is how to avoid spending too much time in recursive procedures, i.e., how to optimise analysis of recursive programs.

The fixpoint algorithm used by the sequencer has a first step which sorts the dependency graph according to the dependency relations. The sorting is based on a partial order relation $\leq$ such that $p \leq q$ if q can call p but p cannot call q. The sorting process proceeds in phases: first all leaf calls are selected (those which do not result in any more calls). Next the ones only making leaf calls, and so on. The process deadlocks when it meets mutual recursion (i.e., a call p calling a call q which in turn calls p). In this case, one of the remaining calls is arbitrarily selected and the process continues.

$\Upsilon_0 \leftarrow [C_1, \ldots, C_i, \ldots, C_n]$;
$\Upsilon_n \leftarrow \Upsilon_0$;
repeat
    $\Upsilon_0 \leftarrow \Upsilon_n$;
    for $i \leftarrow 1$ to $length(\Upsilon_0)$ **do**
        if $(should\_be\_in\_set(C_i))$ **then begin**
            $add\_to\_set(C_i)$;
            $\Upsilon_n \leftarrow \Upsilon_n \backslash C_i$;
        end;
    end;
until $\Upsilon_0 = \Upsilon_n$;

Figure 6.4: Fixpoint Calculation

The actual fixpoint calculation algorithm is a variant of Ullman's semi-naive (or incremental) fixed point calculation algorithm that follows the sorted list of calls. The algorithm uses two data-structures, the initial ordered list of calls, Υ_0, and the final ordered list of calls Υ_n, and is presented in figure 6.4.

The function *add_to_set*, adds its argument to the fixpoint. The crux of the algorithm is the function $should\_be\_in\_set(C_i)$. Given the calls currently defined in the set, it says if the C_i must be in the set. Currently, the system includes C_i in the set if there are any goals in the set that are directly called by C_i and either (i) one of these goals may result in a side effect or (ii) one of the modes for C_i is don't-know. If all the modes for C_i are input or output, then the input modes cannot cause problems by definition, and unification with the output modes can be delayed until after execution of the call.

The work in each iteration may be reduced by removing all the calls that appear in Υ before the first mutually recursive call, as it is always the case that we find if these calls belong to the set or not in the first iteration. But for most programs the iteration process converges very quickly, usually in one or two iterations, and this optimisation is probably unnecessary.

Note that this algorithm does not deal with sharing of variables very precisely. That is, all right-siblings are assumed to share variables with the sensitive goal, whereas in practice they might be independent. A better solution could be used if one knew which goals share which variables. Unfortunately, only abstract interpretation using the operational semantics of the Basic Andorra Model could give a general answer to this problem, and implementing such a feature would make the current preprocessor much more complex.

6.4 Control Annotations

The last phase of the sequencer generates the control annotations necessary to guarantee correct execution of the program. The task of the annotations is to delay goals to the right of the sensitive or indirectly sensitive call until either the sensitive call has been fully executed or it is known that any sensitive calls below the indirectly sensitive call have been executed safely. We use the sequential conjunction for this purpose.

6.4.1 The Sequential Conjunction

"Sequential conjunction" prevents evaluation of any goals to its right until all calls to its left have been completely executed. The operator was previously used in languages such as PARLOG to give sequencing (e.g., near side-effects), and to increase granularity [58].

We preserve the operator '`,`' for the default, parallel, conjunction and reserve the operator '`::`' for sequential conjunction. We allow free mixing of sequential and default conjunctions in a clause, with the sequential conjunction operator not grouping as strongly as default conjunction; thus the clause `a :- b, c :: d, e.` should be read as `a :- ((b, c) :: (d, e)).`

The algorithm to sequentialise sensitive calls with the sequential conjunction is quite simple. If a sensitive call or indirectly sensitive call is the last call in a clause, nothing is done.

Otherwise the conjunction to the left of the call is made sequential.

In figure 6.5 we show an example program. The original Prolog program reads a query, preprocesses it and executes it to obtain a solution. The solution is then used for some calculation. We assume that the procedures `preprocess/2, calculate/1` and `combine/2` are pure Prolog procedures

```
prog(Z) :- read(X),
           preprocess(X, Y),
           query(Y, Z),
           calculate(Z).

query(Y, Z) :-
           write(Y),
           read(W),
           combine(Y, W, Z).
```

Figure 6.5: Example of Sequencing

The results of the algorithm for this example are represented in figure 6.6. Notice that in this case only the parallelism between `preprocess/2` and `query/1` is left.

```
prog :- read(X) ::
        preprocess(X, Y),
        query(Y, Z) ::
        calculate(Z).

query(Y, Z) :-
        write(Y),
        read(W) ::
        combine(Y, W, Z).
```

Figure 6.6: Synchronisation through the Sequential Conjunction

Note that in Andorra-I putting a sequential conjunction after a goal is only necessary if one of its right-siblings can become determinate. Otherwise, the right-siblings can never execute early. Thus, the goal `read(W)` is never determinate and there is no need to protect the goal `write(Y)` from its early execution. The sequencer only generates a sequential conjunction after some goal it (a) at least one of its right sibling may become determinate, and (b) if such a right-sibling is not after some other sequential conjunction. This optimisation avoids overheads due to supporting the sequential conjunction in the current Andorra-I engine.

The sequential conjunction can be more restrictive of parallelism than some other techniques to be discussed next (namely, than the short-circuit technique). Although one should remark that most of the parallelism is likely to arise from the areas of the program where no side-effects or noisy cuts exist, and that these areas should still be clear of sequential annotations.

There are some important advantages to the sequential conjunction. Firstly, it is useful for purposes other than just for supporting Prolog programs. By using the sequential conjunction the preprocessor thus does not need to make any special demands to the Andorra-I engine.

Secondly, the sequential conjunction makes it rather easy for a user to understand which goals have been sequenced. This is particularly useful in the cases when the sequencer has been too pessimistic and sequenced too many goals. In this case, the programmer can simply remove the unnecessary sequential conjunctions.

6.4.2 Other Control Annotations

We next discuss other control annotations considered for use by the sequencer. They include the delay declarations, short-circuits and semaphores. Of special interest is the short-circuit technique.

The Delay Declarations such as NU-Prolog's `when` declarations [117] could be used to do sequencing. The declarations necessary for this would be:

- waiting until a goal becomes leftmost (this can be trivially implemented); this would be used for side-effect builtins.
- waiting until a goal becomes leftmost or some conditions, such as an argument becoming instantiated, are satisfied; this is convenient for noisy pruning operators and sensitive meta-calls.

The main advantage of control declarations is that order constraints are less restrictive for sequencing due to noisy operators and sensitive meta-calls. An important disadvantage is that conditions like waiting for groundness may induce significant overheads.

The Short-Circuit technique has been used to provide sequencing in committed-choice applications [144]. The idea is quite simple. Each goal is extended with two extra arguments, `Left` and `Right`. Goals can execute if they have the token, which is originally placed in the leftmost goal. When a goal succeeds, `Left = Right`, and if `Left` has the token the token goes through. When a goal G creates subgoals, the leftmost's `Left` becomes its parents `Left`, G_i's `Right` becomes G_{i+1}'s `Left`, and the last subgoal's `Right` is G's *Right*.

The short-circuit technique can be adapted for the sequencer. The algorithm is as follows. A global chain is constructed for the entire program. If a call is indirectly sensitive, it receives the `Left, Right` switch, and must wait until `Left` is instantiated. If a call is not sensitive but to the left of a sensitive call it receives only the `Right` variable, and must wait until the variable is instantiated (this corresponds to waiting until the side-effect to their right is finished). Only sensitive calls, including the pruning operators, are allowed to instantiate $Left = Right$. Figure 6.8 shows for this simple example the sequentialisation introduced by the synchronisation variables. Goals are represented only by their first character. The arrows represent dependencies caused by the sequentialisation: when a goal is executed the targets of the arrows can be executed immediately. Notice that when the `read/1` goal in `query` is executed, both the `calculate/1` and the `combine/2` goals can be executed immediately.

```
prog(Z) :- read(X, token, S0),        query(Y, Z, S0, Sf):-
           preprocess(X, Y, S0),                          write(Y),
           query(Y, Z, S0, Sf),                           read(W, Si, Sf),
           calculate(Z, Sf).                              combine(Y, W, Z, Sf).
```

Figure 6.7: Synchronisation through the Short Circuit technique

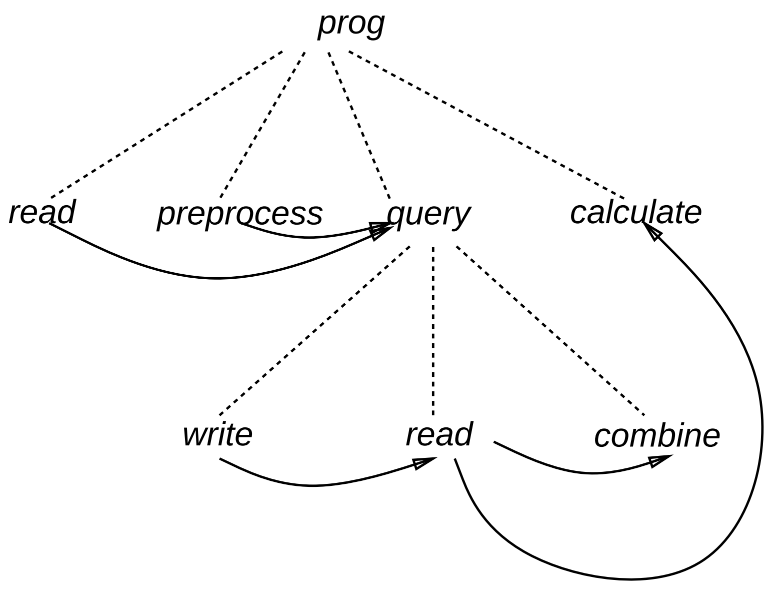

Figure 6.8: Dependencies in Short-Circuit Synchronisation

The main advantage of this method is that all computation to the right of sensitive goal can start as soon as the goal executes. But there are some problems. Firstly, one has the overhead of adding extra arguments to goal. Secondly, the technique is somewhat more complex and demands extra determinacy code in relation to the sequential conjunction. Moreover, it also makes it almost impossible for the user to improve the sequenced program. Lastly, the short-circuit technique assumes a single top-level goal. In order to query any other goal, extra program transformations may be necessary.

The semaphores technique (see Muthukumar and Hermenegildo [110]) uses similar principles to sequentialise independent and-parallel execution of programs.

6.5 Performance Analysis

We next analyse the performance of the sequencer for some applications. The applications consist of the flight allocator FlyPan [3], two versions of the Warplan plan generator [176], two versions of an extract of the Boyer-Moore theorem prover, initially written in LISP [52] and adapted to Prolog by Evan Tick, a logic synthesis program based on heuristic search, a program to detect road markings originally written by Lydia Schaaser, a protein database program, the intuitionistic predicate logic theorem prover of Sahlin *et al.*, the Berkeley PLWAM Prolog compiler, Shen's AND/OR parallel simulator, the SICStus Prolog compiler and the CHAT natural language system. The versions of Boyer differ in that the first, original version uses the builtins `arg` and `functor` to create and decompose terms, whereas the second version uses the builtin `=..` to implement the same operations. The benchmarks were selected as a sample of interesting applications of Prolog.

Table 6.1 shows the procedures that call side-effects. The last column presents the percentage of procedures that call side-effects and gives an idea of how much of the program is dedicated to Input/Output and related operations. The preprocessor avoids generating sequential conjunctions by declaring some of these procedures to be "never determinate".

Typically, Prolog programs read their input, perform a computation, and then present the output. The input and output operations will be about 10% of the source program. The actual percentage of program time spent in input and output operations will often be less than that, particularly for large applications. Input and output operations take usually linear time on their arguments, whereas the actual inner computation will usually grow much faster.

The main exception to this rule are the programs that do `assert` or `retract`. In these programs, side-effects may be embedded inside the inner computation and the sequencer will tend to disallow most early execution of goals.

	Total	Side-Effects Procedures	%
fly_pan	44	1	2%
warplan	20	2	10%
warplan2	28	13	46%
boyer	28	2	7%
dboyer	27	2	7%
satchmo	23	12	52%
road_markings	46	4	9%
protein	68	3	4%
nand	60	5	8%
intuitionistic	80	10	13%
plwam	270	27	10%
sim	344	78	23%
sicstus	376	76	20%
chat	449	28	6%

Table 6.1: Number of Procedures that call Side-effects

Table 6.2 gives the total number of conjunctions which need to be run sequentially in the program. Notice that side-effects have already been processed, and thus conjunctions are sequentialised mainly due to noisy cuts or meta-predicates. Three different approaches were used. Firstly, a pessimistic case where all cuts were noisy and all possibly sensitive calls were considered to be sensitive. Secondly, after using mode information from abstract interpretation. Thirdly, after including some mode declarations.

Note that counting conjunctions gives only a rough measure of how successful the system is. Some conjunctions might be executed more often than other, and some conjunctions might separate more important computations than others.

Without mode information or abstract interpretation, about 20% to 30% of all conjunctions are made sequential. In fact, the number of conjunctions executed sequentially is larger than this, as the system does not generate sequential conjunctions before non-determinate goals. Except for programs with large, pure Prolog computations (such as `chat`) one cannot expect much and-parallelism.

Abstract interpretation reduces the number of conjunctions to sequentialise. The actual results depend on how well abstract interpretation recognises the actual modes. The programs `dboyer`, `nand` and `protein` show good results. In contrast, abstract interpretation does not do so well for programs such as `fly_pan` or `boyer`. In the next chapter we give a more detailed analysis of the abstract interpreter.

	Total	No Analysis		With Abst. Int.		User Annotations	
fly_pan	92	29	32%	12	13%	1	1%
warplan	45	8	18%	8	18%	8	18%
warplan2	51	13	25%	11	22%	11	22%
boyer	24	6	25%	6	25%	2	8%
dboyer	21	6	29%	3	14%	3	14%
satchmo	18	4	22%	4	22%	1	6%
road_markings	163	33	20%	8	5%	8	5%
protein	153	70	46%	50	33%	41	27%
nand	178	51	29%	4	2%	4	2%
intuitionistic	186	45	24%	44	24%	37	20%
plwam	712	267	38%	254	36%	246	25%
sim	1263	396	31%	370	29%	355	28%
sicstus	1632	404	25%	361	22%	336	21%
chat	660	205	31%	185	28%	67	10%

Table 6.2: Number of sequential conjunctions versus total

Finally, the final results were obtained through hand-written mode declarations and quiet cut declarations. These results show that for most of the test programs, and as long as modes data is precise, the sequencing introduced by the system is not a serious problem. For other programs, such as `warplan` and `plwam`, that frequently test whether variables have been instantiated, sequencing results in a Prolog-style execution.

6.6 Further Work

The previous discussion has shown that the main limitations of the sequencer mostly originated from the limitations of the abstract interpreter. In general, more precise data from the abstract interpreter should lead to less sequencing. Other optimisations are possible:

Unfolding Different modes for calls in a program may be treated differently, for instance by creating two separate procedures for calls that may sometimes be sensitive, but sometimes not. To implement this technique one needs precise pattern information, hence it will be connected to designing a more flexible `AND-OR` graph for abstract interpretation.

Allowing Early Execution of More Goals As explained in section 6.3, early executions of more goals could be allowed if the sequencer knew about variable sharing in the Andorra-I execution. This depends on finding either top-down models for abstract interpretation that can follow the basic Andorra model, or in the worst case bottom-up models that can provide information about maximum aliasing [130]. In such cases it should be possible to reduce the number of calls to delay.

6.6.1 Flexible Execution of Builtins

The task of the sequencer is to guarantee correct execution of Prolog programs, or in other words, to guarantee that builtins will be executed according to Prolog's order. Where necessary, the programmer might accept a more flexible semantics for builtins, and indeed might accept an unordered execution of builtins. For instance, reads or writes to two different files might proceed simultaneously or updating different predicates in the data base might proceed in parallel.

In general, it is very hard if not impossible to capture more than some limited cases of this in a compile-time tool, as thorough knowledge of the programmer's intentions is usually necessary. The uses of `assert` in the implementation of the set predicates is one case where one can accept flexible execution of builtins. Other cases where the side-effects only affect some procedures in the programs include asserting and retracting dynamic predicates that are only known to one or two "modules" in a program, and the use of temporary files. Abstract interpretation can detect localised uses of dynamic predicates, but only user information can give absolute certainty on whether some file is just a temporary program file.

6.7 Summary

This chapter described the sequencer. The task of this module of the preprocessor is to guarantee correct execution of Prolog programs through compile-time analysis of the source program.

The sequencer's algorithm proceeds in three steps. First, the original sensitive calls are found. Second, information about sensitive calls is propagated in an `AND-OR` graph representing the program. Finally, annotations (currently sequential conjunctions) are generated to restrict the coroutining.

We also discussed the performance of the sequencer. The results showed the sequencer to be dependent on either user modes or in modes obtained from abstract interpretation. Finally, we proposed some possible improvements to the sequencer.

Chapter 7

Abstract Interpretation

Early execution of goals in Prolog programs can result in incorrect answers. One important case is if the pruning operators in the program are *noisy*, that is, if they force certain bindings of external variables by pruning (different) alternative bindings. Unfortunately, Prolog programs do not specify if pruning is noisy or quiet. In the absence of user input, it is therefore the preprocessor's task to obtain this data.

The preprocessor can use program analysis to determine if a pruning operator is to be executed quietly by Prolog. A pruning operator for a procedure P is guaranteed to be quiet if one of the next two conditions holds for *all* executions of the program: (i) it is found that all *external* variables (i.e., variables appearing in a goal calling P) are never to be instantiated by the procedure before pruning occurs; or (ii) it is found that these variables may be bound, but any other alternative bindings will be equivalent (the variables are constrained, but determinately).

Case (i) corresponds to the traditional notion of *input* arguments: that is, of arguments that are supposed to be tested by a procedure, but not instantiated by it. One important example of input argument is where the argument are known to be *ground*, (i.e., fully instantiated). Ground arguments are quite frequent, and groundness is one of the easiest conditions to determine at compile-time.

Even when variables may be constrained in a non-determinate fashion before execution of a pruning operator, this constraining may not be supposed to affect the actual execution (by Prolog) of the procedure. This is the case of *output* arguments: the state of an output argument should not affect execution of the corresponding procedure, hence it cannot make pruning operators noisy.

Thus, to detect quiet and noisy cuts, one needs to, at compile-time, analyse the source Prolog program. The standard technique used in the global analysis of logic programs is *abstract interpretation*. This is a very general methodology. Firstly, the computations denoted by a program are described in an universe of abstract objects. Secondly, an "abstract execution" in this abstract universe is called to detect the relevant program properties. In the case of Prolog, one can abstract the operational semantics to approximate both the entry and exit substitutions obtained during program execution. Lastly, such data can be used to obtain the "modes" of use of the arguments, and thus to find which pruning operators are quiet and which ones are noisy.

Notice that our goal is to detect if cuts are quiet in the left-to-right execution. Andorra-I can execute goals in advance, but will not delay Prolog goals. Hence, goals calling procedures with cuts will eventually receive all the bindings they would receive from the standard Prolog execution, hence if the cuts were quiet in traditional Prolog, and if they are receiving at least the same bindings for their arguments, then they must be quiet in Andorra-I.

The abstract interpreter used in the preprocessor is the subject of this chapter. The main issues in the design of the system are discussed in detail: the representation of the relationships between goals in the program, the abstraction data types and the iteration algorithm used to obtain the solution. The performance of the abstract interpreter is then discussed. The ensuing discussion shows how further improvements to the abstract interpreter can be obtained.

7.1 Background

Abstract interpretation was proposed by the Cousots [37] as a framework generalising many of the techniques used in global analysis of programs. The technique was first presented in the context of the imperative languages. It assumes the program has been given some *collecting semantics*, i.e., some model which records at the various program points the values of all (and only those) parameters encountered during program execution. Formally, these semantics can be specified as the fixed point $lfp(F)$ of a monotone operator, $F \in L \overset{mon}{\longmapsto} L$, on a complete lattice $L(\subseteq, \perp, \top, \cup)$. Abstract interpretation simply aims at giving an upper approximation, A, to this fixed point $lfp(F)$.

In practice, most abstract interpretation systems simplify the problem. Essentially, we can always represent a program as a set of equations $X = F(X)$. This set of equations is simplified into a new set of equations $\overline{X} = \overline{F}(\overline{X})$, where $\overline{F} \in \overline{L} \overset{mon}{\longmapsto} \overline{L}$ and $\overline{L}$, the *abstract domain*, is some partially ordered set (poset) $L(\sqsubseteq, \perp, \sqcup)$. The new equations must be such that they can be solved iteratively starting from the infimum $\perp$.

The connection between the semantic domain L and its abstract version $\overline{L}$ can now be formalised by a Galois connection: $L \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \overline{L}$. $\alpha \in L \overset{mon}{\longmapsto} \overline{L}$ is the *abstraction* function, and $\gamma \in \overline{L} \overset{mon}{\longmapsto} L$ is the *concretisation* function. α and γ are monotone functions such that:

$$\forall x \in L, y \in \overline{L} : (\alpha(x) \sqsubseteq y) \Longleftrightarrow (x \sqsubseteq \gamma(y))$$

If the abstraction of x is subsumed by the abstract value y, then y must include x as one of the values it approximates, and vice-versa.

We can guarantee that $lfp(\overline{F})$ is a correct approximation of $lfp(F)$ (in the sense that $lfp(F) \sqsubseteq \gamma(lfp(\overline{F})))$ if $\alpha \circ F \circ \gamma \sqsubseteq \overline{F}$. In other words, for an abstract value $\overline{V}$, as shown in the bottom-right corner of Figure 7.1, applying the abstract operation $\overline{F}$ must result in a more general value than abstracting the result of the concrete operation F for any of the values V that $\overline{V}$ abstracts (as shown in the bottom-left corner of Figure 7.1).

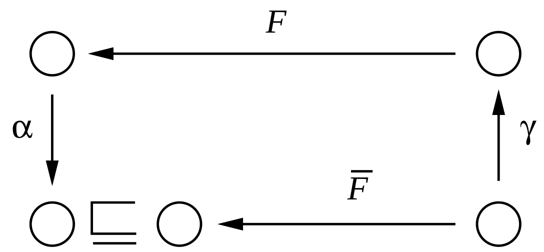

Figure 7.1: Correctness Conditions for Abstract Operations

The previous condition guarantees correctness of abstract interpretation. Termination of abstract interpretation can be achieved by imposing finiteness conditions on the abstract domain $\overline{L}$. In particular, it is sufficient that all chains in $\overline{L}$ are finite.

The initial work by the Cousots [37] presented abstract interpretation in the context of imperative languages. Several examples of data flow analysis (such as expression availability and liveness analysis), type verification and synthesis, and program testing were shown to be concrete examples of abstract interpretation. Abstract interpretation has also been used in the analysis of Prolog. As a semantics-based analysis of programs, abstract interpretation is very much dependent on the type of semantics on which it is based. In the case of logic programs, two semantics are most important:

- Declarative Semantics: Given a program they give the consequences of the program.
- Operational semantics: Given a program, they describe the operations the machine will carry out to execute the program.

As discussed in section 2.1, a model theoretic approach to logic semantics used a fixed point M_P to obtain the minimal model of the program. Abstract interpretation based on declarative semantics follows a similar algorithm to obtain an approximation to M_P. T_p is commonly evaluated in a bottom-up fashion. Corresponding abstract interpretations follow the same strategy and are usually called *bottom-up abstract interpretations*. Bottom-up abstract interpretation inherits the elegance of Logic Programming's declarative semantics, and Marriot and Søndergaard [102] show several examples of its use in program specialisation and error detection. One problem with Herbrand interpretations is that they only provide ground solutions. The declarative semantics of Falaschi et al. [49] support the introduction of non-ground terms and have been used by Giocabazzi et al. to do ground dependency and type analysis [57, 130].

Bottom-up systems provide success-substitutions for programs. In many cases, such as program compilation or parallelism detection, one is interested in finding the entry substitution for goals. This can be performed by using the operational semantics of Prolog. Because Prolog uses SLD-resolution, this type of abstract interpretation is also known as *top-down abstract interpretation*. Mellish [106] introduced the first example of abstract interpretation of logic programs. He applied his scheme to the analysis of modes, determinacy and sharing [107]. The model was later founded in terms of traces [108]. Mode inference systems have been used for some Prolog compilers [78, 192, 101]. Other initial work includes Sato *et al.* that presented a scheme based on an abstract item set construction [140]; Jones and Søndergaard [87], that used denotational semantics to construct the collecting semantics, they also presented occur-check reduction as a new application [148]; Manilla and Ukkonen, that used a simple mode set to do flow analysis [100]; and Debray and Warren [45] introduced a practical mode analysis scheme with a detailed discussion of aliasing (a refined algorithm is given by Debray [44]). Aliasing connects to the problem of compile-detection of sharing between variables, one of the most important applications of global analysis because it can considerably reduce overheads in the implementation of independent and-parallelism. Sophisticated domains have been developed to represent sharing, and have been applied to actual systems such as &-Prolog [81, 109]. Other important work includes Bruynooghe's general framework for abstract interpretation [12]; this framework was applied for several abstract domains and applications, including type inference and compile-time garbage collection [14]. Finally, other applications include Gallagher, Winsborough and others use of abstract interpretation for program specialisation [54, 53, 188], and Taylor and Van Roys's use of abstract interpretation to detect empty reference chains for variables, useful for efficient Prolog compilation [156, 174].

In our case, we were basically interested in finding if the execution of goals could instantiate variables. This can happen during head unification, or due to the operation of some builtins. We were also interested in the more general problem of finding out if calling a goal would result in instantiating of some (and which) of its arguments. We were thus interested in the operational semantics, and consequently in top-down abstract interpretation. We founded our system on Bruynooghe's framework [12] which gives a general description of the abstract interpretation process in terms of a finite abstract And/Or-tree. The framework starts from a concrete And tree (or proof tree), corresponding to the SLD derivation of a concrete goal. *Generalised* And trees are defined as having nodes Q (goals) that are adorned to the left by a call substitutions τ_i and to the right by an exit substitution τ_{i+1}, with the domain of τ_i, τ_{i+1} a subset of $var(Q)$. The basic operations on the tree are:

- *procedure entry* occurs when a leaf P is unified with a clause $H \leftarrow B_1, \ldots, B_n$. If $n = 0$ (empty body) the resulting substitution τ_{i+1} is called a *success substitution.*
- if the call P is for a builtin, a new substitution τ_{i+1} is obtained. The operation is named *interpretation of a builtin.*
- given a tree with a call P adorned to the left but not to the right, a new generalised And tree is obtained by adorning P to the right. If P is the last call of its clause, the new substitution is also the success substitution of the body; otherwise it is the call substitution for the next call. This operation is named *procedure exit.*

Consider a set of queries $Q\lambda_1, \ldots, Q\lambda_n$. Some of the resulting generalised And trees will differ only in the substitutions they contain. We build *abstract* And trees by replacing single substitutions with sets of substitutions Θ. Notice that all notions of generalised And trees can be carried over.

Finally, to cope with the nondeterminism corresponding to the fact that a predicate may be defined by several clauses, we introduce the notion of abstract And/Or-trees. In these trees, each call is associated to an Or-node. The Or-node contains a branch for every clause in the corresponding procedure. Hence an abstract And/Or-tree represents a set of abstract And trees: each element of this set can be obtained by selecting a branch at each Or-node.

Given a set of queries $Q\Theta$, the goal of abstract interpretation is to construct a correct abstract And/Or-tree. By correct one means it should describe all concrete And-trees which can occur when executing a query in $Q\Theta$. The framework guarantees that given an abstract domain and correct implementations of the basic operations, one can build an abstract And/Or-tree which is guaranteed to be a correct approximation to the program. Following this framework, the basic steps in the design and implementation of an abstract interpretation system become:

1. Design an abstract domain which (a) is an *interesting* approximation to the terms built in the program and (b) guarantees termination;

2. Implement the basic operations on this abstract domain: procedure entry, abstract interpretation of builtins and procedure exit;
3. Design a representation of the And/Or-tree: in most cases for efficiency the And/Or-tree is represented as a graph;
4. Design an algorithm to calculate the fixed point starting from the initial query.

In the next sections we show how we implemented a practical mode analysis system by discussing each of the previous points in detail. We also discuss some practical implementation questions.

7.2 The Abstract Domain

To design an abstract domain, we need to define a set of abstract terms that will, firstly provide useful information for our purposes, and secondly, guarantee termination. Termination is guaranteed if the set of possible abstract substitutions $B_j^{out}(B_j^{in})$ is a partially ordered set without infinite sequences $\alpha_1 < \alpha_2 \ldots$. As a practical consideration, the more complex the abstract domain, the more expensive will be its implementation and the more inefficient the resulting implementation.

The previous paragraph makes it clear that one has to decide on a compromise between precision and efficiency. The designers of early systems were interested in simple mode patterns, and thus settled on simple abstract domains. As an example, a very simple domain is given by Debray and Warren [45] consisting of the elements bottom; totally ground terms; totally uninstantiated terms (i.e., unbound variables); and the top element.

This simple mode system does indeed provide a first approximation for our purposes: if a term is ground it cannot be instantiated, if an argument is always an unbound variable, unification with it can be postponed until after a cut. On the other hand, it is very imprecise, and we believed that we could gain advantage from more information, particularly when analysing flat cuts. Considering the previous work on designing abstract domains, several different lines of research can be found: simple extensions to the groundness domain, domains specialised in detecting sharing between variables, domains specialised in detecting the structure of run-time terms.

As extensions to the ground/var domain, Mellish proposed `IU`, corresponding to terms with totally uninstantiated components, Richard Warren *et al.* include `nv` [187, 77] corresponding to terms known to be instantiated.

During execution program variables may, or may not, share run-time variables. This is called the *aliasing problem* and it relates to the important problem of detecting at compile-time conditions for independent and-parallel execution of goals [74].

In the context of occur check reduction, Søndergaard presented an abstract domain that represents (a) if instantiations for two run-time variables may *share* and (b) if a run-time variable may occur several times in the same substitution [148].

A very interesting abstract domain for sharing was also presented by Jacobs and Langen [81]. Muthukumar and Hermenegildo present corresponding abstract unification algorithms [109]. The abstract domain represents the fact that run-time variables may be shared by several arguments. This is represented through sets of sharing information, where a set $[A, B]$ belongs to the abstract substitution if, at run-time, the two variables A and B *may* share. For example, consider a goal $g(X, Y, Z)$, and a corresponding abstract substitution $\{[X], [Y], [X, Y]\}$. This substitution represents that (a) there may be variables occurring only in X and only in Y; (b) there may be variables occurring in both X and Y; (c) there are no variables occurring in Z, hence Z is instantiated to a ground term. In practice this domain is very specialised for detecting sharing, and thus does not provide any information about the internal structure of terms, not even about freeness (i.e., if a compile-time variable is bound). A recent extension by Muthukumar and Hermenegildo [112] adds freeness, basically by combining this domain with traditional mode information.

Abstract domains can be designed to describe the structure of terms at run-time, usually by a combination of mode and type information [13, 6]. In particular, Janssens [86] gives a thorough description of two such domains, that differ on their treatment of aliasing. In her system, simplifications necessary to obtain a practical canonical form for type terms and to guarantee convergence: a *compactness* condition is used to restrict the type graphs that can be built. For efficiency restrictions are also made on the depth of type graphs and on the possible labels for sets of types. Aliasing is represented explicitly. Bansal's [6] domain is also a full type system, but uses type variables to represent free variables. Bansal claims that, therefore, the aliasing problem is not present in his domain.

Type systems provide the most complete information about the state of arguments and are therefore closest to our requirements. They do have a serious problems: type terms can easily become very complex expressions, hence they are rather complex and expensive to implement. Moreover, type systems will be most useful it the program follows some kind of type discipline, but this is often flouted by Prolog programs (e.g., the use of builtins such as the `=..` builtin [114]). Therefore, we decided to implement a simpler system.

In our system, we initially decided to obtain only simple mode information, using an explicit representation of abstract substitutions to represent sharing. We eventually discovered that, for many applications, mode information was insufficient, and decided to implement a compromise system. Our full abstract domain is in fact similar to a domain independently proposed by Taylor [156]. It can give information more detailed than simple modes, but it is still manageable and indeed we have been able to interpret large Prolog programs.

7.2.1 The Structure of the Lattice

The abstract domain we use is as follows. Typewriter font designates a constant or functor appearing in the program and italic font designates a type defined in the abstract domain:

1. $\perp$: bottom element;
2. an actual constant, i.e., `[], a, 3`;
3. *Constant*: the set of all constants, either atom or integer;
4. `functor`$(type_1, \ldots, type_n)$: a term with main functor `functor` and whose arguments are of $type_1, \ldots, type_n$, respectively;
5. $List(type)$: a list of zero or more items belonging to type *type*;
6. $Or(\{type_1, \ldots, type_n\})$: where types $type_1, \ldots, type_n$ are exclusive types of the form 2, 3 or 4;
7. *Ground*: the set of all ground terms;
8. $Var(Label)$: a variable, and *Label* represents sharing information;
9. $\top(Label)$: top element, represents any term, and *Label* represents sharing information.

Figure 7.2 illustrates the structure of the abstract domain.

Unless specified, *type* may be any element of the abstract domain, except for the restriction that terms below a certain depth must be of type 2, 3, 7, 8 or 9, i.e., cannot have further structure. The relation of order between all the elements of the abstract domain is made clear in Figure 7.2. The fundamental differences from a typical ground modes system are:

- Compound terms are introduced in this lattice. They provide a more detailed representation for non-recursive data structures, particularly useful in describing non-ground data structures.

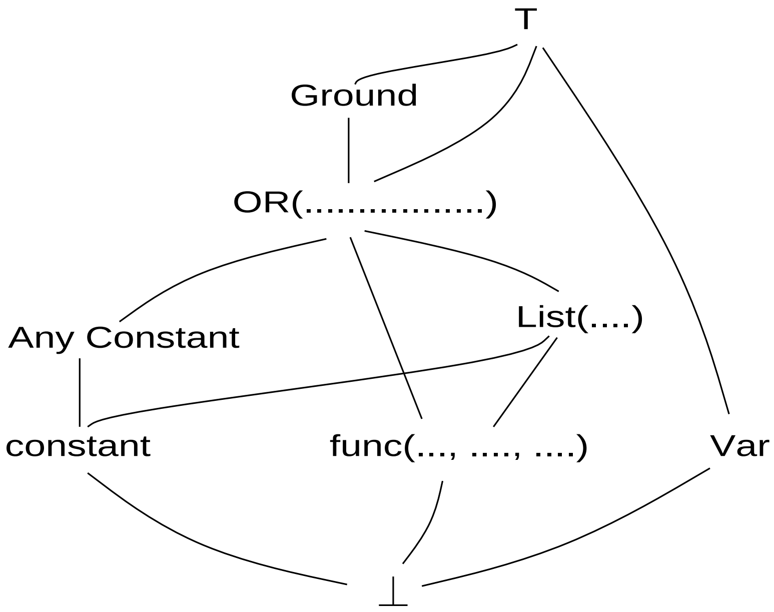

Figure 7.2: Structure of the Abstract Domain

- Lists are introduced as an example of a recursive data structure. They are a very common data structure in Prolog programs and moreover they provided some insight on the problems in implementing a full type-mode system.

We next detail some important problems in the design of the abstract domain. A detailed description of the implementation of the abstract operations on this domain is given in Appendix A.

Detailed Discussion of the Abstract Domain

The abstract domain includes individual constants and *Constant*, that represents any constant. Notice that in other systems, e.g., Taylor's [156], there is a separation between numerical and other constants. We do not believe this to be useful for our system.

The abstract domain includes compound terms. The abstract terms may represent a set of compound terms with the same functor, or a set of compound terms with one of several functors, or with arity zero (the *Or* element). A previous version of our system used, not *Or*, but a *Functor* element. This abstract term did not represent functor names, instead $Functor(type_1, \ldots, type_n)$ represented all the terms with an arity smaller or equal to the arity of *Functor*. In practice, also representing the terms with an arity *smaller* than the arity of *Functor* proved to be cumbersome, whereas preserving the functor names was shown to be very useful, say for analysing metacalls. The main disadvantage of *Or* abstract terms is that the number of elements in an *Or* may become quite large, e.g., a WAM compiler might have one different functor per WAM [178] instruction.

The abstract domain specifies that every element inside an *Or* must have a defined functor (it cannot be an uninstantiated variable). This restriction allows easier reasoning about this category of terms, and thus easier implementation, whilst it does not interfere with our goal of detecting sensitive calls: usually, if a term is possibly a variable, then it may possibly be restricted by the execution.

Lists are a very frequent type in logic programs and our system has been quite successful in detecting them. Almost every large Prolog program uses lists (although many use difference lists with uninstantiated tail). The combination of *List* and *Or* has been shown to be particularly successful.

In contrast to Taylor's [156] domain, the abstract domain allows complex terms inside complex terms (up to a certain depth). Our motivation was to detect cases where the main functor or some argument of the term is tested, but not the entire term. The obvious disadvantage is that the terms constructed during abstract interpretation become much more complex.

Sharing in our Abstract Domain

The Var term corresponds to free variables. Each Var abstract term includes a label L, shared by all free variables that **must** be bound to that same value. In contrast, a label L' for $\top$ represents the class of terms that **may** share free variables with this term. In other words:

$$L_1 = L_2 \Rightarrow \gamma(Var(L_1)) = \gamma(Var(L_2))$$

$$L'_1 \neq L'_2 \Rightarrow vars(\gamma(\top(L'_1))) \cap vars(\gamma(\top(L'_2))) = \emptyset$$

where $vars(T)$ gives all the run-time variables occurring in T. From these definitions, it trivially follows that $\forall Var(L), \top(L') : L \neq L'$, in other words the two sets of labels do not intersect.

This distinction is fundamental to correctly handle aliasing between terms. Note that aliasing in the abstract domain results from unification of free variables in the concrete domain, and that all concrete free variables are represented in the abstract domain either as Var or in $\top$. The aliasing labels between $\top$ elements correspond to Søndergaard's sharing information [148], whereas sharing between variables is similar to the way sharing is treated in say Warren and Hermenegildo's MA^3 [187].

7.3 An Example of Abstract Interpretation

The operation of the abstract interpreter is better exemplified through a small example. We use the well-known quicksort program using difference lists. The program is shown in the following figure.

```
sort([], S, S).
sort([Pivot|Ls], Sf, S0) :-
        split(Ls, Pivot, LSs, LLs),
        sort(LSs, Si, S0),
        sort(LLs, Sf, [Pivot|Si]).

split([], _, [], []).
split([El|Els], Pivot, [El|ElSs], ElLs) :-
        El < Pivot,
        split(Els, Pivot, ElSs, ElLs).
split([El|Els], Pivot, ElSs, [El|ElLs]) :-
        El >= Pivot,
        split(Els, Pivot, ElSs, ElLs).
```

A typical abstract query to this program would be of the form `sort(`*`List(Ground)`*`, _, [])`, that is, to sort a list of ground elements.

The abstract iteration process is an iterative process, where new declarations are added to a table of mode declarations until a fixed point is reached. The first iteration starts from an empty table.

Initially, the abstract query is matched to the first clause for `sort`. Abstract unification is performed between the abstract query and the clause head. Abstract unification proceeds argument by argument. The only possible unifier between any list and the empty list is the empty list, hence both second and third arguments of `sort` must match. The result of abstract unification is thus `sort([], [], [])`, and the pair (`sort(`*`List(Ground)`*`, _, [])`, `sort([], [], [])`) is entered to the table.

After having completed the first clause, the system proceeds to match the second clause. The result of the abstract unification is in this case the substitution `Pivot = `*Ground*`, Ls = `*List(Ground)*, and `S0 = []`. This abstract substitution is now applied to the call for `split/4`, resulting in the following abstract call `split(`*List(Ground)*`, `*Ground*`, _, _)`. The call matches with the first clause of `split` giving the exit substitution `split([], `*Ground*`, [], [])`. The state of the table at this point will be:

	Entry Substitution	Exit Substitution
sort	`sort(`*L(G)*`, _, [])`	`sort([], [], [])`
split	`split(`*L(G)*`, `*G*`, _, _)`	`split([], `*G*`, [], [])`

The next step is to execute the second clause for `split`. Head unification generates the bindings `El = `*Ground*`, Els = `*List(Ground)*`, Pivot = `*Ground*. The arithmetic comparison between two ground terms can succeed or not, for abstract execution we can assume it does. Finally, `split` is reactivated. The current call is `split(`*List(Ground)*`, `*Ground*`, _, _)`. This call is as general as the one stored in the table, and the algorithm returns the value on the table as the abstract exit substitution.

Having completed the last call in the `split/4`, we can obtain an exit substitution for the clause. This exit substitution is obtained by calculating the current values for the variables appearing in the head of the clause, giving `split(`*List(Ground)*`, `*Ground*`, [`*Ground*`], [])`. Notice that this is different from the exit substitution in the table. Calculating the least upper bound of the two gives `split(`*List(Ground)*`, `*Ground*`, `*List(Ground)*`, [])`. This is stored in the table as the current exit substitution.

Execution of the third clause has similar results, but affecting the last argument of `split`. The table after abstract execution of the four clauses now becomes:

	Entry Substitution	Exit Substitution
sort	`sort(`*L(G)*`, _, [])`	`sort([], [], [])`
split	`split(`*L(G)*`, `*G*`, _, _)`	`split(`*L(G)*`, `*G*`, `*L(G)*`, `*L(G)*`)`

After obtaining an exit substitution for `split`, the algorithm can apply that substitution back in the second clause for `sort`. The result is that both variables `LSs` and `LLs` become bound to *List(Ground)*. The next call for `sort` is thus of the form `sort(`*List(Ground)*`, _, [])`, and the exit substitution from the table can be used. Finally, the last call to `sort` is of the form `sort(`*List(Ground)*`, _, [`*Ground*`])`. This new call is now compared to the table, and a least upper bound, `sort(`*List(Ground)*`, _, `*List(Ground)*, is obtained.

This new substitution is more general than the previous, and a new call to `sort` is made. This time, abstract execution of the first clause returns `sort([], `*List(Ground)*`, `*List(Ground)*`)`.

This abstract exit value is used to calculate the exit substitution for the second clause, which is now `sort([Ground], List(Ground), List(Ground))`. The least upper bound of this exit substitution with the one currently in the table returns `sort(List(Ground), List(Ground), List(Ground))`. At this point, the state of the table is:

	Entry Substitution	Exit Substitution
sort	`sort(L(G), _, L(G))`	`sort(L(G), L(G), L(G))`
split	`split(L(G), G, _, _)`	`split(L(G), G, L(G), L(G))`

The values in the table give the final exit substitution for `sort`. We thus have completed the first iteration.

A second iteration is launched, now based on the modes in the table. The results of the second iteration confirm the values from the first, and abstract interpretation theory guarantees we have reached the fixed point and obtained the modes for the program.

7.4 Representing the abstract And/Or-tree

A fundamental problem in the design of an abstract interpretation system is how to represent the abstract And/Or-tree. Consider the program:

```
a(X,Y) :- b(X), b(Y).          b(X) :- d(X).
a(X,Y) :- c(X), b(Y).          c(_).             d(_).
```

The left tree in figure 7.3 gives a concise representation of an And/Or-tree for execution in the simple abstract domain $\{\top, Ground, Var, \bot\}$, or more concisely, $\{t, g, v, b\}$, of the abstract query $a(g, v)$.

First, notice that some branches of the tree are duplicates. A more compact representation is possible by folding these branches onto a single branch. The resulting data structure is an And/Or-graph. The one corresponding to the initial And/Or-tree is also shown in figure 7.3.

There is no loss of precision in this transformation. But, unfortunately, for non-trivial programs the And/Or-graph can still be too detailed. Further constraining of the number of nodes in the graph is thus necessary. More *compact* graphs are presented in figure 7.4. In the left figure, each node corresponds to a procedure call in the program clauses. As both calls to `d(X)` are launched from the clause for `b`, they are folded in a single node. In

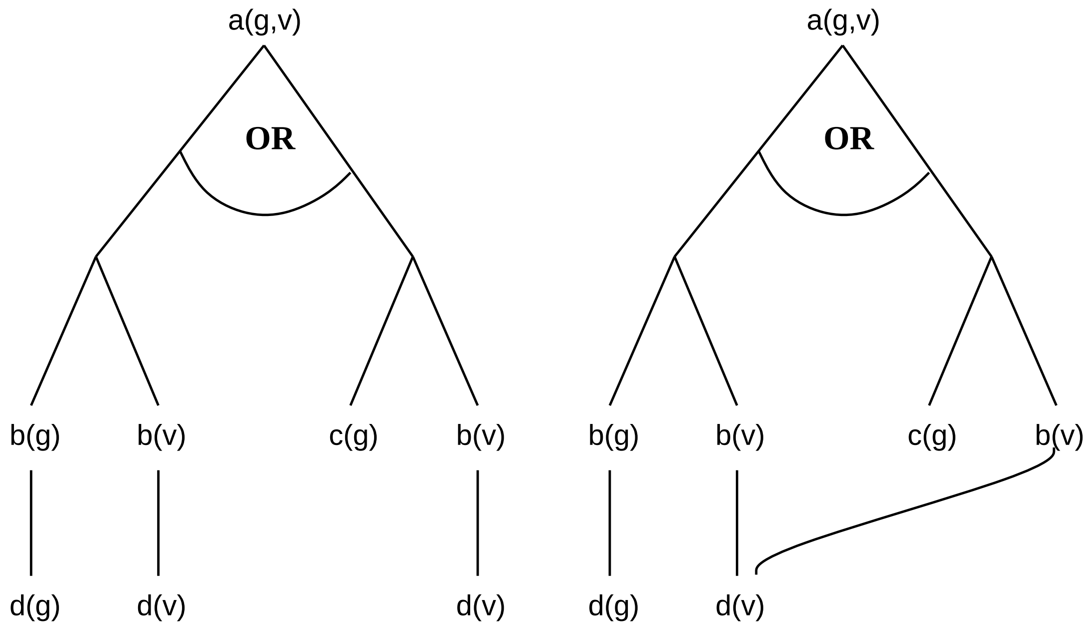

Figure 7.3: And/Or-tree versus And/Or-graph

the figure to the right, each node must correspond to a different procedure. Thus all nodes for `b` are now merged into a single node. The two compact graphs are named call-level and procedure-level representations.

There is a loss of precision in both cases, but more severe in the case of procedure-level representation. Other intermediate representations are also possible (see Janssens [86]). Muthukumar and Hermenegildo [109] suggest that call-level abstract interpretation is particularly useful for the detection of independent and-parallelism. We have implemented both call-level and procedure-level representations, and verified that call-level graphs are indeed more precise, but still not sufficiently accurate for some problems (as discussed in section 7.6).

And/Or-graphs are usually implemented as memo tables. (An alternative is use to streams [175] but that depends on support from some coroutining facilities), One can store the memo table in the Prolog database or use a dictionary. Choosing the first solution is less elegant, but allows the use of backtracking in the implementation of the abstract interpreter, with the corresponding advantage of needing less memory space.

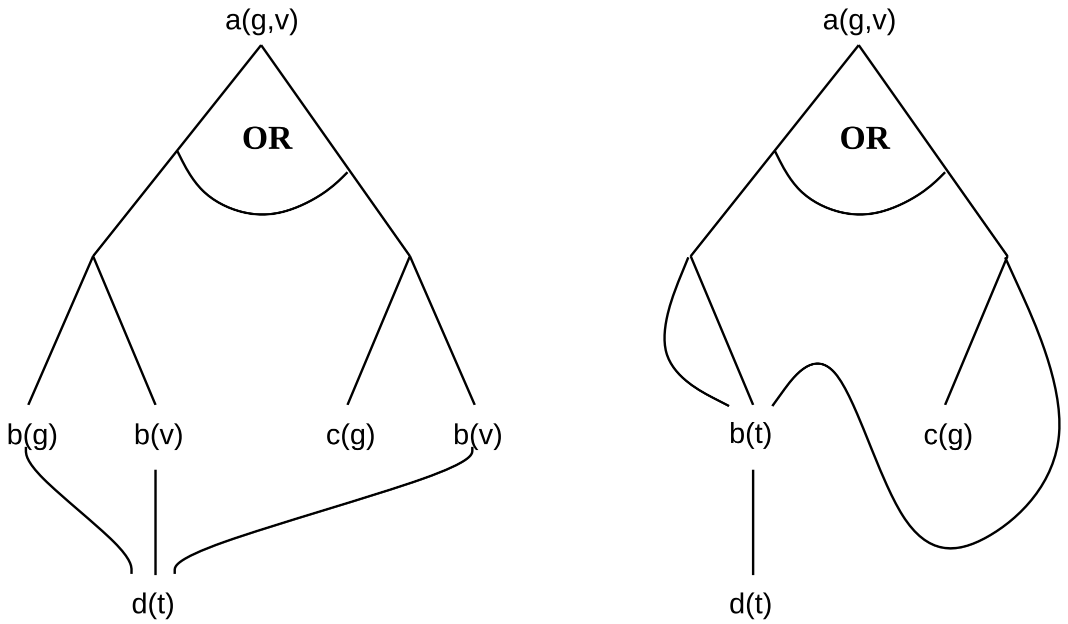

Figure 7.4: Two compact And/Or-graphs

7.5 Calculating the Fixed Point

Abstract interpretation can be seen as the process of decorating the And/Or-graph with the abstract entry and exit substitutions. In Bruynooghe's algorithm [12], this process is executed top-down. For each call P, the algorithm distinguishes four cases:

1. builtins, in this case abstract interpretation for built-ins is performed;

2. nodes P with no ancestor node with a call for the same predicate, in this case perform procedure entry;

3. nodes P for which there is an ancestor P' with a call for the same predicate such that all $B_j^{in} \equiv B_j^{'in}$ (up to renaming). In this case, one uses a renaming of $B_j^{'out}$ as the output substitution. If $B_j^{'out}$ is not available, return $\perp$. Notice that if at some later point one obtains values new values B_j^{out} such that $B_j^{out} \sqsubseteq B_j^{'out}$, one has to repeat the computation based on P. In this case it is said that the node P *refers back* to the ancestor P'.

4. nodes such that there is an ancestor P' but for which the previous case does not apply. In this case, execute as in 2. Bruynooghe introduces further restrictions if the depth of restriction exceeds a certain limit: if $B_j^{'in} \sqsubseteq B_j^{in}$ proceed as in 3 otherwise calculate $B_j^{'in} \sqcup B_j^{in}$ and proceed as in case 2.

When procedure or call-level representations are introduced, the important restriction on the recursion level becomes a restriction on if it is the first visit to that call or procedure.

In our system we basically implement a version of this algorithm. With procedure-level representation, only one call for P can exist, which simplifies the implementation of cases 2 and 3. Otherwise, to correctly implement case 2 for every call C the system looks up the database for a call C' with matching entry substitution (we call this operation *look-up*).

A problem occurs when the same call receives several different entry substitutions, i.e., it may result from several different branches in the And/Or-graph. In this case, when a new substitution $B_j'^{in}$ for a call P is received, one might have to redo all the branches depending on this call. To simplify the implementation, we use a step-by-step algorithm: for each step, execution proceeds top-down left-to-right; if modes for a goal change, the algorithm marks this step as not having converged, but does not try to redo other branches calling that goal. In order for the algorithm to converge, each step is allowed to use the abstract substitutions of the previous step when looking at a new goal: Essentially it calculates the least upper bound with the entry modes from the previous step, and the initial output substitution is initially the output substitution calculated in the previous step (this is useful for recursive calls). The final algorithm is:

1. Initialise every $B_j'^{in}, B_j'^{out}$ to $\bot$.
2. Initialise *Converged* to `true` and $\forall P : ancestor(P) = false$.
3. Execute using a left-to-right depth-first search rule. For the current call P:
   (a) If the current $B_j^{in} \sqsubseteq B_j'^{in}$ replace B_j^{in} by $B_j'^{in}$, else go to step 3b. If $ancestor(B_j)$ is `false`, set $ancestor(B_j)$ to `true`, and do procedure entry. If $ancestor(B_j)$ is true, the exit substitution is $B_j'^{out}$, and this node refers back to the node that set $ancestor(B_j)$ to true.
   (b) Set $B_j'^{in}$ and B_j^{in} to $B_j'^{in} \sqcup B_j^{in}$, $B_j'^{out}$ to $B_j''^{out}$. Set *Converged* to `false` and $ancestor(B_j)$ to `true`. Perform procedure entry.
4. When all goals have been executed, if *converged* is `true`, exit. Otherwise, go to 2.

The main disadvantage of the algorithm is that when a step fails to converge the algorithm must recompute from the beginning. The advantages are that it is simple to implement and, that compared to an algorithm which would recompute branches immediately, it is not very vulnerable to situations where a call would be recomputed several times only to eventually find out that some other branch not yet tried generates a new substitution and forces a new recomputation.

7.5.1 Other Fixed Point Algorithms

In our system we were interested in obtaining a simple, easy-to-implement algorithm. More sophisticated and complex algorithms have been proposed in the literature. O'Keefe proposed an elegant algorithm where nodes are ordered by dependency relations and their values are initialised either to $\perp$ or a value obtained from rules. A stack is used to maintain and propagate the nodes that change [120]. Unfortunately, to maintain the stack one cannot use backtracking.

Several fixed-point computation algorithms, specifically designed for abstract interpretation of logic programs, have been proposed more recently [113, 172, 96]. Essentially, Muthukumar's algorithm [113] does immediate recomputation of the first, topmost calls when their modes change. To prevent extraneous computations, he classifies clauses as recursive or non-recursive. Non-recursive clauses are tried first and give a base approximation. Computation for a recursive call is then repeated until it reaches a local fixed point. Local fixed points propagate up the tree until they form a total fixed point. The representation is slightly more complex: mainly it needs to classify procedures; local-fixed points are still vulnerable to changes somewhere else in the program, although classification of procedures should prevent most cases. Both Muthukumar [113], and Le Charlier et al. [96] use a dependency graph to avoid redoing computation.

Our system was designed to keep the fixed point algorithm as simple as possible. Even so, and as used by Muthukumar's, there is a very good advantage in doing non-recursive clauses first: the non-recursive clause can generate a first solution to the procedure. This solution can be used by the recursive clauses. As a result, the number of iterations can be decreased. To gain some of the advantages of this we apply a very simple principle in our algorithm: clauses for a procedure which include calls to the procedure itself (directly recursive) are tried after any non-direct recursive clauses.

7.5.2 Unfolding the And/Or graph

We previously have assumed that the nodes of the And/Or-graph are fixed. Unfolding the And/Or-graph can be useful in two situations. First, when the same goal is called with "very different" modes. In this case, instead of directly calculating the least upper bound, one can process separately both entries. A similar situation occurs when two exit substitutions for the same call are "very different". Again the two exits substitutions can be processed separately. A simple definition for "very different" is if one argument is in one case a free variable, in the other a non-variable term.

Van Roy [172] gives an interesting example of the need for unfolding. The example is:

```
main :- a(9,a, _, _, _, _, _, _, _, _).

a(0, _, _, _, _, _, _, _, _, _).
a(N, A, A, C, D, E, F, G, H, I, J) :- N1 is N-1,
            a(N1, A, C, D, E, F, G, H, I, J, A).
```

AQUARIUS will treat each call $a(i, \ldots)$ separately and hence is able to ascertain that the exit substitution for the initial substitution will have all its elements ground.

We have applied this optimisation to our system. We define a function *modes_closeness* that verifies when two entry modes are "different". If so, we create a new alternative node for the different mode set. Note that as the process converges, two nodes which were initially "different" may move up the lattice and cease being "different". To address this problem, at step 2 of the previously described algorithm, we verify if the nodes do not obey *modes_closeness*, and if so, merge them.

7.6 Performance of the Abstract Interpreter

The literature includes several references to the performance of abstract interpretation systems [77, 156, 101, 174]. We now analyse the practicality of our abstract interpreter, first by giving a brief complexity analysis and then by studying its performance on several benchmark programs.

7.6.1 Practicality of Abstract Interpretation

The complexity of abstract interpretation is a function of the complexity of the fixed point algorithm, and of the complexity of each operation. O'Keefe [120] gives $O(dmax \times size\ of\ problem)$ as a worst case complexity for any fixed point calculation algorithm (in the worst case, it would do one operation for all the elements of the lattice). In our case $dmax$, the size of the lattice, corresponds to the maximum length of a chain $a(\perp, \ldots, \perp) \sqsubseteq \ldots \sqsubseteq a(\top, \ldots, \top)$, and *size of the problem* corresponds to the number of nodes of the problem (in our case, the number of nodes in the And/Or-graph).

We now need to consider the operations in the abstract domain. Of the several operations performed, the most important are abstract unification, projection of an abstract term from the environment, comparison of abstract terms, and calculation of the least upper bound. Abstract unification in our domain is proportional to the size of the input terms times the maximum size of the environment: each argument of a term is considered once, so this operation is linear on the size of the term, but access to the environment takes time proportional to the environment size (at least, in the current implementation). Comparison between abstract terms and calculation of the least upper bound take linear time on the size of the abstract term, whilst projection of the current environment into a term takes time proportional to the size of the environment times the size of the term. The size of the environment is a function of the number of variables appearing in a corresponding abstract clause, and of the size of the abstract substitutions for each variable. In the worst case, this size can grow exponentially with the depth of the abstract domain.

The complexity of the system will be, in the worst case, the product of the number of operations performed times the complexity of each operation. Briefly, this can be described as:

$$O(dmax \ \times \ size\ of\ And/Or\ graph \ \times \ (size\ of\ abstract\ goal)^2)$$

A conclusion from this brief analysis is that execution time will grow very quickly with the depth of the abstract domain, and not so quickly with the number of the nodes in the abstract domain. In practice, the growth will be rather limited, as many terms can converge (say to $\top$ quite quickly) and as a good fixed point calculation algorithm will avoid doing the same computation several times. We next discuss how varying the maximum depth and the size of the And/Or-graph affects performance in actual real application benchmarks.

Table 7.1 gives the times needed to perform abstract interpretation for the suite of benchmarks used for the sequencer. The times were obtained under SICStus Prolog 2.1 running in a Sun SPARCserver 470. The first three columns give three different measures of program size: number of lines in the source files, number of calls in the program, and the time (in seconds) necessary to compile the programs to SPARC assembly code. The next columns give the times (in seconds) to do analysis by four different versions of the abstract interpreter. The first two use a call-level representation of the And/Or-graph, with respectively maximum term depth of two and three, and the second use a procedure-level representation, respectively with maximum depth of either two or three.

It is interesting to compare compilation times with analysis times (this is shown graphically for the four versions of the compiler in Figure 7.5). In general, our system is slower than the SICStus compiler. This result agrees with Taylor's and Van Roy's [156, 174], in suggesting that precise abstract interpretation can be expensive. Notice that performance of our system could still be much improved (immediate improvements are by using even better fixed-point

	Size as			Graph Type			
				Procedures		Calls	
	Lines of Text:	Calls:	ISP compile:	Max. Depth: 2	3	Max. Depth: 2	3
fly_pan	917	155	2.4	3.1	4.3	5.9	8.0
warplan	192	98	0.98	-	9.0	-	19.5
warplan2	698	206	1.5	-	10.0	-	23.6
boyer	411	62	2.7	4.6	7.4	7.0	12.9
dboyer	418	59	2.7	4.3	6.9	6.0	11.3
satchmo	187	74	1.15	2.8	7.6	4.7	10.8
road_markings	799	268	6.8	3.6	7.0	6.9	12
protein	2287	537	8.6	6.4	8.3	10.5	12.5
nand	706	271	4.3	9.4	12.7	21.6	33.7
iltp	905	408	7.7	27.7	85	109	586
plwam	3031	1439	15	30.9	35	75	82
simul	4268	2036	24	80	378	289	14749
sicstus	4354	2252	26	78	128	464	1186
chat	4891	1266	39	79	170	144	544

Table 7.1: Size versus Runtime (in seconds)

algorithms and by improving the data representation for environments). Also parallelism (particularly or-parallelism in exploring the And/Or-graph) can be envisioned as a means to speed up the search.

The time ratio of call-level to procedure-level abstract interpretation can vary substantially. In general it gets worse for longer programs. Increasing the maximum depth of terms can even have a worse effect on execution time, as expected. Notice that in many cases abstract terms will be quite simple (e.g., of the form `atom`, Var or $\top$), and will have fixed depth. Finally, performance varies strongly with each program, and does not depend directly on size. For each program, it depends strongly on how easily the fixed point computation can find the correct solution, and on the complexity of the abstract terms the system manages to detect.

7.7 Precision of Abstract Interpretation

We can now address the value of abstract interpretation. We first give a classical application, mode analysis. The number of ground and unbound arguments gives a good idea of the precision of the system in detecting noisy cuts (a performance analysis was given in section 6.5). As an aside of some interest, we also discuss the effectiveness of this tool for detecting

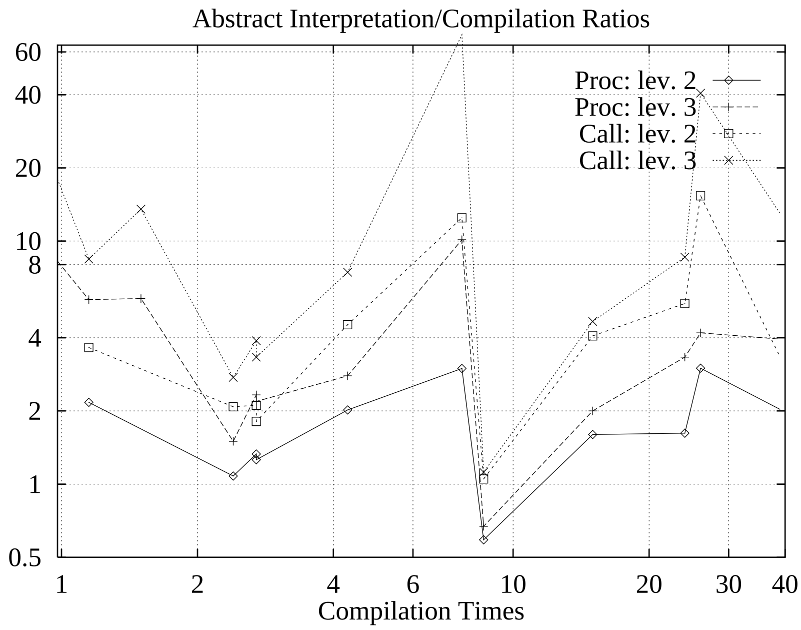

Figure 7.5: Performance of Abstract Interpretation

independent and-parallelism.

Table 7.2 shows what modes were found by abstract interpretation. We first show the modes obtained by using a simple procedure based And/Or-graph. The table presents the number of ground, unbound, bound (but possibly not to a ground term) and unknown arguments.

The results are very good for programs that process ground terms such as `nand`, `dboyer` and `protein`. The `boyer` program is an example of the impact of builtins. The interpreter is able to handle `=..` much better than `functor,arg`, and the results are thus much better. For the three last programs, the results for `iltp` are probably also due to the use of `functor,arg`, and because `ipltp` uses open recursive data structures. The results for `plwam` are actually good considering that this compiler uses the logic variable intensively for meta-programming. The results for `chat` are limited again by the use of `functor,arg`, and by difference lists. Comparing with other systems, the results for `boyer` and `nand` are slightly better than the results obtained by Van Roy [174].

7.7.1 Depth Level and Call Graphs

Table 7.3 shows the effect of increasing the depth of the maximum abstract term in the analysis from 2 to 3. There are some improvements, but not very substantial.

	Total	Modes (percentage of arguments)			
		ground	var	nonvar	any
fly_pan	137	55	22	5	18
warplan	69				
warplan2	117				
boyer	72	3	24	13	61
dboyer	71	72	28	0	0
satchmo	65	5	29	37	29
road_markings	145	6	39	0	0
protein	197	55	29	13	3
nand	292	76	24	0	0
iltp	274	15	16	32	38
plwam	831	14	23	16	48
simul	1810	23	24	8	46
sicstus	1533	18	24	9	49
chat	2105	18	37	6	40

Table 7.2: Precision of Abstract Interpretation Analysis

Table 7.4 shows the effect of using a call-based graph, instead of the procedure-based graph. The improvements in this case are more substantial. Note that because there are many more nodes, there are more calls in the graph.

Table 7.5 shows the effects of combining the two improvements. The results are quite similar to using abstract term maximum depth of 2.

The conclusion is that there is a definite advantage in using call graphs: precision for non $\top$ terms improves up to more than 10%. More detailed description of terms has less significance and only `iltp` benefits seriously. Although the usual caveat that other programs may behave very differently applies, the results do suggest that for our system there is no big gain in using a depth of more than two, but there is a gain in using a more detailed description of the And/Or-graph, such as the call-level representation.

Independent And-Parallelism Even if arguments are detected to be $\top$, the system is still able to produce some sharing information that can be used to detect independent and-parallelism. We adapted the abstract interpretation system to &-Prolog's compiler [74]. Sharing and groundness information is used by the &-Prolog compiler to generate CGEs according to the CDG method [111]. The results of using our system of abstract interpretation in &-Prolog's compiler are compared with Muthukumar and Hermenegildo's sharing interpreter [109] in table 7.6.

	Total	Modes (percentage of arguments)							
		ground		var		nonvar		any	
fly_pan	137	55		22		5		18	
warplan	69	6		19		25		51	
warplan2	117	43		15		18		24	
boyer	72	3		24		13		61	
dboyer	71	72		28		0		0	
satchmo	65	5		29		45	(+8)	22	(-8)
road_markings	145	6		39		0		0	
protein	197	55		29		14	(+1)	3	
nand	292	76		24		0		0	
iltp	274	15		16		32		37	(-1)
plwam	831	14		24	(+1)	18	(+2)	44	(-4)
simul	1816	25	(+2)	25	(+1)	14	(+6)	35	(-9)
sicstus	1529	18		25	(+1)	10	(+1)	47	(-2)
chat	2104	18		37		6		39	

Table 7.3: Increasing Depth to 3

The results show that the system is comparable to the sharing interpreter. In fact, it performs better in matrix where it verifies the ground test in the CGE that creates most of the parallelism (though the new freeness abstract interpreter should be able to eliminate that test [112]). Better results could be obtained considering data about when two terms are known to share.

7.8 Analysis and Future Work

So far, we have described the design and implementation of a practical system for abstract interpretation. The goal of this system was to use abstract interpretation in order to find noisy cuts and sensitive builtins. Current results show that (a) the system is robust, (b) for most programs the system is able to obtain useful data, and (c) there is definitely still room for improvement. This section analyses firstly immediate improvements which experience showed to be of interest and lastly how fundamental changes could improve the system.

7.8.1 Refinements

There are several obvious improvements to the system. As regards low-level considerations, improving the representation of environments and the representation of the memo-table should result in much better performance. More significant improvements are possible in

	Total	Modes (percentage of arguments)							
		ground		var		nonvar		any	
fly_pan	287	56	(+1)	23	(+1)	9	(+4)	12	(-6)
warplan									
warplan2									
boyer	72	6	(+3)	19	(-5)	20	(+7)	56	(-5)
dboyer	97	76	(+4)	24	(-4)	0		0	
satchmo	137	3	(-2)	23	(-6)	37		37	(+8)
road_markings	225	6		39		0		0	
protein	530	61	(+6)	26	(-3)	12	(-1)	1	(-2)
nand	810	76		25	(-1)	0		0	
iltp	921	20	(+5)	14	(-2)	32		34	(-4)
plwam	2061	18	(+4)	23	(-1)	20	(+4)	40	(-8)
simul	3555	23		25	(+1)	10	(+2)	42	(-4)
sicstus	4467	21	(+3)	24		8	(-1)	47	(-2)
chat	3983	19	(+1)	39	(+2)	6		35	(-5)

Table 7.4: Using Call Based Abstract Interpretation

the:

Abstract Domain Two limitations affect the abstract domain: detailed analysis of sharing and the need for full type information. As regards sharing, a simple solution is to use information from a different abstract interpreter, e.g., Muthukumar and Hermenegildo's [109]. A more sophisticated approach would be to use their approach in the domain proper. As regards full types, one can contemplate an evolution to full (and complex) recursive types in the style of Janssens' [86], or the simpler solution of extending the abstract domain.

The benchmarks show that three new abstract types could improve performance. First, a new element between Var and $\top$ could indicate that a term is a variable, even when sharing is not precisely known. Second, a type subsuming Or could be used to represent the output of `functor/3` and other (frequent) cases where the main functor's name is unknown.

The main limitation of the abstract domain is its inability to represent open recursive data structures, i.e., recursive data structures with free variables. An important example are difference-lists, which are quite common in Prolog programs (almost as common as the use of complete lists). The obvious solution would be to include a new type specialised for difference lists. A simpler alternative is based on the concept of *linear terms*, i.e., terms where variables are guaranteed to appear only once. In this case we would extend the abstract domain Or to include variables known not to appear in the other elements.

	Total	Modes (percentage of arguments)							
		ground		var		nonvar		any	
fly_pan	287	56	(+1)	23	(+1)	9	(+4)	11	(-7)
warplan	161	8		20		35		36	
warplan2	273	47		16		19		18	
boyer	99	6	(+3)	18	(-6)	20	(+7)	56	(-5)
dboyer	97	76	(+4)	24	(-4)	0		0	
satchmo	137	4	(-1)	23	(-6)	44	(+7)	29	
road_markings	225	6		39		0		0	
protein	530	61	(+6)	26	(-3)	12	(-1)	1	(-2)
nand	810	75	(+1)	25	(-1)	0		0	
iltp	921	20	(+5)	14	(+2)	32		33	(-5)
plwam	2061	18	(+4)	22	(-1)	22	(+6)	38	(-10)
simul	3555	25	(+2)	27	(+3)	16	(+8)	32	(-14)
sicstus	4465	21	(+3)	24		8	(-1)	47	(-2)
chat	3994	19	(+1)	39	(+2)	7	(+1)	34	(-6)

Table 7.5: Using a Call based graph and Depth Level 3

	CGEs	indep tests		ground tests	
		share	struct	share	struct
qsort	1	0	0	0	0
matrix	5	1	0	5	5
boyer	5	5	5	3	3

Table 7.6: Abstract Interpretation for IAP

Improving Performance In section 7.5.1 several fixed point algorithms are discussed. Algorithms such as Muthukumar's [113] and Le Charlier *et al.*'s algorithm [96] seem particularly interesting in that dependency graphs should reduce unnecessary recomputation. Although one can expect a more complex algorithm to reduce the number of inferences to perform, more complex algorithms will have higher overheads.

Abstract compilation is an elegant transformation that can be simply described as a specialisation of a program for the particular abstract domain being considered [45]. One can go a step further and compile the program into WAM-like code [153]. Implementing a Wam emulator in C can give one or two orders of magnitude speed-up. This is quite interesting for a stable version of the system, but loses the flexibility of the Prolog implementation.

Unfolding the And/Or-graph In subsection 7.5.2 we analysed the advantages of creating new nodes in the And/Or-graph when the same predicate is called with different modes.

A related problem is typical in the use of `functor` and `arg`, say the case of `copy_term`:

```
...
copy_term(X,Y) :-
        functor(X, Na, Ar),
        functor(Y, Na, Ar),
        copy_term_it(Ar, X, Y).

copy_term_it(0, _, _).
copy_term_it(N, X, Y) :-
        N > 0,
        arg(N, X, A),
        arg(N, Y, B),
        cp(A, B),
        N1 is N-1,
        copy_term_it(N1, X, Y).
```

consider the abstract call $copy\_term(a(g,g),\_)$. The term `Y` is built argument by argument, and the current abstract interpreter will approximate $\_ \sqcup a(\_,\_) \sqcup a(g,\_) \sqcup a(g,g)$ as $\top$. In this case one would like to keep the intermediate calls away and return only the final solution $a(g,g)$. To do so, the system needs to provide:

- Precise Interpretation of BuiltIns: in the example above the range of arities from `functor` should be given. Other builtins one needs to support are `is`, arithmetic comparisons, and `arg`.
- Detailed Conditions for Creating a New Node: this is a similar problem to partial evaluation. A simple and useful condition is to allow only deterministic unfolding, that is, a new node that is not sufficiently "different" would only be created if it was the only possible matching goal.

Our experience shows that performance of the abstract interpreter is so far constrained by three main problems: difference lists, the use of `functor` and `arg` to build and destroy terms, and limitations in the And/Or-graph. Solving these problems should significantly improve the quality of abstract interpretation.

7.8.2 The Future

Recent work on abstract interpretation is placing more emphasis on theoretical frameworks and on abstract interpretation of concurrent or constraint logic programming languages. Of some interest is the use of *magic sets* to obtain answer substitutions from bottom-up analysis [119, 89]. Initial algorithms seem to be not more precise than conventional frameworks, although they do result in more elegant implementations. A problem with magic sets is that they operate by transforming the original program clauses, thus the naive application of magic sets will generate the compact procedure-level and/or-graph.

It would be interesting to do operational-semantics based abstract interpretation for the Basic Andorra model (this is related to doing abstract interpretation for the committed-choice languages). In both cases, the difficulty is that execution order is not fixed and may indeed vary from run to run. Therefore, it is difficult to build the correct and/or-graph. Moreover, it becomes less clear what is the correct mode for a goal, as a goal may be called with several different modes, according to the way it is scheduled. In the case of the Basic Andorra model, one also has to find out which calls are non-determinate. This adds a further level of complexity. This work may be aided by techniques similar to the ones developed by Ueda and Morita [166], that provide mode dependencies between goals, or by recent work on abstract interpretation of the committed-choice languages and of the constraint logic programming languages [103, 32, 29, 55, 80, 48, 5].

The Cousots have suggested that using widening/narrowing might provide better results than Galois connections [38]. In general, this corresponds to having a more flexible approach to global analysis of program. In particular one would use infinite domains, and one could combine global analysis with program transformation (Gianotti and Hermenegildo propose a simple but useful system [56]). Such “flexible” techniques are probably necessary for the precise analysis of large, complex programs.

7.9 Summary

This chapter presented one of the components of the preprocessor, the abstract interpreter. This component is used in the analysis of Prolog programs. The resulting mode information is used by other modules of the preprocessor.

Abstract interpretation is a methodology used in the global analysis of programs. Our system used top-down abstract analysis to detect the input and output patterns of goals. The system is based on Bruynooghe’s framework. The abstract domain improves on ground and var abstract domains by representing compound terms and the recursive data type lists. The abstract domain is also able to represent sharing. Execution of the program is represented through an And/Or-graph. This graph is implemented in two compact forms, one where each node corresponds to a procedure in the program, the other where each node corresponds to a call to a procedure in the program. A fixed-point iteration algorithm that proceeds until finding the correct approximation is also implemented in our system.

The performance data presented showed the system to be slower than a Prolog compiler, but still able to analyse real Prolog applications. The results of the analysis tend to be better for smaller benchmarks. Analysis of the applications show the main limitations to be the And/Or-graph, some limitations on the abstract domain, and some limitations on the processing of builtins. Finally, some improvements to the abstract interpreter were suggested.

Chapter 8

Andorra-I Prolog

The user language for Andorra-I is Andorra-I Prolog, a language that extends Prolog with the implicit coroutining available in the Andorra selection function. We have so far showed how Andorra-I can support Prolog applications and still allow parallelism. We will now concentrate on the extensions that Andorra-I Prolog offers over Prolog, that is, on the advantages of the implicit coroutining in Andorra-I.

We show that the implicit coroutining of Andorra-I can be particularly useful for problems which have a simple declarative formulation, but where a rigid left-to-right selection function is inadequate. We give two important examples. The first example shows how coroutining is needed to run a committed-choice style program efficiently. The second example shows the coroutining can dramatically reduce the search space for a logic program.

The flexibility of the Andorra-I system raises the question of whether Andorra-I Prolog should also be extended to support programs written for committed-choice languages such as Flat PARLOG or FGHC, or for programs written for coroutining languages such as NU-Prolog, or the constraint logic programming languages. We present the main issues in fully supporting these languages.

8.1 Coroutining with the Andorra Selection Function

We first give an example of how the Andorra selection function gives rise to coroutining in the style of the committed-choice languages. The example (due to Warren [181]) is a unidimensional version of Conway's game of Life. In the game of life a set of cells evolves for a number of steps. Each cell can be either `on` or `off`. A cell is `on` if one and only one neighbour is `on` *in the previous step*, and `off` otherwise. The evolution of the 8-cells board for a specific initial state is shown by figure 8.1. Notice that the last state is *stable*, that is, the state of the cells will not change further.

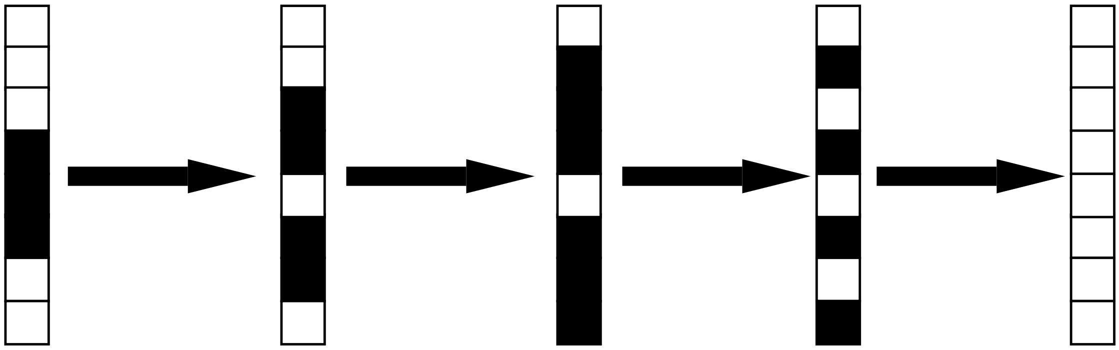

Figure 8.1: The Game of Life

A simple program that describes the cell automaton is shown next. The top-level procedure `life/1` gives the setting of the board. The argument to `life/1` is the number of steps to execute. The procedure `cell/4` describes a new step for a cell. If `Steps` is `0`, the life of the cell is over. Otherwise, the first three arguments to `cell` contain the states from the initial up to the last step of the cell and of its two neighbours. The second argument contains the cell itself, whereas the first argument contains the left neighbour and the third argument the right neighbour. Finally, `event` says that a cell, the middle argument, is `on` if only one of its neighbours is `on`.

```
life( Steps ):-
        L1 = [off|LL1],    L2 = [off|LL2],
        L3 = [off|LL3],    L4 = [on|LL4],
        L5 = [on|LL5],     L6 = [on|LL6],
        L7 = [off|LL7],    L8 = [off|LL8],
        cell(L1, L2, L3, Steps),     cell(L2, L3, L4, Steps),
        cell(L3, L4, L5, Steps),     cell(L4, L5, L6, Steps),
        cell(L5, L6, L7, Steps),     cell(L6, L7, L8, Steps),
        cell(L7, L8, L1, Steps),     cell(L8, L1, L2, Steps).

cell([X1|L1], [X2|L2], [X3|L3], Steps):- Steps > 0,
        StepsLeft is Steps-1,
        L2 = [NX2|NL2],
        event(X1, X3, NX2),
        cell(L1, L2, L3, StepsLeft).
cell([_], [_], [_], 0).
```

```
event(off, off, off).
event(off,  on,  on).
event( on, off,  on).
event( on,  on, off).
```

Executing this logic program in Prolog would be very inefficient. Prolog would do the unifications and then start the leftmost call to `cell`, that is it would try to obtain the evolution of the cell 2. In the first step, Prolog would be able to establish that the cell 2 will be `off`. In the second step, Prolog would not know the state of the cells 1 and cell 3, thus it will need to guess for cell 2. The guessing will be repeated for the next steps. This guessing will be expensive and unnecessary, as the game of life is a determinate process.

Executing this program in both the Basic Andorra Model and the committed-choice languages will generate a very different behaviour. When `life` is executed, all the unifications and the calls to `cell` become available for execution. All of them are determinate (or would be able to commit in committed-choice language notation), so any call can execute. Notice that calls to *cell* will always be determinate, but calls to `event` will only be determinate when both neighbours are instantiated (that is, if the previous step has been performed on them). But all these neighbours will eventually instantiate their variables; thus all the `event` goals will become determinate, and Andorra will do the whole computation determinately.

The coroutining inherent to the Basic Andorra Model is also advantageous in search problems.

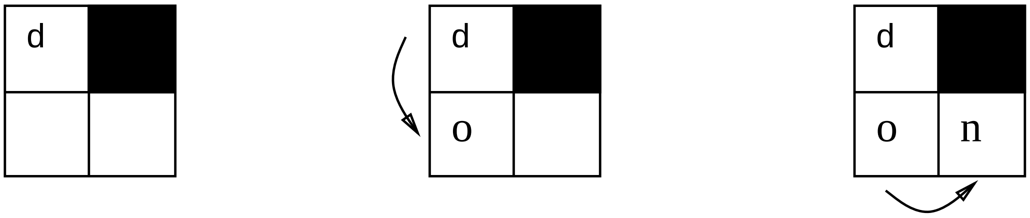

Figure 8.2: Crossword Puzzle Solution

A simple example is the crossword generation program based on Van Hentenryck [171]). The generator uses a database of words, in this case "on", "do", "d" and "o", and the goal is to distribute them across the crossword board. The board is represented as a rule and words are represented as facts. Figure 8.2 shows how these words are distributed to form a puzzle.

```
board(V1, V2, H1, H2) :-
```

```
        size(V1, 2), size(V2, 1),
        size(H1, 1), size(H2, 2),
        letter(1, V1, A), letter(2, V1, B), letter(1, V2, C),
        letter(1, H1, A), letter(1, H2, B), letter(2, H2, C),

letter(1, on, o).             letter(1, do, d).
letter(2, on, n).             letter(2, do, o).
letter(1,  d, d).             letter(1, n, n).

size(on, 2).                   size(do, 2).
size(d, 1).                    size(n, 1).
```

The rule for `board` establishes the relations between all the cells of the crossword puzzle. The facts for `letter` simply establish that the third argument is the character occurring in the word given by the second argument at the position given by the first argument. The facts for `size` give the size of each word.

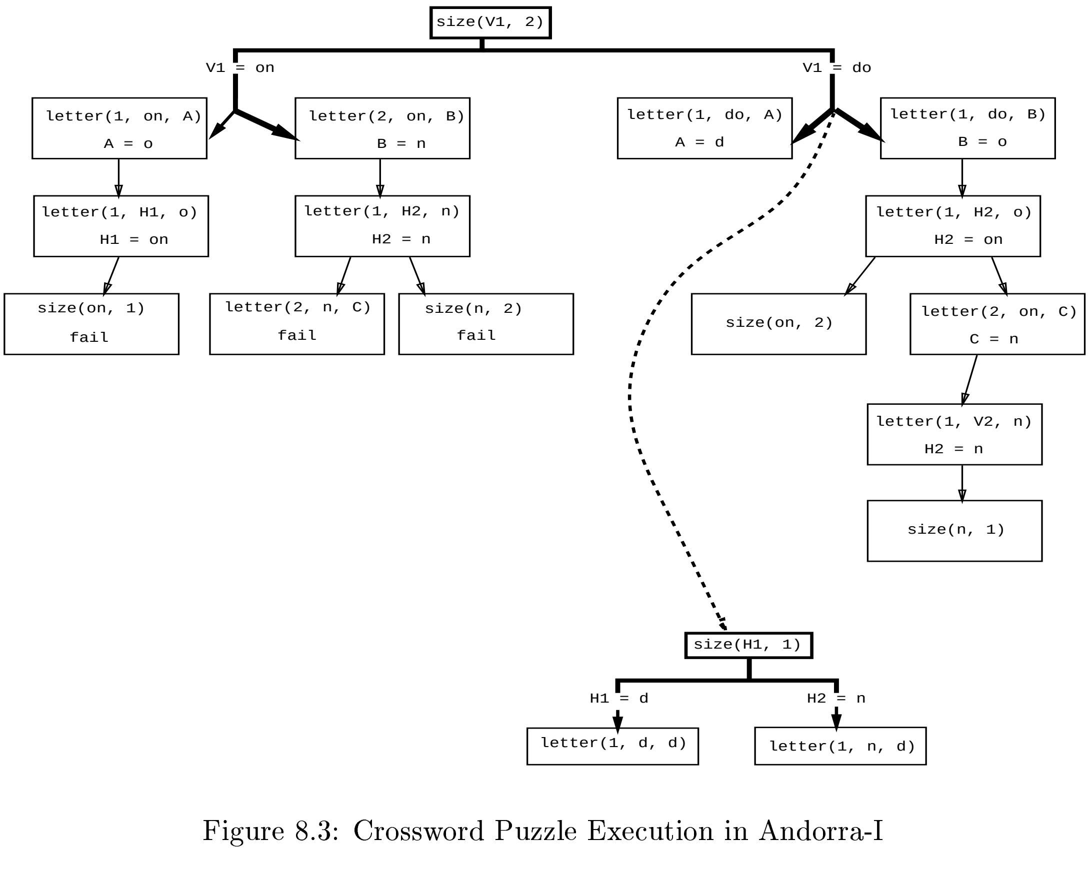

Figure 8.3: Crossword Puzzle Execution in Andorra-I

Figure 8.3 shows the Andorra execution. Each box shows one reduction, lighter boxes show determinate reductions, darker boxes nondeterminate reductions. Andorra will first make a choice on, say, the leftmost `size` goal. The choice can be between one of two facts, `size(on,`

2) and `size(do, 2)`. Selecting the fact `size(on, 2)` makes the two goals `letter(1, V1, A)` and `letter(2, V1, B)` determinate, which in turn make the goals `letter(1, H1, A)` and `letter(1, H2, B)` determinate, which in turn make the goals `size(H1, 1)` and `size(H2, 2)` and `letter(2, H2, C)` fail. In contrast, selecting the fact `size(do, 2)` will again make the goals `letter(1, V1, A)` and `letter(2, V1, B)` determinate. Notice that, whereas binding the variable `B` makes a further goal determinate, the binding for `A` is unable to make the goal `letter(1, H1, A)` determinate as two clauses match this goal `letter(1, H1, d)`. Eventually execution obtains determinate bindings for the variables `H2` and `C`, and makes a choice to obtain the bindings for `H1`.

The program would deadlock in the committed-choice languages because some choices must be made by one of the `size` goals. Prolog would blindly try all the alternatives for the first `size` goal, then for the second `size` goal, and so on. Figure 8.3 shows the Andorra execution. Andorra-I still needs to try the alternatives for the first goal in table, `size(V1, 2)`, but the search is better informed. For instance, after choosing `V1 = do`, Andorra-I determinately infers the bindings for the variables `B, H2, C`, and `V2`, and only needs to make a choice for `H1`. The end result is that Andorra-I has to perform much fewer reductions.

The coroutining in Andorra-I gives several advantages over Prolog:

- Because it executes the determinate goals first, Andorra-I can reduce the number of alternatives to try.
- Determinate goals need to be tried only once, rather than re-executed at different branches of the search space.
- Andorra-I is not sensitive to the textual order of determinate goals, although it still cares about the textual order of nondeterminate goals.

There has been much interest in applying the Andorra selection function as a better strategy for search problems. Yang showed how the coroutining of Andorra's execution can be exploited in cryptography applications [190]. These advantages are one of the motivations for the language Pandora [4], that uses the Basic Andorra Model to extend the PARLOG language. Section 11.1.3 contains a brief description of Pandora. Pandora programs show examples where the early execution of determinate goals gives some (though not all) of the advantages of constraint logic programming languages, whilst remaining in the Horn clause framework.

8.2 Andorra-I and the Committed-Choice Languages

We have previously shown that some programs written for the committed-choice language can run easily in Andorra-I. In fact, and while running determinate goals, Andorra-I's execution mechanism is similar to the one used by the committed-choice languages. We would like to take advantage of this to run the applications developed for the committed-choice languages, but taking care not to make Andorra-I just a hodge-podge of all logic programming languages available.

We discuss the main features of the flat committed-choice languages individually. Note that there are quite a few committed-choice languages, with different features. Important differences are on the conditions to commit to a clause, and on when to perform output unification. To simplify our discussion, we compare Andorra-I Prolog with Flat PARLOG, taking care to remind the reader that many of the features we discuss for PARLOG are common with most other committed-choice languages.

Both Flat PARLOG and Andorra-I Prolog can use flat commit to enable commitment to a clause. Whereas in Andorra-I Prolog, commit applies as soon as it is quiet, in PARLOG a goal can commit to a clause only if head unification and goals in the body of the clause can execute without binding external variables. The two commits are therefore *not* equivalent. For instance, a program which may work in Andorra-I may deadlock in Flat PARLOG. Conversely, a program which may terminate in Flat PARLOG, may run forever in Andorra-I. Consider the simple program:

```
mode strict(?).

strict(any) <- strict(Y).
```

This program will deadlock in PARLOG, but will start an infinite loop in Andorra-I. Programs may also succeed in Andorra-I but deadlock in PARLOG:

```
mode eager(?).

eager(any).
```

The PARLOG definition of commit gives more direct control over when to execute goals, whereas the Andorra-I definition arguably agrees more with the notion of commit as a pruning operator designed to discard uninteresting paths in the search tree.

Both Andorra-I Prolog and PARLOG support sequential conjunction, albeit with a different syntax. PARLOG supports sequential-search, that means that some clauses may only be tried if other clauses fail (other languages use `otherwise` for similar purposes). Most (but not all) of the functionality of sequential-OR is provided by cut in Andorra-I.

Finally, PARLOG uses different meta-predicates than Prolog. One example is the built-in `data/1`. This is similar to `nonvar/1`, but will wait until its arguments are bound. If the program does never deadlock, Andorra-I will only select determinate goals, and therefore execute `nonvar/1` as `data/1`. On the other hand, PARLOG also offers the built-in `var/1`, that succeeds immediately if its argument is a variable. Clearly, programs that use this form of `var/1` depend on the committed-choice model of execution. The Andorra-I system does provide a low-level builtin to support this form of `var/1`, but as they assume committed-choice style execution they are incompatible with Prolog, and therefore Andorra-I could only support them either through a different preprocessor, or through direct access to the low-level facilities of the system.

Committed-choice languages include other features such as atomic unification, read-only variables, meta-calls and pragmas for work management, with different languages supporting quite different features. Trying to support all the features of all these languages would be impractical, and would make our original goal to support Prolog more difficult. We therefore decided to support in Andorra-I Prolog only the features that fit well with our goal of supporting Prolog. In practice this means that most flat programs that use commit, the sequential-OR and sequential-AND, and some meta-predicates, will run also as Andorra-I Prolog programs. If meta-predicates such as the parallel `var/1` are used, one currently has to use the low-level features available in the Andorra-I Target Language.

8.3 Andorra-I and Extensions to Prolog

Several languages, such as NU-Prolog or Prolog-II, extend Prolog with facilities for corouting. One quite important facility is the delay declaration used in MU-Prolog, variants of which have since been used in several Prolog systems. One example of a delay declaration is:

```
name(Atom, List) when Atom or ground(List).
```

The declaration means that a call to `atom/2` in NU-Prolog can only execute if either its first argument is bound, or if its second argument is fully instantiated. Note that execution of programs with delayed goals is quite different from left-to-right execution. For instance, programs for these languages may flounder, that is, deadlock, if the conditions for some goals are never fulfilled. NU-Prolog supports Prolog features such as cut, and problems also arise when sensitive goals interact with delayable goals. The programmer is expected to avoid any possible interactions, mainly by avoiding cut.

The Andorra-I engine provides the basic machinery necessary to support delay declarations. Given this support, we can run most programs with delayable goals in Andorra-I, with the exception of *delayable goals that are sensitive or indirectly sensitive.* To explain why, consider the following program.

```
:- d(X, Y) when X.

g(X,Y) :-
  d(X,Y),
  X = a,
  Y = b.

d(X, Y) :- var(Y).
```

and the query `g(X,Y)`. The answer to this query depends on whether `Y` was bound before `d/2` was woken up, and it is not even clear which answer should be given by Prolog. For instance, interpreted SICStus Prolog will give a different answer to `g(X,Y)` than compiled SICStus Prolog (the problem arises because the compiler optimises unification and the two goals `X = a, Y = b` are executed as a single goal by the compiler).

In general delaying sensitive goals is a bad idea, even for traditional Prolog systems, and there is no point in supporting this in Andorra-I.

We next discuss how the preprocessor can support Prolog programs extended with delay declarations, assuming that we never delay sensitive goals. We show how to implement sequencing for such programs, and how the engine can support delay declarations.

- Prolog programs with delay declarations still need sequencing, as non-delayed goals obey the same restrictions as before. Consider now a sequenced program:

  ```
  g :- a(X, Y) :: d(Y, X).

  a(X, Y) :- b(X), c(Y).
  ```

```
:- b(X) when X.

c(Y) :- var(Y).
```

The call `b(X)` is for a delayed goal, which can only execute after some goal (in this case `d/2`) binds the variable `X`, and `c/1` is a sensitive goal. Clearly, `c/1` must be executed before `d/2`, but the sequential conjunction also seems to imply that `b/1` can only be executed after `d/2`.

In fact, the sequential conjunction is only needed because `c/1` is sensitive. The correct sequencing is to allow `d(X,Y)` to execute as soon as as `c(X)` executes.

The solution to this problem is to observe that if delayable goals are not constrained by any ordering, there is no advantage in waiting for them. Therefore, the sequential conjunction should not wait for delayed goals. In the example, `d/2` will only wait for `b/1`, observing the correct semantics.

- The Andorra-I engine does not directly support delaying of goals, but it does allow for a goal to be delayed until determinate, through the low level declaration `det_only`. We next show how we can use `det_only` together with the pruning operators to obtain the functionality of delay declarations

  We give an example for a `freeze/2` builtin (originally from Prolog-II [33]), that delays execution of a goal in the second argument until the first argument is bound. Code for `freeze/2` in Andorra-I with `det_only` is:

```
:- det_only freeze/2.

freeze(V, G) :- nonvar(V), ! , call(G).
freeze(_, _).
```

  This code will execute when the first argument is bound. Two clauses are needed because otherwise `freeze` will always be determinate.

  The NU-Prolog wait declarations for the version of `name/2` we have shown before can be implemented through commit and the declarations.

```
:- det_only name/2.

name(Atom, List) :- nonvar(Atom), | , name_Code(Atom, List).
name(Atom, List) :- ground(List), | , name_Code(Atom, List).
```

Finally, it is important to remark that we can use these principles to support constraint languages, such as CHIP or the CLP languages. Constraints can be considered as delayable goals, that are executed by the constraint solver. Andorra-I does include a finite-domains constraint solver, and Gregory and Yang describe the advantages of constraint style programming in Andorra-I [59].

8.4 Andorra-I Prolog

We conclude by summarising Andorra-I Prolog. Andorra-I Prolog is an extension of Prolog:

- That therefore supports Prolog programs;
- That can use coroutining to improve the search space of programs, and to run committed-choice applications

Andorra-I Prolog does introduce new features:

- Mode declarations, that are similar to the modes declarations found in other Prolog systems.
- The sequential conjunction, that guarantees left-to-right execution between some goals
- Quiet cuts and quiet commits, that provide a safer way to prune. The user can declare that a cut or commit is quiet, or the system can try to deduce it by compile-time analysis.
- Extensions to support constraint systems, currently finite-domain variables in the CHIP style [47]. Andorra-I Prolog can also support delay declarations.

Coroutining and quiet pruning operators are the most important extensions to Prolog. Further examples of the performance advantages of coroutining are shown in section 10.5.

As regards quiet pruning, the main advantage of quiet pruning is that guarantees that the same solutions will be pruned in the presence of coroutining. The benefits of quiet pruning therefore apply not only to Andorra-I, but also to other languages that use coroutining.

8.5 Summary

In this chapter we presented the advantages of the coroutining possible with the Basic Andorra Model. We showed that Andorra-I can give much of the functionality of committed-choice languages. We also showed that Andorra-I Prolog can support previous coroutining extensions to Prolog systems, and also constraint logic programming extensions.

Chapter 9

The Determinacy Analyser

To obtain coroutining and and-parallelism, Andorra-I must be able to detect if goals are determinate. This is the task of the determinacy analyser. For pure Prolog procedures, a goal is determinate when either a single clause matches the goal, or when the goal must fail. If the procedure includes cuts or commits, the pruning operators may make the goal determinate by pruning all but one matching clause.

The chapter first discusses the determinacy problem. Then follows a description of the determinacy algorithms used in the preprocessor. A suggestion for further improvements and a comparison with systems tackling similar problems is also made.

9.1 Detecting Determinacy

Determinacy for a pure Prolog procedure results from having at most one clause to match the input. A straightforward algorithm to verify if a call is determinate at run time is:

1. Test all clauses in the program. If only one matches the input goal, output a *commit* to that clause. If none matches, output that the goal *fails*. Otherwise go to step two.
2. *Wait* until a variable appearing in the call is instantiated. Go to step one.

We still need to address what is meant by "testing" a clause. A discussion on how much work should be tried to determine the determinacy of goals is part of section 4.2.1. The conclusion was that efficient testing can be implemented by verifying only the heads of clauses and builtins in the bodies of clauses (the so-called *flat determinacy*). Furthermore, it is simpler to make the algorithm read-only, that is, to avoid binding external variables during determinacy analysis. The same reasoning as for the committed-choice languages applies here: (i) binding external variables is harder to implement because each clause will need to have its own environment, as the bindings should only be available to other goals after committing to a clause; (ii) binding external variables can only make a goal determinate in the cases where the same external variable occurs at several places in the goal, as otherwise binding the external variable will succeed for every clause. These are the definitions we follow in this chapter.

The algorithm presented before is quite inefficient because whenever a variable in the call is instantiated, all clauses would be tried anew. It is possible to improve on this, for example by remembering if a clause has been shown not to match. Further optimisations are possible, but the resulting algorithms will still force to test all the clauses one by one.

We decided to implement an alternative algorithm for detecting determinate calls, based on the observation that a clause becomes the only solution to a goal because an argument or group of arguments becomes instantiated. Moreover, for each argument, we can detect if a clause is a single solution by extending the indexing algorithm used in the implementation of Prolog. The basic idea of the algorithm is thus to generate at compile-time a decision graph, in our case formed by an "or" of *decision trees* corresponding to each argument. We shall call the decision graph for the detection of determinacy a *determinacy tree*.

It is interesting to compare the complexity of both approaches. In the first case, the basic operation is comparing the goal with the head of a clause. According to the previous discussion, it can be implemented as a read-only unification, and therefore takes time proportional to the size of the input goal. Execution of builtins in the body of the clause will usually take constant time. The worst case time for the naïve algorithm is then:

$$\mathcal{T} \qquad \alpha \qquad n * v * Size(G)$$

where $\mathcal{T}$ is the actual time we need to prove (or disprove) determinacy, which is proportional to the product of n, the number of clauses, v, the number of variables that were instantiated and forced a retry of the goal, and $Size(G)$, the size of the goal (see section 2.1 for a definition of size of a term). Note that the size of a goal will grow as the goal is further instantiated, but it is always smaller than the size of the goal when most instantiated. We also know that the actual number of variables in the goal is always less than the size of the goal, hence v must be less than the size of the goal. We thus have the following approximation:

$$\mathcal{T} \qquad \leq \qquad \mathcal{K} * n * Size(G)^2$$

This tells us that, in the worst case, the naïve algorithm takes time proportional to the number of clauses and to the square of the size of the input goal, or in other words it is $O(n * Size(G)^2)$.

Complexity analysis for a problem related to generating determinacy code, namely the problem of indexing using a minimal number of parameters and not needing to use choicepoints, was made by Hickey and Mudambi [78]. They prove that the problem is NP-complete. Palmer and Naish [124] also give an elegant demonstration that the actual problem of generating a complete determinacy tree is NP-hard, resulting in large increases for compilation time and code size.

The combinatorial explosion we mentioned before arises only in a few cases, mainly when a goal becomes determinate through variable combinations of several arguments, or through connections between subarguments of different structures. In most applications, determinacy is easily detected from looking at a single arguments or at a simple combination of arguments, and considering complex combinations of arguments would only result in more complex code that can actually slow-down run-time execution. To obtain a simpler and more practical system, the preprocessor generates code that detects the more important cases of determinacy, but that may ignore some determinate goals. For most programs this seems to be quite sufficient.

9.2 Determinacy Code for Pure Prolog

In Figure 9.1 we show an example of how the preprocessor compiles determinacy code for a small database.

```
p(a, a, a).
p(b, a, a).
p(a, b, a).
p(a, a, b).
```

The figure shows the determinacy tree for the procedure. The dashed lines represent choices for arguments, that is how different arguments can be used to detect determinacy, and the arrows represent choice on values of an argument. Choices on values select a clause or set of clauses based on some value. Notice that choices for values are exclusive. In contrast, arguments are not exclusive and at run-time several arguments may be tried simultaneously. For instance, as soon as any of three arguments becomes instantiated, the engine will try the corresponding decision tree. The example also shows a situation where the analyser needs

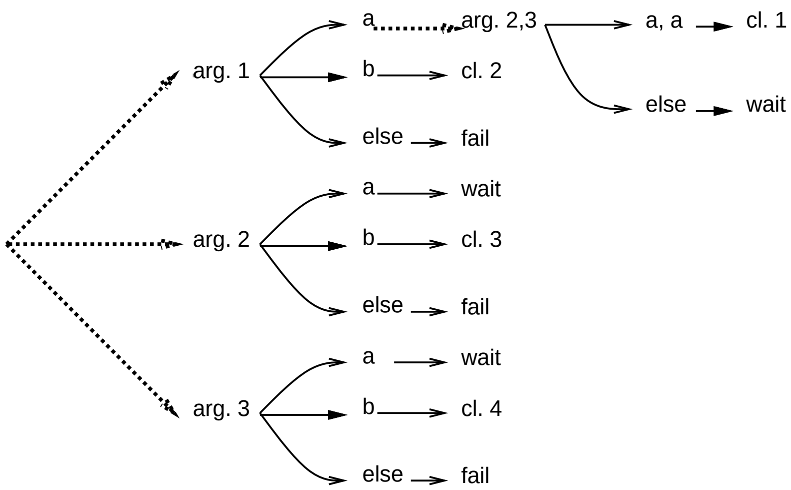

Figure 9.1: A procedure and corresponding determinacy code.

to expand an argument's decision tree, in this case because in order to verify if the goal `p(a, a, a)` is determinate we need to test all its arguments. Notice that testing two arguments together does not make any clauses goals determinate (either goals are determinate because of a single argument or because of the three arguments) and therefore no specific code was generated for them. Finally, the combination of the three arguments that is needed to verify that the goal `p(a, a, a)` is determinate might have been called from the trees for any argument, but the analyser only associated it with the tree for argument one. All these optimisations are necessary to constrain the size of the output code and avoid duplication of work.

The exits of the decision graph correspond to either verifying that the call should *fail*, or that it should *commit* to a clause, or that it should *wait*. The last case means that there is no way of making the call determinate, or some other alternative argument should be used to verify if the clause is determinate (the case in this example).

The actual algorithm used by the analyser is made more complex by the need to handle compound terms, full unification and builtins. The resulting algorithms and the actual instruction set use many of the ideas originally developed in the contexts of indexing in Prolog [178, 173] and decision trees used in compiling committed-choice languages [91]. The algorithm proceeds in three steps. First, it generates a decision tree for each argument. If all the tips of an argument's decision tree correspond to committing to a clause or failing, then the argument provides a way to find if the clause is determinate, the tree is satisfactory and compilation for that argument may terminate. Otherwise the analyser will try to expand the remaining tips by looking at other arguments, until either the tree is satisfactory or no more arguments are available. Finally, the actual code is generated in the classical way.

We next discuss each step in detail.

9.2.1 Generating the Argument's Decision Tree

The algorithm does not generate decision trees for each argument directly. Instead, the algorithm first finds out for which clauses and in what way an argument alone can make the clauses a single solution. It does that in a way similar to the algorithms used to generate decision trees for FCP [91], and we thus use a similar notation to describe the variables used by the algorithm.

The first step is *normalisation*. Basically, normalisation replaces the head of a clause by a trivial head and a set of equality tests or builtins. Equality tests are of the form $X = T$, where X is a variable and T a *simple term*, i.e., a variable, a constant or a compound term in which the arguments are pairwise different variables. To these tests are then added all the tests corresponding to builtins in the body. Currently the system considers the following builtins as tests:

arithmetic comparisons: corresponding to the builtins `=:=`, `<`, `>`, `<`, etc;

term comparisons: corresponding to the builtins `==`, `@<`, `@>`, etc;

type builtins: corresponding to builtins `integer`, `atom`, etc.

We shall call these builtins *tests*. The preprocessor only considers tests placed before a pruning operator.

The result of normalisation is a canonical-form procedure, consisting of canonical-form clauses each of the form $\langle i, G \rangle$ for clause number i with guard G, with guard G being a set of tests (or a conjunction of literals). The head arguments are always named $Z_1, \ldots, Z_n$ for a procedure of arity n. Finally, variables occurring in a position j of a compound term occurring inside some compound term inside the argument i is named $Z_{i,\ldots,j}$. For example, the clause:

$$a(a, b(B, c(C))).$$

would be transformed into the canonical-form clause:

$$a(Z_1, Z_2) \quad :- \quad Z_1 \ = \ a, Z_2 \ = \ b(Z_{2,1}, Z_{2,2}),\ Z_{2,2} \ = \ c(Z_{2,2,1}).$$

We also need to establish a partial order among variables appearing in the tests. Intuitively, if a variable represents a compound term, the variables representing the subterms will be after the variable representing the full term. We call the relation $\geq_\psi$ (for example, $Z_{2,2} \geq_p siZ_2$).

Arguments and Tests: Given an argument j and a clause $C = \langle i, A \rangle$, we say that a test g in A belongs to the argument j if and only if it is a function of Z_j or if there is a test g' such that g' belongs to j and $g \geq_\psi g'$, that is:

$$g \quad \in \quad \langle j, i, A \rangle \quad \equiv \quad (g \in A \wedge (g(\ldots, Z_j, \ldots) \vee (\exists g(\ldots, Z_k, \ldots) \in A \wedge Z_k \geq_\psi Z_j)))$$

Furthermore, we say that a set of tests G from a clause A belongs to j if every test g in A belongs to j. The set of all these tests is denoted $\langle j, i, A \rangle$, where j is argument number and i and A represent the clause. For example, for the procedure shown next:

```
a(X, f(a)) :- atom(X).
a(X, f(b)) :- integer(X).
```

We obtain the following sets:

$$\begin{array}{lcl}
\langle 1, 1, \{atom(Z_1), Z_2 = f(Z_{2,1}), Z_{2,1} = a\} \rangle & = & \{atom(Z_1)\} \\
\langle 2, 1, \{atom(Z_1), Z_2 = f(Z_{2,1}), Z_{2,1} = a\} \rangle & = & \{Z_2 = f(Z_{2,1}), Z_{2,1} = a\} \\
\langle 1, 2, \{integer(Z_1), Z_2 = f(Z_{2,1}), Z_{2,1} = b\} \rangle & = & \{integer(Z_1)\} \\
\langle 2, 2, \{integer(Z_1), Z_2 = f(Z_{2,1}), Z_{2,1} = b\} \rangle & = & \{Z_2 = f(Z_{2,1}), Z_{2,1} = b\}
\end{array}$$

Notice that the same test may belong to several arguments: a test such as, say, an equality test, may use several different arguments.

Exclusive tests Following Kliger, we say that, for two tests g and g', g implies g', or $g \Rightarrow g'$, if for every substitution θ, $g\theta$ is true implies that $g'\theta$ is true.

We say that two tests g and g' are *exclusive*, or $g \perp g'$, if there is no substitution θ such that both $g\theta$ and $g'\theta$ are true.

Exclusive sets of tests We generalise for sets of tests by saying that two set of tests $G = g_1 \ldots g_n$ and $G' = g'_1 \ldots g'_m$ are exclusive, that is, $G \perp G'$ if there is no substitution θ such that both $g_1\theta \wedge \ldots \wedge g_n\theta$ and $g'_1\theta \wedge \ldots \wedge g'_m\theta$ are true.

One important property of exclusive sets of tests if that if we add tests to a set G exclusive to some other set G', then the union will still be exclusive to G', that is:

$$G \perp G' \Rightarrow G \cup A \perp G'$$

This property holds because we assume tests are only executable when sufficiently instantiated.

For an argument j, given two clauses C_{i_1} and C_{i_2} and the corresponding belonging test sets $\langle j, i_1, A_1 \rangle$ and $\langle j, i_2, A_2 \rangle$ we say that the clauses C_{i_1} and C_{i_2} are exclusive for the argument j, $C_{i_1} \perp_j C_{i_2}$ if and only if the corresponding tests sets are exclusive.

Note that if the condition $\perp_j$ holds between two clauses for some argument j, then by sufficiently instantiating the argument the two clauses will exclude each other. E.g., in the previous example both the first and second clauses are exclusive for both arguments ($C_1 \perp_1 C_2$ and $C_1 \perp_2 C_2$). This means that by sufficiently instantiating either the first or the second argument only one clause will match.

Minimal Sets of Exclusive Tests Given that two clauses exclude each other for an argument j, it is of interest to know which tests are responsible.

To do so, we define *minimal exclusive set of tests.* Given an argument j and two clauses $C_{i_1} = \langle i_1, A_{i_1} \rangle$ and $C_{i_2} = \langle i_2, A_{i_2} \rangle$ the set of tests G_{i_1} is said to be a minimal set of exclusive tests for that argument and those clauses if and only if:

$$\begin{array}{c} G_{i_1} \subseteq A_{i_1} \\ \wedge \\ G_{i_1} \perp A_{i_2} \\ \wedge \\ \neg(\exists G'_{i_1} \mid G'_{i_1} \subset G_{i_1} \wedge G'_{i_1} \perp A_{i_2}) \end{array}$$

In relation to the previous example, the minimal sets of exclusive tests for the first argument, in relation to clause 1 and 2, are:

$$G_{1_1} = \{atom(Z_1)\}, G_{1_2} = \{integer(Z_1)\}$$

There are also minimal exclusive sets for the second argument, which are:

$$G_{2_1} = \{Z_2 = f(Z_{2,1}), Z_{2,1} = a\}, G_{2_2} = \{Z_2 = f(Z_{2,1}), Z_{2,1} = b\}$$

Notice that there may exist several solutions for the same argument. We next give an example:

```
b(f(a, b), _).
b(f(d, c), _).
```

In this case, there are two minimal sets of exclusive tests for each clause, which are:

$$G_1 = \{Z_1 = f(Z_{1,1}, Z_{1,2}), Z_{1,1} = a\} \quad G_2 = \{Z_1 = f(Z_{1,1}, Z_{1,2}), Z_{1,1} = d\}$$

$$G_1' = \{Z_1 = f(Z_{1,1}, Z_{1,2}), Z_{1,2} = b\} \quad G_2' = \{Z_1 = f(Z_{1,1}, Z_{1,2}), Z_{1,2} = c\}$$

In both cases, the tests in G_1 can exclude the second clause and the tests in G_2 can exclude the first clause.

Residual A clause is said to be a single solution to a goal if matching the clause implies that all other clauses fail. For an argument j to make a clause single solution it is necessary that the tests in this clause will be exclusive with the tests in every other clause. We use the notion of `residual` of a clause in relation to an argument to define this.

The residual of a clause in relation to an argument, $\mathcal{R}_j(C_i)$ is the set of all the clauses such that there is no minimal set of exclusive tests for that argument. If $\mathcal{R}_j(C_i)$ is empty we say that the argument j makes the clause C_i single-solution. If:

$$\forall i, \qquad \mathcal{R}_j(C_i) = \emptyset$$

that is, if the argument j makes all clauses single-solution, we say that the argument is *sufficient* for determinacy. In contrast, if,

$$\forall i, j \neq i, \qquad C_j \in \mathcal{R}_j(C_i)$$

that is, if there is no case when two clauses are mutually exclusive for the argument j, we say the argument is *useless* for determinacy.

In the previous examples, for the procedure `a/2`, both the first and second argument are sufficient for determinacy, whereas for the procedure `b/2` the first argument is sufficient for determinacy, but the second argument is useless.

9.2.2 The Determinacy Algorithm for Individual Arguments

The first step of the determinacy algorithm is to create an array indexed by argument and clause numbers. For each entry of that array, the determinacy algorithm stores (i) the residual for that clause and argument and (ii) the union of all the minimal sets of exclusive tests between the clause and all the other clauses. Note that if all the tests in the union are satisfied, then only the clauses in the residue can succeed.

Before we describe when arguments are combined, it is important to explain how one verifies if two clauses are exclusive for some argument.

Adding Tests to Compare Clauses

Suppose we are looking at the argument j and comparing two clauses, C_1 and C_2. We have found that the current tests G_1, G_2 are insufficient, and we need some more tests to make the two clauses mutually exclusive.

The tests will have involved the variables $Z_j, \ldots, Z_{j,\ldots,k}$. with the usual order $\geq_\psi$ between variables. We classify tests into three categories. *Simple* tests correspond to building a term, and are of the form $Z = a$ or $Z_j = f(Z_{j,1}, \ldots, Z_{j,n})$, *unary* tests correspond to tests with only one argument (such as `atom(X)`), *binary* tests to tests with two arguments. In general, unary and structure tests are easier to analyse, compile and to execute, and the algorithm tries them first.

Our algorithm expands two sets of tests G_1 and G_2 in clauses C_1 and C_2 so that the two sets will become exclusive. To do so, it uses four sets of tests G_1, G_2, NG_1 and NG_2, where NG_1 are the tests from A_1 not in G_1, and similarly for NG_2, and two sets of variables $\mathcal{Z}_1$ and $\mathcal{Z}_2$. It also uses two sets of variables $\mathcal{Z}$, such that Z is in a set $\mathcal{Z}$ if $Z \geq_\psi Z_i$ and Z appears as an argument of a $g \in G$. The set $\mathcal{Z}_1$ corresponds to the variables from G_1 and the set $\mathcal{Z}_2$ to the variables from G_2.

The algorithm tries first to expand G_1 and G_2 with simple tests, by considering variables Z in the $\mathcal{Z}_1$ such that there is no Z' for which $Z \geq_\psi Z'$. If a simple test is found for one such Z, the test is removed from the NG_1 and added to G_1, and the variables appearing in the the right-hand side of the test are added to $\mathcal{Z}_1$. If G_1 and G_2 are still not exclusive, the algorithm tries to fetch a simple test for Z from NG_2 to G_2. If there are no such tests the algorithm backtracks and tries a different Z from $\mathcal{Z}_1$. If there is one such test but G_1 and G_2 are still compatible, the algorithm calls itself on the new arguments. When no more simple tests remain, the tries unary tests, using a similar technique. If no more unary tests are available, the algorithm tries binary tests.

This algorithm does not necessarily give a minimal set of tests (because it always uses compound tests first), but it gives the tests that are more efficient to try first.

Verifying if Two Tests Are Exclusive

To implement the previous algorithm we also need a function to verify whether two sets of tests are exclusive. The preprocessor uses an inference mechanism to implement it. This mechanism tries to find one value that satisfies both sets of tests, and if there is no such value, it succeeds.

In general, the problem of finding whether two sets of tests are mutually exclusive is equivalent to the problem of determining a solution to a general set of Prolog primitives and thus insolvable. In fact, some measure of completeness can be obtained for Herbrand tests, but the introduction of arithmetic tests does limit completeness. In the worst case, to find if the arithmetic tests exclude a clause, one may have to solve arbitrarily complex problems, such as the Fermat problem [78]:

```
fermat(X, Y, Z, yes) :-
        integer(X), integer(Y), integer(Z), integer(N),
        X > 0, Y > 0, Z > 0,
        X^N + Y^N =:= Z^N, N > 2.
fermat(_, _, _, no).
```

The first clause should always fail because Fermat's conjecture, that there is no integer only solution to the equation $X^N + Y^N = Z^N$, is true.

In the preprocessor we were interested on finding practical heuristics that approach the problem. These heuristics are not necessarily complete but should handle most of the practical cases.

Simple tests can be compared by simply applying them to two initial terms and then verifying if they are still unifiable.

Unary tests are somewhat more complex. After both sets of simple tests are applied, unary tests are at first tried on the resulting term. If they succeed, then the unary tests from both sets are compared. For example, if one set of tests includes $atom(X)$ and the other $integer(X)$, the sets are exclusive.

The most complex tests to support are the binary tests. They include equalities, that is tests of the form $X = Y$, and more general comparisons. If both arguments of the tests are instantiated, then it is possible to decide if they are exclusive. Otherwise, the system resorts to a set of heuristics which try to process the most common cases. These cases include comparisons resulting in non-intersecting ranges of possible values for the same variable and different tests with the same arguments.

Obviously, all these heuristics are not complete. For example, transitivity of tests is not handled by the current rules, as this example from Korsloot and Tick [94] shows:

```
a(X, Y) :- X < Z.
a(X, Y) :- X > Y, Y > Z.
```

The example is practice handled by the preprocessor's normalisation algorithm.

Even when two sets of tests are not always exclusive, it would still be of interest to study when the two sets of tests are mutually exclusive. The current preprocessor does not include this feature, mainly because it would be complex to implement and could result in a much larger determinacy code.

9.2.3 Combining Arguments

If all the arguments are either sufficient for determinacy or useless for determinacy, there is no point in combining several different algorithms. Otherwise, more determinate goals may be found by combining several arguments. Combining arguments will be useful if it does provide new information, that is if by combining arguments either some clause will become exclusive with all other clauses, or no clause will match. Moreover, combining arguments is only useful if it is not tried somewhere else before.

We next present an algorithm to find combinations of arguments that detects whether clauses can become a single solution to a goal, through looking at multiple arguments. The algorithm works on a set of combinations of arguments that if expanded may make more clauses a single solution.

The algorithm uses the following variables: As and As_0, the set of argument combinations to expand, and NAs, the set of argument combinations that should not be further expanded. The algorithm uses a store $\Gamma(\langle j \rangle)$, such that for an argument or combination of arguments:

$$\Gamma(\langle j \rangle) = \{\langle i_1, i_2 \rangle \mid \neg(i_1 \perp i_2)\}$$

That is, the stores hold all pairs of clauses that are not exclusive for that argument or combination of arguments.

- Initialisation: we set NAs to be the set of single arguments that are either sufficient or useless for determinacy, and $As = As_0$ is set to the remaining arguments.
- While As is not empty:
  - Select one $x = \{a_1, \ldots, a_{j-1}\}$ from As; $As := As \backslash \{x\}$;
  - For every element a_k of As_0 not appearing in x, form a $y = \{a_1, \ldots, a_{j-1}, a_k\}$ and:
    1. if a subset of y appears in NAs, skip the next steps: notice that only the sets whose last element is a_k need to be tested;
    2. $\Gamma(y) := \Gamma(x) \cap \Gamma(\{a_j\})$;
    3. if $ClausesIn(\Gamma(y)) < ClausesIn(\Gamma(x))$ then store y as a solution for the clauses made single solution
    4. if $\Gamma(y) = \emptyset \vee \Gamma(y) = \Gamma(x)$ then $NAs := NAs \cup \{y\}$ else $As := As \cup \{y\}$: either the combination needs to be further expanded, or there is no point in expanding it.

The main goal of the algorithm is to prune combinations of arguments as much as possible. To do so, the algorithm uses the set NAs to store all useless or sufficient combinations of arguments, and tests any new combinations with this set. Still in the worst case it is possible that all combinations are useful, and that the algorithm may generate all the 2^{arity} combinations. This results from the exponential nature of the problem, and is avoided in the preprocessor by providing a depth bound on the number of arguments to combine.

The function $ClausesIn$ gives all the clauses that are still not determinate. It can be defined quite simply as:

$$ClausesIn(S) = \begin{cases} \{i, j\} \cup ClausesIn(S') & S = S' \cup \{i, j\} \\ \emptyset & S = \emptyset \end{cases}$$

This function is used in step 3 to verify if intersecting more arguments makes any more clauses a single solution. If so, the combination is immediately stored in a table and associated to the clauses. This table associates clauses to combinations of arguments that make them determinate and is the actual output of the algorithm.

Example We next give a small example of the algorithm, with the procedure:

```
c(1, 1, 1, _).
c(1, 2, 2, _).
c(2, 2, 3, _).
```

The third argument is sufficient for determinacy, and the fourth is useless, therefore the initial values for As is $\{\{1\},\{2\}\}$ and for NAs is $\{\{3\},\{4\}\}$. We first select $x = \{1\}$ and expand it to form the sets $\{1,2\},\{1,3\}$ and $\{1,4\}$. We first try $y = \{1,2\}$.

No element of y appears in NAs, thus we can calculate a new store. This gives $\Gamma(y) = \Gamma(x) \cap \Gamma(\{2\}) = \langle 1,2\rangle \cap \langle 2,3\rangle = \emptyset$, and $ClausesIn(y) = 0$. Previously, for the first argument only the third clause was a single solution, now all the clauses will become a single solution.

Finally, NAs :=$\{\{3\},\{4\}\} \cup \{\{1,2\}\}$, and $As := \{\{2\}\}$. All remaining combinations of arguments are disabled by step 1 and the algorithm terminates with the result that when looking at the first and second clauses from the first argument we should proceed to look further at the second argument.

Ligatures

In step 2 it is implicit that intersecting the non-excluding clauses as calculated in the first part is sufficient to find the effect of joining two arguments. This is not always true, as the next example shows:

```
a(X, X).
a(1, 2).
```

Initially, both arguments are unable to use the equally test, resulting in $\Gamma(\langle 1\rangle) = \{\langle 1,2\rangle\}$ and $\Gamma(\langle 2\rangle) = \{\langle 1,2\rangle\}$. The intersection of these two sets would result in $\Gamma(\langle 1,2\rangle) = \{\langle 1,2\rangle\}$, whereas the two clauses are actually exclusive.

The problem is that some tests are only useful when one only looks at several arguments together. We call these tests *ligatures*.

In our current implementation, while calculating the residuals, the algorithm also looks at tests of the form $test(Z_i, Z_j)$ and uses simple tests to verify if they exclude other clauses. If so, they are stored in a table of ligatures. This table groups pairs of arguments and tests, and gives the clauses that become mutually exclusive when such tests hold. When step 2 is called, this table is checked, and if ligatures exist for the new argument the corresponding pairs are deleted from the intersection. In the example, a ligature exists for the pair of arguments $\langle 1, 2\rangle$ and results in deleting the pair of clauses $\langle 1, 2\rangle$, thus making the combination of arguments successful.

It is possible that ligatures may be more sophisticated, for example connecting two arguments deep down a compound term. In practice this situation does not seem to arise often, and we do not search for these cases in the current algorithm.

9.2.4 Compiling Decision Trees

The final step of the algorithm is to generate the actual sequence of instructions that are followed by the engine to find determinacy code. In this chapter we discuss the compilation algorithms. How the Andorra-I engine supports the determinacy code can be found in chapter 10.

At this stage, it is already known the tests and their combinations necessary to make clauses a single solution. The aim of this algorithm is therefore to organise these tests into an efficient data-structure, typically a tree or graph. The algorithm tries to minimise (i) code size, by only placing tests in the least number of places, and (ii) the depth of the tree, in order to minimise the number of operations needed at run-time to find determinacy.

To describe the algorithm we first to define the *literal residue*, $\mathcal{R}_l(C, g)$, of a clause $C = \langle i, A\rangle$ with respect to a test g as:

$$\mathcal{R}_l(C, g) = \begin{cases} \langle i, A \backslash g\rangle & G \in A \\ \emptyset & otherwise \end{cases}$$

In other words, the literal residue simply removes the test from the clause whenever present, or returns empty.

The algorithm works argument by argument. For each argument, it selects a test, and the clauses that include that test (for which the literal residue is non-empty). It then creates a node in the decision tree for the test, with two alternatives corresponding to success or failure of the test. For the success case, the algorithm tries to discriminate among the clauses that match the test, by calling itself only for these clauses but without the test. For the failure case, the algorithm calls itself on the clauses that do not include the test.

When no more tests are left for a clause or set of clauses, the argument must check the collisions table built by the previous steps. The cases are: (a) the tests lead to a single clause and further the argument makes that clause a single solution, hence the algorithm can generate a `commit` instruction; (b) there is no way the tests can make the clause(s) determinate, and the algorithm must generate a `suspend` instruction; (c) further arguments are needed. In the last case, some bookkeeping is performed, and the algorithm is called again for the next arguments.

The algorithm is presented by the function *decision-graph* in Figure 9.2. Its arguments are P, consisting initially of all the pairs argument and clause for the argument j, *Suspend* saying where to go if analysis of the argument must stop, $Fail$ saying where to go if the current test fails, and $Table$ the collision table that includes information about which clauses are determinate for the argument, and about which (if any) combinations of arguments are necessary to make them determinate. Its output is a graph. Inner nodes are quadruples consisting of a test, and three links telling where to go if the test succeeds, fails or must suspend. Outer nodes are instructions to commit to a clause, fail or give up analysis in this branch.

For every call the *Suspend* argument is made to point to the decision graph generated for the next argument. The last argument is made to point to the constant `wait` meaning that there are no more ways to make the goal determinate.

The function *single-clause* is called when only a clause with no tests is left. In that case, the function *check-collision-table* checks $Table$ to verify if more arguments are necessary. If so, a new call to decision-graph is made for those extra arguments. If several alternative combinations of arguments exist, then the algorithm behaves as the one for the entire procedure, by generating code for each combination and combining them in a such a way that if one combination of arguments does not make the goal determinate, another combination will be tried.

The function *find-test* has to search P for tests. If no or a single test is available, then the function is trivial. Otherwise, it is sometimes important to select a good test first. Basically, it is necessary to follow the partial order $\geq_\psi$ between variables in tests. Further ordering of the remaining tests is discussed in section 9.2.7.

The function *many-clauses* is similar to *single-clause*, but manipulates several clauses. It

```
decision-graph(P, j, Suspend, Fail, Table)
    if (P = ∅) then return(Fail);
    if (P = {⟨i, ∅ >})
        - we can generate code for a single clause
        - either if the procedure has a single clause
        - or if only a single clause matches the test
        return(single-clause(P, j, Suspend, Fail, Table);
    else
        - multiple clauses
        g := find-test(P));
        if ((g = ∅)
        - no more tests, extra arguments need to be tried
            return(many-clauses(P, j, Suspend, Fail, Table));
        else
        - there is a test g:
            - generate code for the case test succeeds
            P_yes := {⟨i, A'⟩ | ⟨i, A'⟩ = R_l(⟨i, A⟩, g)};
            G_yes := decision-graph(P_yes, j, P_no, Suspend, Table);
            - generate code for the case test fails
            P_no := {⟨i, A⟩ | ∅ = R_l(⟨i, A⟩, g)};
            G_no := decision-graph(P_no, j, Fail, Suspend, Table);
            - link the resulting code
            return(node(P_yes, P_no, Suspend, Table));
```

Figure 9.2: Decision-Graph Algorithm for Individual Arguments

may happen that only some of these clauses can be made determinate. Even then, the algorithm must call decision-graph for *all* the clauses, in order to be able to propagate failure correctly. If no tests exist and the clauses can never be determinate, the function generates a leaf with value *suspend*.

The function *check-collision-table* operates on the collision table. It receives as arguments the collision-table itself, the argument, and the clause number. It then checks if there are any combinations of arguments for that clause involving that argument number. To avoid duplication of code, it is very important that a combination of arguments will only be used once. This is implicitly implemented in the algorithm.

Examples We use the procedure `c/4` (from page 148) to exemplify code generation. The code is a disjunction for the three possible cases, corresponding to the first, second, and third argument.

Consider the first argument. The function *decision-graph* can select the test $Z_1 = 1$ first. The literal residues for that test include two clauses, the first and second clause. There are two possible cases now.

- Calling *decision-graph* for the literals residues when the test succeeds gives two clauses but no remaining tests for that argument. Thus, the function *find-test* returns no more tests and *many-clauses* is tried. This function returns that by trying tests from the second argument the clauses become single-solution, and the algorithm moves on to the second argument. The test $Z_2 = 1$ is selected first. In the *yes* case, the algorithm calls *decision-graph* with an empty residue, and can commit to the first clause. In the *no* case, the algorithm goes to test the second clause. Selecting the test $Z_2 = 1$ returns two alternatives, either committing to the second clause if the test succeeds or failing.
- After removing the clauses such that $Z_1 = 1$, only the third clause and the test $Z_1 = 2$ are left. The algorithm thus can commit to the third clause if the test succeeds, and fail otherwise.

The result for the second argument alone is that one can commit to the first clause if $Z_2 = 1$, but cannot commit to a clause if $Z_2 = 2$. If both tests fail, one can fail.

The results for the third argument are that one can commit to the first clause if $Z_3 = 1$, to the second if $Z_3 = 2$, and to the third if $Z_3 = 3$.

The resulting graph is shown in Figure 9.3.

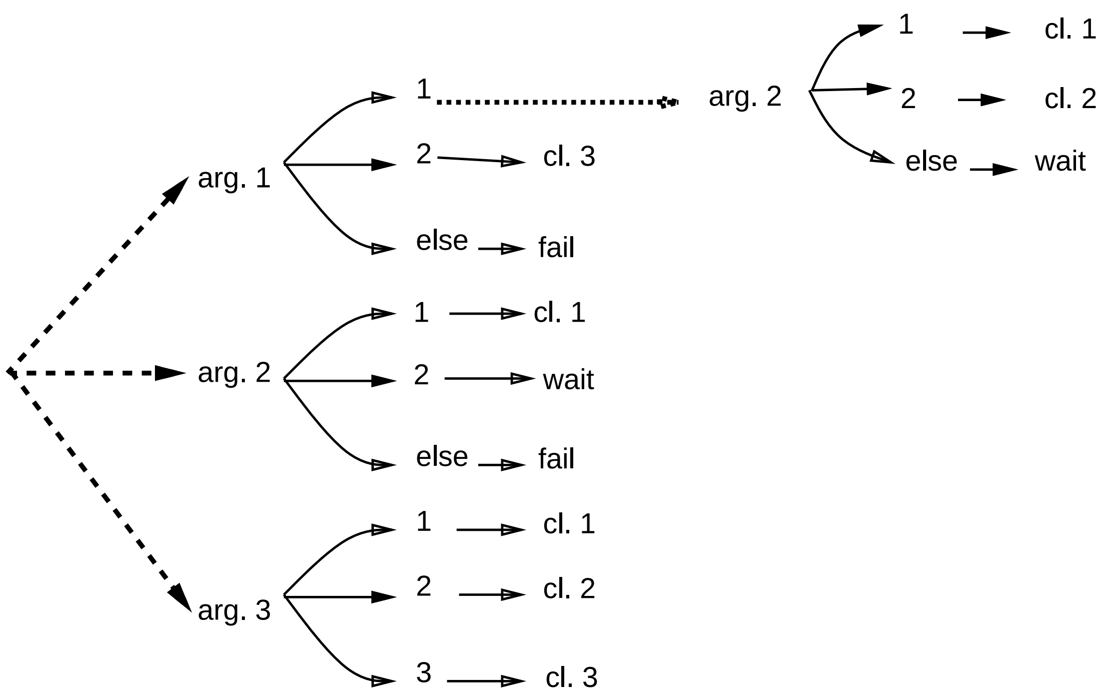

Figure 9.3: Determinacy Graph for a Pure Procedure

The next example shows how the determinacy code processes structures. The example is the procedure:

```
q([_]).
q([_,_]).
q([_,_,_]).
```

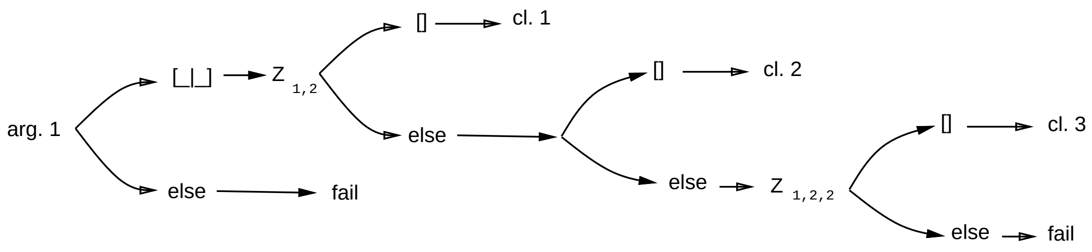

Figure 9.4: Determinacy Graph for a Procedure With Compound Terms

The resulting graph is shown in Figure 9.4. The first step of the determinacy preprocessor recognises that the tests necessary to make the clauses determinate are $Z_1 = [\_|\_], Z_{1,2} = []$, for the first clause, $Z_1 = [\_|\_], Z_{1,2} = [\_|\_], Z_{1,2,2} = []$, for the second clause, and $Z_1 = [\_|\_], Z_{1,2} = [\_|\_], Z_{1,2,2} = [\_|\_]$ for the third clause. The third step groups these tests into the graph shown in a simplified form by Figure 9.4.

Finally, we show an example of binary tests, in this case used to compare arguments (modes are used to reduced code size, as discussed in section 9.3.3).

```
:- mode partition(+, +, -, -).

partition([], _, [], []).
partition([X|Xs], A, Smaller, [X|Larger]) :-
        A <  X,
        partition(Xs, A, Smaller, Larger).
partition([X|Xs], A, [X|Smaller], Larger) :-
        A >= X,
        partition(Xs, A, Smaller, Larger).
```

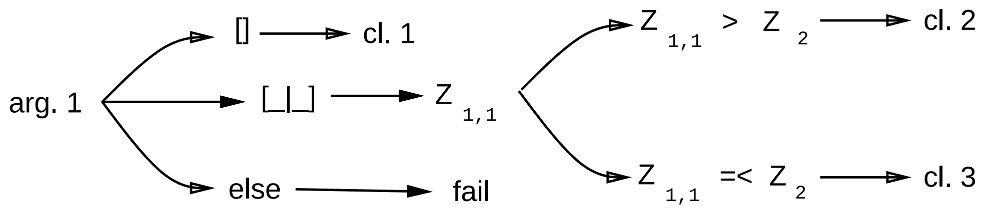

Figure 9.5: Determinacy Graph for a Procedure With Binary Tests

The graph is presented in Figure 9.5. The graph only considers the first two arguments. If the first argument is instantiated, the first step of the algorithm detects that the tests $Z_2 < Z_{1,1}$ and $Z_2 >= Z_{1,1}$ are necessary for determinacy. The third step then groups this step into the graph shown in Figure 9.5.

9.2.5 Failure

It is convenient to explain propagation of failure in some detail. The $Fail$ label tells where to go if the entire procedure fails. Initially, the function *decision-graph* is called with $Fail$ set to `fail`, meaning that no clause satisfies the goal, and therefore that the goal should fail. Each call to *decision-graph* creates a node with two subtrees, the `yes` and `no` trees. The `yes` tree fails to the result of the `no` tree, and the `no` tree fails to the fail point for the caller.

Note that because we assumed that, for the `yes` tree, the leaf test has succeeded, then it is sometimes possible to prove that the entire, or at least segments of, the `no` tree could be avoided, and failure inside the `yes` tree could jump directly to $Fail$. This is implemented in Kliger and Shapiro's [91] residual trees by using the idea of negative residuals, $\mathcal{R}_{no}$. The implemented version of the preprocessor supports a residual $\mathcal{R}_{nl}$, which tells what clauses are available after failure of a test.

9.2.6 Suspensions

Suspension of goals may happen in two cases. In the first case, an argument of a test may still be uninstantiated, and the current branch of the determinacy tree must suspend and wait until the argument is instantiated. In the second case, it may be found that analysis of the current branch of the determinacy tree does not lead to any way of making the goal determinate. Note that in the latter case, there may still be some active branches that may make the clause determinate, so it is still possible that the goal will become determinate in some other way.

In order to handle suspension efficiently, the analyser needs to know how the engine implements suspension and resumption of work. Korsloot and Tick present a detailed discussion of several strategies [94]. In our case, the main priority is to avoid redoing work when a suspended goal is reactivated. To do so we need two guarantees offered by the Andorra-I engine. Firstly, if the engine suspends work at some point, it may resume execution from the same point; secondly, if the engine restarts work from a certain point P_1, and the point to go if suspending was S_1, and if it continues again up to a point P_2, where the point to try after suspending is again S_1, the machine will *not* execute the code starting from S_1 again.

In practice the engine does not always always guarantee the first rule. The size of a suspension record (see section 3.2.1) is fixed, and it is not always possible to store there all the arguments of a test. This may happen with two types of tests, compound tests and binary tests. For such cases, the analyser has to generate code such that the engine will suspend at a previous point where all the arguments were available (instead of suspending where it stopped), and hence needs to redo some work.

The second guarantee avoids repeating execution of argument trees. Consider a decision tree with N argument sub-trees, one for each argument. According to the algorithm, if executing the decision tree for argument $i-1$ suspends, it will point at the decision tree for i. The first time the algorithm is called, imagine execution reaches argument $i-1$ and then suspends. It will thus next try argument i, and maybe suspend again. Suppose now that argument $i-1$ is restarted and suspends again. In this case the engine prevents execution of the tree for i, and no work is redone.

A simple alternative scheme would be to have all trees pointing at `exit`, thus avoiding any possibility of redoing work. In this case there would have to be a separate operation to start all the alternative branches of the tree. This would need slightly more complex support from the engine than what is needed now, albeit needed for less points in the program. Hence we believe the current solution is more efficient.

9.2.7 Optimisations

We next discuss the most important optimisations performed by the system.

Merging Nodes

The main issues in optimisation are how to merge nodes to create switch nodes, how to delete extraneous testing, and the issue of how to propagate failure in order to detect determinacy.

As in indexing for the WAM, the main optimisation of this system is to merge simple testes of the form $Z = constant$ or $Z = f(\ldots)$ into a large node, which we will call the *switch node*. Switch nodes consist of two tests: first the type of the argument, i.e., if the argument is a constant, a structure or a pair, and then for each type, its value. There are several advantages to this optimisation:

- switching on these tests will now be done in fixed-time, not in a time proportional to the number of tests;
- the resulting code is much more compact;
- finally, and because it is known that all the tests in the switch exclude each other, then all the subtrees for the switch node must exclude each other, so they can all inherit the same $Fail$ point.

Switch nodes are very common as most programs contain some switch nodes and many do not need any other testing. In terms of the algorithm, the following changes are relevant:

- The function *find-test* may return either a set of simple tests and associated clauses for the same variable, or a single pair test clauses.
- In the first case, a switch node is generated. The switch node connects to a subtree for each possible argument, plus an alternative branch in case all the tests fail. This

last branch is connected to the fail label. Each subtree is generated as before, but they all receive the same fail label.

- The switch node is implemented in WAM style by a "test on type" node, calling two "switch on argument" nodes. The first verifies if the argument is a constant, a pair or a compound term. The second are called if the argument is a constant or compound term, and select respectively on the value of the constant or on the main functor of the compound term. Notice that the "switch on argument" nodes are only generated if there is at least one test, otherwise the corresponding label from the "test on type" node is connected to the otherwise alternative.

Otherwise Commitment

Our algorithm is geared at finding the tests in a clause that make it the single solution for the goal. But sometimes, although no tests can make a goal determinate, a clause can still become a single solution simply because all the other clauses have failed. A typical example is shown next:

```
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).
```

The example shows a rather frequent situation. If $Z_1 = Z_{2,1}$ both clauses match the input, and the goal is not determinate. But if $Z_1 \neq Z_{2,1}$ then the first clause fails and the goal can determinately commit to the recursive clause.

Because our algorithm is designed to minimise the number of tests that are performed to find determinacy, it is sometimes the case that some tests necessary to exclude clauses may be discarded. The following changes avoid this:

- In the first step, we include tests, not as long as they exclude all other clauses, but as long as they exclude *all but one clause.*
- Expansion of arguments and ligatures should be used and stored in the table if they result in the clause becoming a single solution or if there is only an alternative clause.
- When compiling tests for a clause, if the clause is nondeterminate because of only one other clause C_j, and if the current $Fail$ label is `fail` then the fail branch for the current test should point to C_j.

We next show another example of this situation. The procedure is shown next:

```
a(a, X, X).
a(X, a, X).
a(X, X, a).
```

Note that if all three first arguments are instantiated then the only nondeterminate query is `a(a, a, a)`, all other queries have a single matching clause, if at all. The actual code generated by the preprocessor is shown in Figure 9.6.

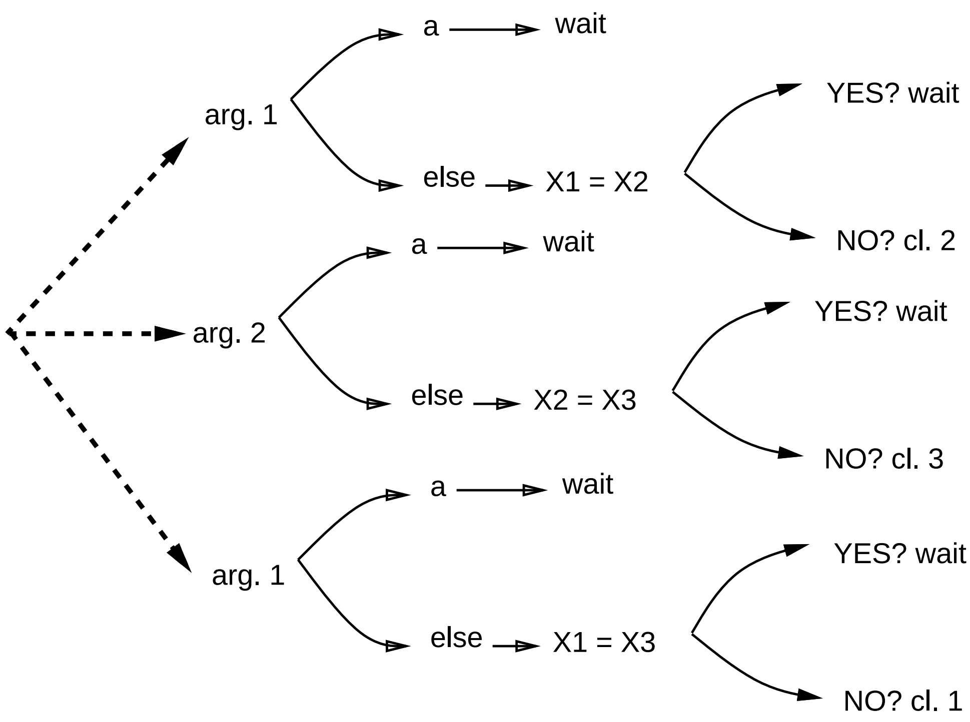

Figure 9.6: Determinacy Graph Through Failure

The goal can commit to the second clause if either the first argument is not bound to `a` and the first argument does not unify with the second, and similarly for the other arguments.

A more complex situation arises from implicit failure. Consider the following simple program [149]:

```
father(terach, abraham).
father(terach, haran).
father(haran, lot).
father(haran, yiscah).
father(terach, nachor).
father(abraham, isaac).
father(haran, milcah).
```

The goal `father(X,X)` should fail (there is no one father of himself) and therefore should be recognised as determinate. Unfortunately, the current algorithm will generate code that will wait until one of the arguments becomes instantiated. This in turn results from the fact that the test $Z_1 = Z_2$ has no matching clause is not *explicitly* registered in the program, and therefore is not considered by the algorithm.

This last example shows the limitations of not being able to bind external variables to detect determinacy. In section 9.4.2 we present a more detailed discussion of this general problem and of possible solutions. Obviously, there is a solution for this specific problem, which is to verify if a condition like $Z_i = Z_j$ or $Z_i \neq Z_j$ can never hold, and generate an extra test node comparing the two arguments in that case.

9.3 Extensions to the Algorithm

The three main extensions necessary when detecting determinacy for Andorra-I are generating code for procedures using commits or cuts and the user mode declarations. We discuss their implementation in detail.

9.3.1 Procedures using Commit

In Andorra-I *flat* commits can be used to say that a clause is the only solution for a goal. A commit is defined to be flat when all goals before the commit (the *guard* of the commit) are test goals. A goal may commit to a clause defined with a flat commit if all unifications and builtins performed before the commit can succeed without binding any external variables, **or** if the clause is the only solution for the goal.

The Algorithm

Our approach to compiling procedures with pruning by commit extends the algorithms used for pure procedures. Briefly, these algorithm select which clauses to commit to, and then a simple compilation algorithm compiles the guards of the clauses. The resulting algorithm is thus:

- The first step, establishing arguments' decision trees is left as is.
- The second step, combining decision trees for arguments, is left as is.
- The compilation algorithm is changed so that if a leaf has a commit associated to it, then the tests before the commit which have not been tried so far will be compiled

as a guard. If these tests succeed, the compilation generates a `commit` node; if one of the tests fail, either (a) the clause was determinate without the commit, and the analyser goes back to the usual algorithm to find the fail label, or (b) the clause was only determinate because of the action of the commit and the analyser generates a `suspend` node. The algorithm is shown in figure 9.8. The code assumes that every clause has a commit.

- Finally the analyser has to generate code for each guard. The code is very similar to the code generated by a Prolog or a PARLOG compiler.

```
parmerge([], L2, L3) :- |,
    L2 = L3.
parmerge([H|L1], L2, L3) :- |,
    L3 = [H|Ls3],
    parmerge(L1, L2, Ls3).
parmerge(L2, [], L3) :- |,
    L2 = L3.
parmerge(L1, [H|L2], L3) :- |,
    L3 = [H|Ls3],
    parmerge(L1, L2, Ls3).
```

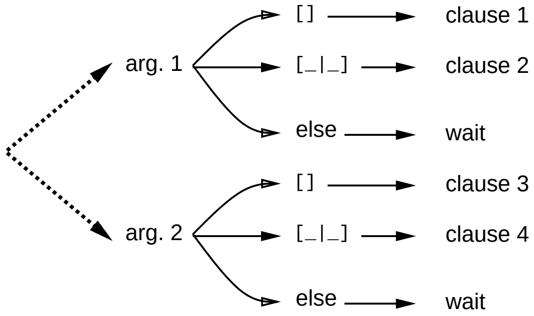

Figure 9.7: Determinacy Graph for Parallel Merge.

The resulting determinacy code is very similar to code using decision graphs for committed choice languages [92], the main difference is that we select the tests that were used in the first phase to do indexing. An example is shown in the figure 9.7.

The example shows the determinacy code for the parallel merge procedure (very well known in committed choice languages). The determinacy code we generate waits until one until of the two first arguments becomes a list, and commits to the corresponding clause. This is the expected behaviour in the committed choice languages [58].

Failure and Commit The previous algorithm does not always detects determinate failure. Consider the previous example. In this case, the algorithm may suspend simultaneously on the first or second arguments. Say that one of the arguments becomes instantiated to a value different from the pair or empty list, e.g., `parmerge(tree(_,_,_), L, O)` or `parmerge(L, tree(_,_,_), O)`. In both cases, there are still some clauses available and the goal should not fail. Consider now the goal `parmerge(tree(_,_,_), tree(_,_,_), O)`. This goal should fail, but the problem is that the test node for the second argument is called under two different circumstances:

- when the first argument suspends and the algorithm tries committing via the second argument;

commit-graph$(P, j, Suspend, Fail, Table)$
    if $(P = \emptyset)$ then return$(Fail)$;
    if $(P = \{\langle i, \emptyset\rangle\})$
        - code for a single clause
        return(try-clause-pruning$(P, Suspend, Suspend)$;
    else
        - multiple clauses
        $g :=$ find-test(P));
        if $((g = \emptyset)$
        - no more tests, extra arguments need to be tried
            if $(j = arity)$
              - no more arguments, generate a sequence of "try" nodes.
              $P = C \cup P'$;
              $Continuation =$ commit-graph$(P', j, Suspend, Fail, Table)$;
              return(try-clause-pruning$(C, Continuation, Continuation)$);
            else
              $j' := j + 1$;
              $P' =$ find-new-args$(P, j', Table)$;
              return(commit-graph$(P', j', Suspend, Suspend, Table)$);
        else
        - there is a test g:
            - generate code for the case test succeeds
            $P_{yes} := \{\langle i, A'\rangle \mid \langle i, A'\rangle = \mathcal{R}_l(\langle i, A\rangle, g)\}$;
            $G_{yes} :=$ commit-graph$(P_{yes}, j, P_{no}, Suspend, Table)$;
            - generate code for the case test fails
            $P_{no} := \{\langle i, A\rangle \mid \emptyset = \mathcal{R}_l(\langle i, A\rangle, g)\}$;
            $G_{no} :=$ commit-graph$(P_{no}, j, Fail, Suspend, Table)$;
            - link the resulting code
            return(node$(P_{yes}, P_{no}, Suspend, Table)$);

Figure 9.8: Generating a Decision-Graph for procedures with commits only

- when the decision tree for the first argument fails.

In this example we have the second case. The problem here is that if the test in the second switch node fails again, then the two cases should result in different actions:

- the goal should wait for a solution until the first argument becomes instantiated;
- the goal should determinately fail.

It is possible to separate the two cases at compile time, but in the worst case we would be forced to duplicate execution of many of the guards. A run-time solution is proposed by Kliger and Shapiro [91]. Briefly, the implementation of FCP(|,:,?) uses a *suspension table*, initially empty. The suspension table stores positions where the algorithm suspended. Whenever a node fails, the implementation withdraws the corresponding position from the table and tests if it is empty. If not, other nodes are still available and there may still be a solution for that goal. Otherwise, the goal has determinately failed. The suspension table is used for other purposes in FCP(|,:,?). In the case of Andorra-I a counter would be sufficient.

9.3.2 Procedures using Cut

To be consistent with the standard meaning of cut, the determinacy code may commit to a clause with a flat cut if that cut does not bind any external variables, and all previous clauses are guaranteed to fail. The main new concept introduced by the cut is thus the concept of *previous* clause, that is, of an ordering between clauses. We thus extend the pure procedures determinacy analysis algorithm in the following ways:

- The first and second steps are kept.
- A new step goes about each clause and verifies if all previous clauses had (a) a cut associated with them or (b) excluded the current clause for one argument i. If so, a cut is associated with the leaves for that clause.
- The compilation algorithm is changed to separate clauses into *sections*. Sections are contiguous groups of clauses for which the current argument either always has, or always has not a simple test. In most cases procedures have only one section, there is a test in every clause, or two sections, there is a test in every section except in the last section that functions as a catch-all clause [28].

cut-graph($P, j, Suspend, Fail, Table$)
  if ($P = \emptyset$) then return($Fail$);
  if ($P = \{\langle i, \emptyset\rangle\}$)
    - code for a single clause
    return(try-clause-pruning($P, Suspend, Fail$);
  else
    - multiple clauses
    $P' :=$ find-section(P, j));
    - find the last clauses:
    $P'' := P \backslash P'$;
    - generate code for the last clauses:
    F :=cut-graph($P'', j, Suspend, Fail, Table$);
    - generate code for the first clauses:
    return(section-graph($P', j, Suspend, Fail, Table$);

Figure 9.9: Generating a Decision-Graph for procedures with cuts only

Figure 9.9 shows a description of the procedure *cut-graph*, that generates code for procedures with cuts. Sections are compiled one by one. Inside a section there may be two cases, (a) there may be tests for the argument and in this case the function is very similar to commit-graph, but being careful to always respect clause ordering, or (b) no tests are available. In case (b) indexing may be tried on other argument, or code may be simply generated as a sequence of tries of guards.

- Finally the analyser has to generate code for each guard. The algorithm is identical to the one used for commits.

The actual execution of a flat cut by the engine is similar to the execution of code supporting optimised shallow backtracking in Prolog [17]. The analyser uses the decision tree for an argument as indexing code, and then for each clause, compiles the head and builtins before the cut into special instructions. At runtime, the indexing code is tried first, selecting the subset of clauses to try. The engine will then execute head unification and the builtins for each clause sequentially. Execution of head unification or a guard will suspend if it needs to bind an external variable. If head unification and all builtins succeed, the call will commit to that clause. If one of them fails, the next clause will be tried, until no more remain, and the call fails.

The next procedure gives a simple example of the use of cut to promote determinacy.

```
notequal(A, A) :- !, fail.
notequal(qqq(_), _) :- !, fail.
notequal(_, qqq(_)) :- !, fail.
notequal(_, _).
```

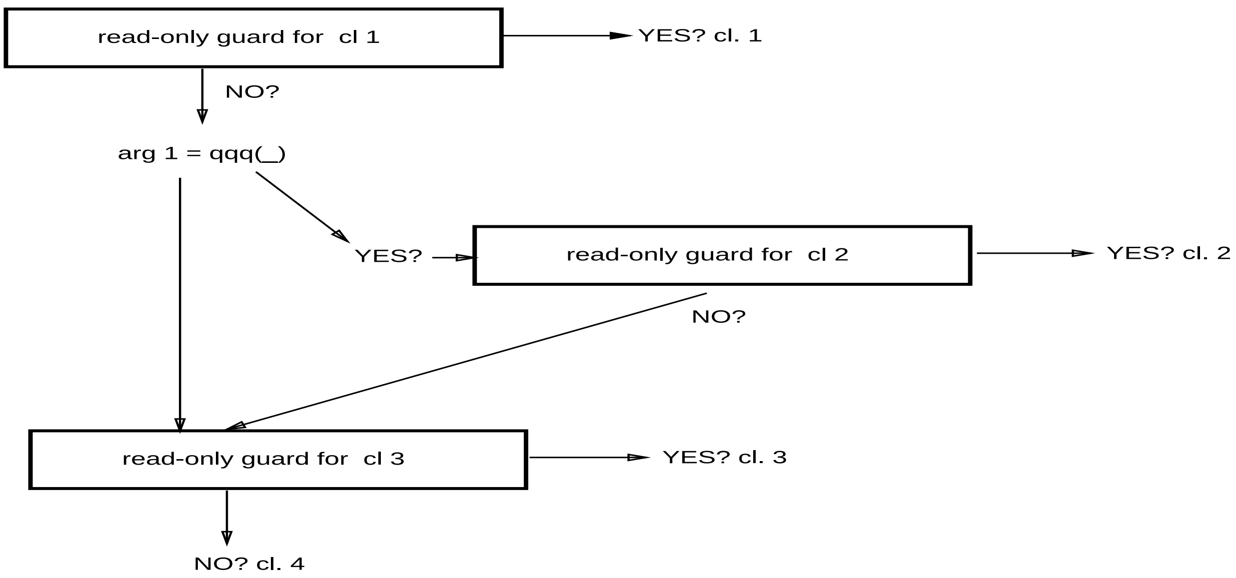

Figure 9.10: Determinacy Graph for a Procedure With Cut.

The resulting code is shown in Figure 9.10. The first and second steps do not find ways to make the clause determinate, and it is the cut analysis that does most of the work in detecting how the cut makes goals determinate. The code inside the boxes verifies when the head and guard of the cut is read-only.

9.3.3 User Declarations

Andorra-I uses Prolog style mode declarations give information on the state of instantiation of arguments. Input mode declarations say that an argument will be instantiated when goals for the procedure are called, output mode declaration say that the output arguments should not interfere with execution of the procedure.

Only output mode declarations affect the determinacy analyser; the analyser simply ignores the output arguments, thus avoiding wasted work in testing arguments which probably will never become determinate.

The decision to ignore input mode declarations deserves some discussion. In languages like PARLOG [58] or Mu-Prolog [116] input declarations are used to delay execution of a procedure until the input arguments are instantiated. Such behaviour could indeed have easily been implemented in the Andorra-I preprocessor. We decided not to, attending to the fact that our mode declarations are supposed to represent the execution of programs initially written for Prolog, and thus should not be mistaken with input declarations used to control execution. There is no reason, though, why future preprocessors for other languages could not enforce such restrictions, and the current machinery could be used in a straightforward manner to support such "strict" input arguments.

9.4 Discussion

The determinacy analyser has been implemented and used to compile a wide variety of applications for Andorra-I. The results have been quite successful, and for all programs we tried the analyser has been able to generate correct code.

We next discuss the two main questions that arise in the use of the analyser. We first discuss how the complexity of the algorithms used in the preprocessor affects performance of the system, and we next discuss how the preprocessor could be adapted to detect **all** determinate goals. We finally present an overview of the background work that was important in the development for our system and compare the system with related work.

9.4.1 Complexity Analysis and Performance

A simple complexity analysis for the determinacy analyser gives, for each step:

- As regards construction, in the first step we need to compare each clause with every other clause. A simple implementation would be of $O(n^2)$, where n is the number of clauses. A more sophisticated implementation using hashing or trees could perform better in the average case.

  While comparing clauses, as a first approximation one can say that each step would take in the worst case $O(t^2)$, where t is maximum the number of tests in the clause. This result assumes that the inference mechanism takes constant time given two tests and compares all tests. Sophisticated inference mechanisms need to test combinations of tests and will not obey this approximation.

- Considering the "merge" step, the worst case is the case where all combinations of arguments need to be generated. In this case, complexity would be $O(n^2 * 2^m)$, where m is the arity, and n is the number of clauses.

- Finally the complexity of the code generation step can be obtained from the fact that each test generated by the previous step will always be considered, but only once. In this case, the complexity of this step results from the number of tests generated by the previous step. Considering that n tests were generated for each combination of the arguments, then the worst case is $O(n * 2^n)$.
- As regards the introduction of cuts and commits, it generates a new linear factor on the number of clauses n. Complexity analysis for the compilation process is not much changed.

Notice that the entire algorithm has been designed to avoid generating combinations of clauses, unless absolutely necessary. In practice it is extremely rare that all combinations of arguments will be generated, and most of the time spent by the analyser is spent on the actual code code generation, and particularly on the first step, as comparing tests is quite time consuming.

9.4.2 Simplifications Performed by the Preprocessor

The current system makes simplifications while generating the determinacy code. These simplifications are necessary in order to generate efficient code, but one loses the ability to detect all determinate goals, and in fact we currently only guarantee that a determinate goal will be always recognised if the conditions for determinacy are made explicit, by using the cut or commit.

Although in our experience the current system does seem to be able to recognise most cases of determinacy, it may still be desirable to be able to guarantee that, at least for some predicates, *all* determinate calls will be recognised. Considering that any decision tree based algorithm will either result in exponential code, or will avoid an explosion of code by making simplifications, and also the difficulties of having exact inference mechanism for all types of tests available in logic programming languages, we believe that the best solution for a complete implementation of determinacy is to use our algorithm only to constrain the number of clauses to try, and then return to the original definition of determinacy, that is to simply wait until all clauses except one are guaranteed to fail.

The extensions we need for the analyser to support this algorithm are simple. Indexing can be provided by reusing the current algorithm, and replacing tips which do not result in committing to a clause or failing by calls to an algorithm directly implementing the definition of determinacy.

An interesting optimisation results from the observation that we need only to know that two clauses can still match to prove that a call is not determinate. The analyser thus only needs to generate code to manage a pool of two "test" clauses. Initially, two clauses are immediately added to the pool. Whenever one of them fails, it is removed from the pool, and one of the remaining clauses outside the pool is added to the pool. If no clauses are left outside, the algorithm commits to the clause remaining in the pool. This algorithm has two basic advantages over a naïve algorithm: (a) it may test fewer clauses in the case where the goal will never become determinate; (b) it will have less run-time overheads, as analysis will only suspend for two clauses at most, and therefore suspension records will only have to be created for these two clauses.

The actual code to verify when a call fails can be initially implemented in Andorra-I as a call to a special routine which will wait until head unification fails, followed by code to verify whether certain builtin goals fail. Most of the principles used to compile guards for cuts or commits can also be applied in a straightforward manner to verify when a call fails.

We next show how code for a pool of four clauses would look like:

```
init_pool      <1>
add_pool       <3>
add_pool       <4>
close_pool     <6>
```

The instruction `init_pool` starts a pool for the goal. The instructions `add_pool` add a new clause to the pool. Finally the `close_pool` instruction adds a clause to the pool and closes the pool, guaranteeing that no more clauses are available.

9.4.3 Related Work

Our work in determinacy analysis has been influenced by two main sources. The work on indexing and sophisticated indexing techniques developed for Prolog, and on the compilation techniques developed for the Flat Concurrent Logic Programming Languages, particularly for FCP. We discuss them now in detail.

For procedures with several clauses, by default conventional implementations of Prolog will create a *choice point* and try each clause in sequence. In many cases, the input goal is such that by doing simple tests one can easily verify that only a few clauses can match. Most Prolog systems thus introduce *indexing*, initially a technique where one argument of the input goal is tested to reduce the number of clauses to try.

Indexing appeared in Warren's DEC-10 Prolog system [177]. In this case, indexing consists of tests on the first argument. Similar ideas are followed by the WAM, with indexing also performed on the first argument [178]. The tests performed on the first argument consist of a test on type, to verify if the term is a pair, a general compound term or a constant. If the argument is a general compound term or a constant, hashing may be performed on the main functor of the compound term or on the value of the constant.

Studies show that choice point creation and maintenance have a serious impact on the performance of most Prolog programs [160, 104, 163, 105], and that further optimisations can be applied to the WAM. Several improvements have indeed been proposed for the WAM, although most of them preserve the basic indexing scheme of the WAM [11, 16]. Slightly more ambitious modifications are proposed in the YAP Prolog compiler, particularly tests are also made on the first arguments of pairs [132].

Even more ambitious schemes extend indexing by considering several arguments, looking inside compound terms, and considering tests in the body of the clause. The scheme proposed by Van Roy and others [173], uses indexing on several arguments to form a *selection tree*. In their algorithm, only the top functors are considered and shallow backtracking is proposed as a solution to reduce overheads in choice-point creation.

In contrast, Hickey and Mudambi [78] propose a more powerful scheme which compiles the heads of clauses in such a way that most cases of determinate goals will be recognised. First, unification of output arguments is delayed until the body of the clause. *Switching trees* are then constructed by considering exposed positions, initially consisting of the goal's arguments and eventually including arguments from compound terms or constructed by the tree itself, and expanding them if the corresponding subtrees differ in at least two rules. The algorithm is applied recursively until there is a single solution or no more positions can be expanded. In the former case, the positions of the clause which have not yet been considered are compiled and finally there is a jump to the body of the clause. In the latter case, a sequence of try-retry instructions is used to link the code for all the clauses.

This indexing scheme was also shown to generate exponential code for some examples. The authors propose a *quadratic indexing algorithm* where a graph is constructed instead of a tree to prevent this situation.

More recently, Zhou et.al. propose a *matching tree* scheme for the compilation of Prolog [192]. Matching trees are constructed by merging common test patterns among clauses. Arguments of compound terms and builtins can be used to construct a matching tree.

Kliger and Shapiro argue that such schemes are not cost effective for the compilation of Prolog programs [91]. Firstly, in many cases choice-points are really necessary, and a sophisticated indexing scheme will not help. Second, unless the mode declarations are known, there is a risk of doing indexing on output arguments, which will never be instantiated.

The advanced indexing systems we discussed claim to overcome the last difficulty by using global analysis, in the form of abstract interpretation, to provide the modes of use for the program. As regards the first problem, the authors claim that important benefits can be obtained solely by reducing the number of clauses to try, and that the cases where goals become determinate are numerous enough to justify their systems.

In practice, most current Prolog systems preserve the simpler indexing scheme of the WAM, or variations thereof, with some systems trying instead to make shallow backtracking more efficient (as suggested by Van Roy and others). Shallow backtracking schemes are implemented in well-known Prolog systems such as SICStus [17]. In this scheme, a choice point is only completely built when an user-defined goal is launched. This way, the overhead of backtracking due to failure in head unification is minimised. The advantages of shallow backtracking systems is that they result in code-size similar to Prolog while avoiding the disadvantages of fleshing a full choice-point.

Kliger and Shapiro also suggest that sophisticated indexing would be more useful for the flat committed-choice languages, where the above mentioned problems do not arise. They initially proposed *decision trees* as a way to compile FCP(|,:,?) programs. Decision trees are constructed by considering tests appearing in the clauses. For each test and a group of procedures a *restriction*, later called *residual*, is essentially defined as the remaining clauses after removing the procedure, and the *otherwise-restriction*, later named as *otherwise-residual* as the clauses that do not match the test. Compilation proceeds by selecting tests and calculating residuals until no more tests are available.

One problem with decision trees is that the resulting code may increase exponentially. These cases do seem to occur, although not frequently. Decision graphs [92] are an answer to the problem. Decision graphs introduce an additional edge labelled *continue*, that connect the subgraphs emanating from a vertex to their *otherwise* sibling. The main idea in decision graph is that a clause induces **at most** one path on the graph. The key modification to the decision tree algorithm is thus that a clause is propagated from a vertex to at most one edge: if a clause does not care about one test it is propagated to the *otherwise* subgraph, whereas in the decision tree the clause would be propagated to every subgraph. The authors also emphasize that it is important to select the best tests first. A simple heuristic is given to choose a "best" test. Basically, one first chooses a test for which the most clauses *care*. If many such tests exist, one selects the one with minimal variability, that is a test with the smallest number of possible results.

Considering the usefulness of these methods for detecting determinacy, one problem that immediately arises is that all these methods depend on having a fixed mode for calling a procedure. But it can be the case in Andorra-I that the same procedure may be called with several different modes, and in every case we want to find determinacy. In this sense, the Basic Andorra Model is more eager to commit to a clause than, say, a committed choice language, and any algorithm to detect determinacy has to support this. Notice that if the modes of use are fixed we can use the previous techniques, and in fact our compilation algorithm for commits is very similar to the decision graph technique, whereas for cuts it approaches Prolog indexing, with shallow backtracking being replaced by simple read-only tests.

Furthermore, in the previous discussion the techniques were seen as an aid to the efficient compilation of the corresponding languages. In systems implementing the Basic Andorra Model, there is a difference on emphasis as determinacy detection is a fundamental part of the execution model itself.

In our case we decided to concentrate on determinacy detection, and to delay analysis of how to integrate efficient compilation and determinacy code generation until this problem was fully resolved. In this case, we believe that our main problem was trying to generate the most compact code possible, and to avoid explosions of code size to which these approaches are vulnerable. Similar problems are considered in another recent implementation of the Basic Andorra Model, the NUA-Prolog system [124]. NUA-Prolog also does separate indexing for each argument and then uses *clause sets* to combine the several clauses. One characteristic of NUA-Prolog is the use of modes: only when the input modes are sufficiently instantiated will the goal be executed.

A different approach for the compilation of Pandora is proposed by Korsloot and Tick [94]. The authors adapt Kliger's method to compile Pandora don't-know procedures. As in the decision graph algorithm, before a node is created it is compared with other expanded nodes, and in case an equivalent node appears somewhere else an edge to the equivalent node is returned instead.

Although the authors claim completeness, given the appropriate inference mechanism for comparing tests, no justification is made for the case discussed in 9.2.7. Because their algorithm does not have many simplifications, it is quite likely to generate large code, particularly for problems involving compound terms.

9.5 Summary

This chapter described the determinacy analyser. First the fundamental issues in determinacy were discussed, and an algorithm for the detection of determinacy for pure Prolog procedures was given. The algorithm finds which tests are necessary to make arguments determinate, how to combine decision trees from different arguments to make clauses a single solution to a procedure, and how to generate the actual code for each argument's decision trees. Issues including optimisation of decision trees, ligatures, and suspension and resumption of the analysis were also discussed.

Next it was discussed how modes can be used to reduce the amount of determinacy code, and how the algorithm can be expanded to handle procedures with cuts and commits. The actual definitions of when cuts and commits can be used to make goals determinate was discussed in detail.

Finally, the system was analysed in terms of complexity and of the necessary simplifications. A dynamic algorithm was presented. Related work was discussed and compared when relevant.

Chapter 10

Compiler-Based Andorra-I Implementation and Performance

This chapter presents the design of the Andorra-I engine and schedulers, and discusses its performance. The initial implementation of the engine was an interpreter written in C. More recently, a compiled version of Andorra-I was implemented. The engine was extended with an emulator that executes WAM-like instructions generated by the preprocessor.

The chapter first gives an overview of how the Basic Andorra Model can be executed. The fundamentals of the Andorra-I engine are then presented, with emphasis given to the new issues in Andorra-I. An overview of the scheduling issues for Andorra-I is also given. We give a description of how Andorra-I is compiled, by describing the Andorra-I abstract machine and giving an overview of the compiler. The chapter finally presents data on Andorra-I's performance for a set or interesting applications and benchmarks.

10.1 The Andorra-I Engine

Andorra-I programs are executed by *teams* of abstract processing agents called *workers*. Each worker usually corresponds to a physical processor. Each team, when active, is associated with a separate or-branch in the computation tree and is in one of two computation phases:

Determinate For a team, as long as determinate goals exist in the or-branch, all such goals are candidates for immediate evaluation, and thus can be picked up by a worker. This phase ends when no determinate goals are available, or when a determinate goal fails.

In the first case, the team moves to the non-determinate phase. In the second case, the corresponding or-branch must be abandoned, and the team will backtrack in order to find a new or-branch to explore.

Nondeterminate If no determinate goals exist, the leftmost goal (or a particular goal specified by the user) is reduced. A choicepoint is created to represent the fact that the current or-branch has now forked into several or-branches, while the team itself will explore one of the or-branches. If other teams are available, they can be used to explore the remaining or-branches.

Figure 10.1 shows the execution phase in terms of a pool of determinate goals. The figure shows that the determinate phase is abandoned when either no more determinate goals are available or when the team fails, and the determinate phase is reentered either after creating a choice point, or after backtracking and reusing a choice point.

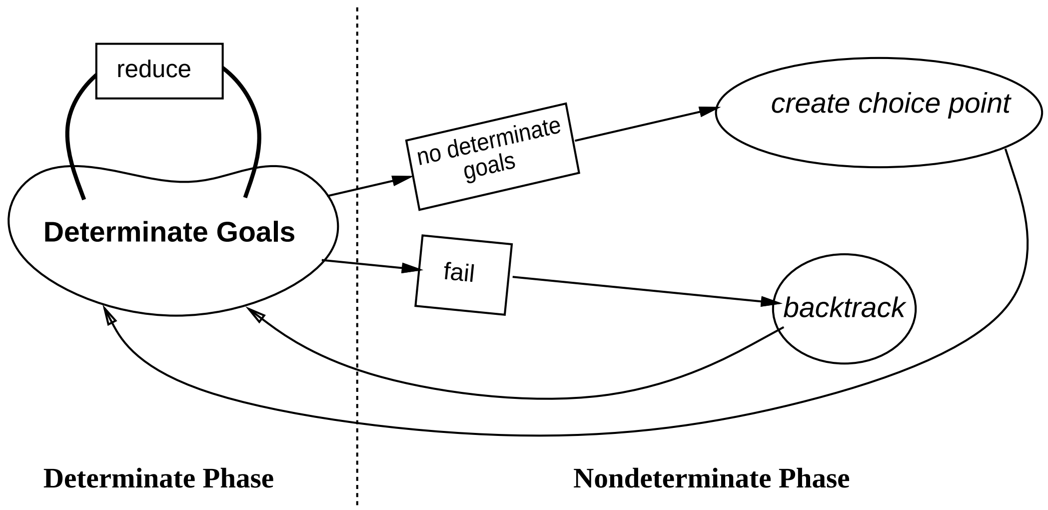

Figure 10.1: Execution Model of Andorra-I

During the determinate phase, the workers of each team behave similarly to those of a parallel committed-choice system; they work together to exploit and-parallelism. During the non-determinate phase, and on backtracking, only one particular worker in the team is active. We call this worker the *master* and the remaining workers *slaves*. The master performs choicepoint creation and backtracking in the same way as an or-parallel Prolog system.

10.2 Principles of Andorra-I

The Andorra-I engine was designed by Yang. The implementation techniques used in Andorra-I are largely based on JAM, a parallel implementation of PARLOG [40], as regards the treatment of and-parallelism, and on the or-parallel Prolog system, Aurora [98], as regards the treatment of or-parallelism. However, the integration of both types of parallelism does introduce a number of new implementation issues.

10.2.1 Data Areas and Memory Management

The main data structures of Andorra-I follow the efficient stack management techniques developed for Prolog, and are in general similar to Aurora. That is, we have four stacks, *local stack, heap, choicepoint stack* and *trail*. As regards the local stack, in order to exploit and-parallelism easily, and following most implementations of committed-choice languages, *goal stacking* is used. Goal records are similar to KL1-B's goal records, but contain different bookkeeping information. The heap contains all compound terms, some (but not all) run-time variables and as in JAM, the data structures for suspension of goals. As in SICStus Prolog, the choicepoint stack contains choicepoints and the trail contains markers to all conditionally bound variables. In addition, since Andorra-I follows the SRI model in exploiting or-parallelism, it uses *binding arrays* to represent the variable binding environments of or-branches. Finally, to help support and-parallelism, Andorra-I uses *run queues* to contain goals that are runnable during and-parallel computation.

Space in the four stacks and the binding array can be recovered by backtracking. Andorra-I also recovers space in the local stack when the goal which is on the top of stack is determinately reduced.

All the workers within a team share the same heap, local stack, choicepoint stack, trail and binding array. The first four data areas for each team are visible to the other teams. During or-parallel execution they become "cactus stacks", as each team grows its own section of the stack.

Andorra-I distributes space on each stack among the workers of a team using a *chunk scheme*. In this scheme, the stacks and binding array are divided into chunks which allow each worker to allocate space independently; a worker claims a new chunk from the master of its team whenever its current chunk is full. Because the master is solely responsible for managing space on the stacks, a slave can easily move to another team without having to worry about space management issues. Chunks are reallocated at each new choice-point: this way choice-points become more compact and it is possible for a variable's offset in the binding array to continue to represent its seniority (e.g., for the purposes of variable to variable bindings) as in Aurora. The drawback of the chunk scheme is that chunk overflow has to be checked whenever a stack is extended, including when creating new variables or structures. However, tests show that the overhead imposed by the chunk scheme is within a reasonable range, about 6% in the interpreted version.

Each worker has its own run queue. The union of all run queues in the team represents a distributed common pool of work for the team. The actual management of the run queues is the province of the *and-parallel scheduler*, described later.

At the end of the determinate phase, Andorra-I must be able to quickly find the correct nondeterminate goal to execute, which is normally the leftmost one. Andorra-I uses a data-structure, the *sideways-linked* chain, to link together all goals [134]. The sideways-linked chain allows easy access to the leftmost goal and easy updating, but needs the updatable variables described later.

10.2.2 Data Structures for Suspension of Goals

Similarly to the JAM, Andorra-I uses *suspension records* to mark suspended goals. For each variable with suspended goals a linked list of suspension records, the *suspended goal list*, stores one record for each point where a goal suspended on that variable. When a new goal suspends, its suspension record is added to the head of the list. When a variable with suspended goals is bound, each record is executed.

There is a problem when a goal suspends on several variables: several workers may try to restart the same goal, or the engine may try to restart an executed goal. In both JAM and KL1-B an intermediate structure is created. In Andorra-I a special field of the goal records and the suspension records is used, the `Ex` field. Figure 10.2 shows the use of this field in the goal records (the ones on top) and in the suspension records (the ones at the bottom). `Ex` is initially a free variable in the goal record (see GR3). When the goal suspends the first time, `Ex` is made to point to an `Ex` field in the suspension record (see GR1), new suspensions are made to point at this same `Ex` record (see GR2), which is instantiated to the atom `_end` when the goal is finally executed. When waking up a goal, Andorra-I tests if `Ex` is instantiated: if so the goal is skipped, otherwise the determinacy code for the goal is restarted. Only when the determinacy code commits to a clause are the `Ex` fields bound. This technique is probably less efficient than JAM's indirect pointer. JAM also optimises suspension on a single variable, a scheme that could also be adapted to Andorra-I.

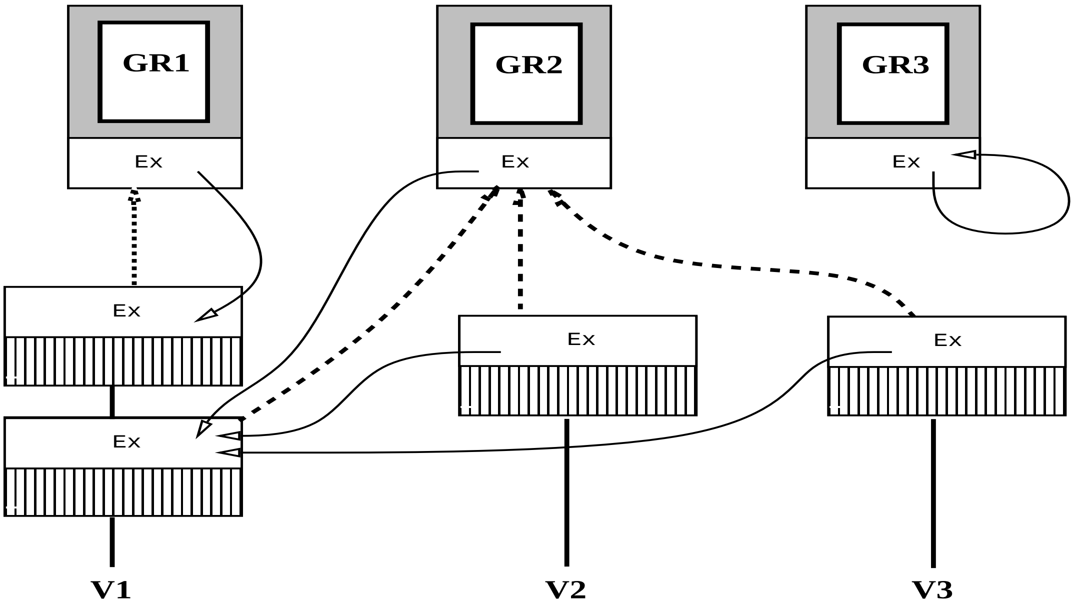

Figure 10.2: Implementing Variable Suspension in Andorra-I

10.2.3 Variables

Variables in Andorra-I reside in goal records or in the heap. As in Aurora, a variable is represented by a variable cell, initialised with a specific tag plus the offset of a new cell in the binding array. According to the SRI model, an unconditional binding (i.e., made to a variable younger than the last choice point) is written into the variable cell; a conditional binding is written into the binding array, with the variable cell kept unchanged. The offset in the variable cell has two uses. It indicates the location in the binding array where the conditional binding is stored, and also represents the age of the variable, necessary to compare seniority of variables.

Updatable Variables

In Andorra-I, updates to certain fields of system data structures need to be backtrackable. The suspended goal list is one example. To update the list in constant time, each time a new suspension record is added to the list a direct pointer to the end of the list is used. Thus, during forward execution, Andorra-I repeatedly overwrites the end of the list in order to extend it. On backtracking, the system needs to restore the list to the state it had before the last choicepoint.

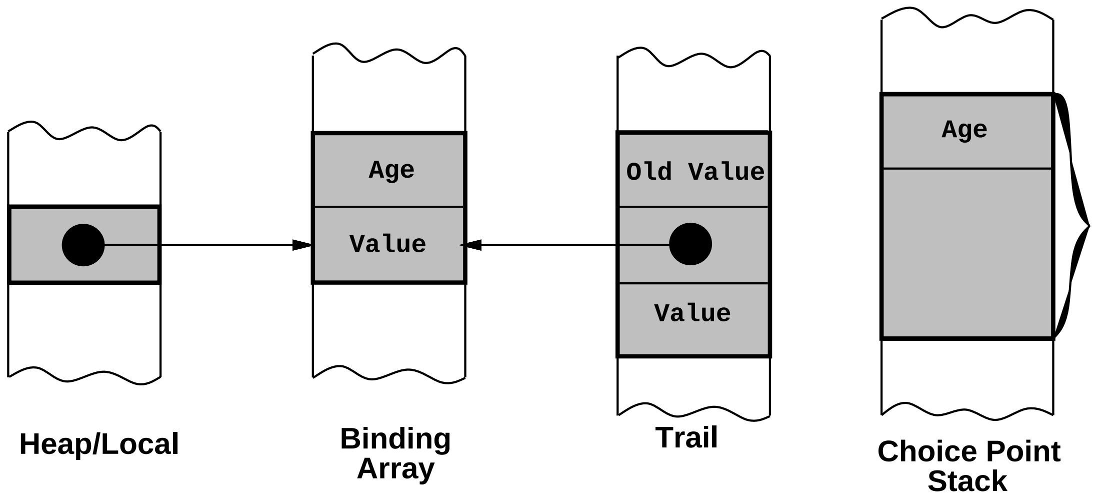

Figure 10.3: Updatable Variables

Updatable variables implement backtrackable updates. An updatable variable can be assigned to more than once. As for bindings, an assignment is said to be *unconditional* if no choicepoint has been created since the variable was created, otherwise it is *conditional*. Updatable variable are represented as value cells, which store any values assigned unconditionally, together with an offset to a binding array location, which stores any values assigned conditionally (see figure 10.3). The binding array location also contains the number of the last choicepoint that existed at the time the conditional assignment was made. Successive conditional assignments corresponding to the same choicepoint do not have to be individually trailed. Only the first binding is trailed by recording on the trail the address of the variable and its previous conditional value (taken from the binding array). Andorra-I also leaves space for its ultimate new conditional value; this is filled in when the next choicepoint is created, by scanning the new trail entries. The trail entries thus ultimately contain both old and new values. The old values are restored into the binding array on backtracking, i.e., when a team moves up the search tree. The new values are installed into the binding array when a team moves down the search tree.

Updatable variables have been successfully applied in Andorra-I wherever we needed to implement backtrackable system data structures. They are also useful for other purposes, such as representing finite domain variables [59].

10.2.4 Andorra-I Extensions

Andorra-I must implement the sequential conjunction and the cut. The *sequential conjunction* is implemented through the sideways-linked chain. Sequentialised goals wait until the side link variable of their parent goal dereferences to a sequential conjunction goal.

The cut and commit are implemented in a way similar to the WAM. A pointer to the choice point at the time of the cut is stored and all choice points up to that point are pruned. Note that some pruning operators will be executed by the determinacy code: in that code the engine will not try to reexecute the guard of the cut and will limit itself to launching the goals in the body part of the clause.

10.3 Scheduling in Andorra-I

There are three independent schedulers in Andorra-I: the and-scheduler, the or-scheduler and the top-scheduler. The *and-scheduler* schedules the and-parallel work to be performed by workers within a team, the *or-scheduler* schedules the or-parallel work to be performed by the teams themselves and the *top-scheduler* schedules the distribution of workers by teams.

10.3.1 The And-Scheduler

The and-scheduler was designed by Yang and is similar to the one used in PARLOG [42]: each worker has a private and a public queue. Work is sent to the private queue unless the public queue is empty. If a worker's queues are empty, it will try to find work in the public run queues of other workers in its team. For flexibility, the run queues in Andorra-I are designed to be accessible from both the front and the back. The engine always takes tasks from the front of the queue, but it can release tasks either to the front or to the back of the queue so that some tasks have higher priority than others (the programmer can specify this through annotations).

Synchronisation in a Team A team needs to synchronise either when a worker finds a failure or all the unexecuted goals are non-determinate. Andorra-I implements synchronisation through a set of flags shared by the team.

One of the flags indicates failure: any worker can set it to `true`. All workers test frequently (every reduction) if this flag is set. After the failure signal is accepted, workers can independently (i.e., in parallel) undo the bindings they made during the previous determinate phase. The master starts the non-determinate reduction only after all slaves have finished undoing their bindings.

Each worker has a run queue, hence detecting that there are no determinate goals is more complex than if a common queue was used. The current implementation uses a centralised method similar to JAM's. Every slave verifies if everyone else's run queue is empty. If so, it enters a `wait_for_master` state. Only the master can acknowledge deadlock.

10.3.2 The Or-Scheduler

Or-parallel scheduling in Andorra-I is provided by the same schedulers that do or-scheduling for Aurora; no adaptation was necessary since Andorra-I was made to conform to the Aurora scheduler interface [151]. Notice that, as far as the or-scheduler is concerned, an Andorra team behaves as an Aurora worker and no knowledge of the structure of a team is necessary. Andorra-I has been using the Bristol scheduler [8].

10.3.3 The Top-Scheduler

The Andorra-I top-scheduler, designed by Dutra [?], is a fundamental component of the Andorra-I system. This scheduler is responsible for the management of teams. During execution, the top-scheduler can create new teams, discard teams or move workers from one team to another. The top-scheduler dynamically adapts to situations where or-parallelism is dominant by creating more teams, to situations where and-parallelism is dominant by concentrating workers in a few teams, and to situations where some teams have more work than others by shuffling workers around teams.

Results for the top-scheduler [?] show that it performs as well as user distribution of teams in applications where one form of parallelism is dominant that it can perform better than using a fixed configuration of workers when both forms of parallelism are present.

10.4 Compiling Andorra-I

The initial version of Andorra-I was an interpreter. In order to obtain better performance a compiled version of Andorra-I was developed. The Andorra-I compiler is a part of the preprocessor. It compiles Prolog clauses into WAM-like abstract machine instructions. These instructions are then executed by an emulator which is part of the engine. We next present the abstract machine and describe how code for the abstract machine is generated by the compiler.

10.4.1 The Andorra-I Abstract Machine

The Andorra-I abstract machine is a WAM-style abstract machine for the execution of logic programs in the Basic Andorra Model. Development of this abstract machine was heavily influenced for Crammond's abstract machine for Parallel PARLOG, JAM [41].

The instruction set assumes the existence of a program counter, `PC`, giving the next instruction to execute, of a structure pointer `S`, pointing to the current argument in a compound term being currently written or read, of argument registers, $A_1, A_2, \ldots$, pointing to arguments in the current goal, and of temporary registers, $X_1, X_2, \ldots$, available to store temporary registers. Differently from the WAM or JAM, temporary and argument registers may be different. Argument registers may be temporary registers, or may be positions in a goal frame placed in the goal stack.

Andorra-I is a goal-stacking system (versus the WAM environment based system). The JAM uses environments to implement the sequential conjunction, which is currently implemented as a separate goal in Andorra-I. Being a pure goal-stacking implementation, the Andorra-I abstract machine does not need permanent variables.

Choicepoint Manipulation Instructions

The following instructions create and manipulate a choice-point:

```
try     L
retry   L
trust   L
```

As in the WAM, the instruction `try` creates a choicepoints, the instruction `retry` reuses the choicepoint, and the instruction `trust` reuses the choicepoint and discards it.

Determinacy Instructions

The first determinacy instruction verifies if an argument is instantiated, if so, it copies the argument to a temporary register, otherwise, the emulator creates a suspension-frame and jumps to alternative code.

`if_g_var_sus`	A_i, X_j, L_{sus}
`if_g_var_sus_hi`	A_i, X_j, L_{sus}
`if_s_var_sus`	Si, X_j, L_{sus}
`if_s_var_sus_hi`	Si, X_j, L_{sus}

The `_hi` instructions create a high-priority goal-frame (meaning that the and-scheduler should favour the corresponding goal over other goals).

The next instructions are called after the arguments are found to be instantiated. They correspond to simple test nodes in the determinacy graph. The first three instructions switch according to the type of the arguments, the last two according to its value.

`switch_on_term`	X_j, L_s, L_l, L_c
`switch_on_list`	$X_j, L_{List}, L_{Nil}, L_{Ot}$
`switch_on_constant`	$X_j, \{\{C_1, L_1\}, \ldots, \{C_n, L_n\}\}$
`switch_on_structure`	$X_j, \{\{S_1, L_1\}, \ldots, \{S_n, L_n\}\}$

Binary tests demand more work from the determinacy analyser. In this case, one extra argument or subargument in some structure must be fetched from the goal. The instructions must be able to access an argument inside a goal and verify if sufficiently instantiated to execute the builtin, and are similar to the previous instructions. Finally, the test is executed with the next instruction:

`call_bip` BIP, L_{yes}, L_{no}

This instruction has two addresses, one taken if the builtin succeeds, the other taken otherwise.

`wait`

The instruction `wait` is called when there are no more ways to make a goal determinate and the determinacy code must suspend until the goal becomes more instantiated.

We next show an example of determinacy code for the procedure shown in page 148. The determinacy code generated by the compiler for this procedure is:

```
          if_h_var_sus          L0,1,1
          switch_on_term        1,L1,fail,fail
L1
          switch_on_cons        2,fail,{{1,L2},{2,L3}}
```

```
L2
        if_h_var_sus        L0,2,1
        switch_on_term      1,L4,fail,fail
L4
        switch_on_cons      2,fail,{{1,L5},{2,L6}}
L0
        if_h_var_sus        L7,2,1
        switch_on_term      1,L8,fail,fail
L8
        switch_on_cons      2,fail,{{1,L5},{2,L7}}
L7
        if_h_var_sus        wait,3,1
        switch_on_term      1,L9,fail,fail
L9
        switch_on_cons      3,fail,{{1,L5},{2,L6},{3,L3}}
```

The labels `L6`, `L5` and `L3` correspond respectively to the compiled code for the second, first and third clauses. Note the use of the special label `wait` as place to execute when a test suspends and there are no alternative ways to make a goal determinate. Only when executing the code for `wait` will a goal actually suspend.

Read-Only Unification Instructions

The determinacy analyser calls the compiler to generate code for guards. This code basically tests if the arguments are sufficiently instantiated, that is if unification is read-only. The instruction will dereference its argument. If the argument is bound it executes much like a normal WAM instruction. If the argument is unbound it will create a suspension record and jump to the code indicated by the label L_{sus}. If the test fails it will jump to the code indicated by the label L_{fail}.

`wait_variable`	$A_i, X_i, L_{sus}, L_{fail}$	`wait_variable_ground`	$A_i, X_i, L_{sus}, L_{fail}$
`wait_value`	`Xn`, A_i, L_{sus}, L_{fail}		
`wait_constant`	`C`, A_i, L_{sus}, L_{fail}	`wait_nil`	A_i, L_{sus}, L_{fail}
`wait_structure`	`F`, A_i, L_{sus}, L_{fail}	`wait_list`	A_i, L_{sus}, L_{fail}

For instance, the instruction `wait_constant` will simply try to match its argument with a constant. Two cases deserve a more detailed explanation. Some tests can only be performed if their arguments are ground, thus the `wait_variable_ground` instruction. The `wait_value` instruction corresponds to read-only unification between any two terms.

The next instructions correspond to subarguments of compound terms. They are similar to the previous instructions:

`read_x_variable`	X_j, L_{sus}, L_{fail}	`read_x_variable_ground`	X_j
`read_x_value`	X_j, L_{sus}, L_{fail}		
`read_constant`	C, L_{sus}, L_{fail}	`read_nil`	L_{sus}, L_{fail}
`read_structure`	F, L_{sus}, L_{fail}	`read_list`	A_i

The instructions correspond to the different cases for subarguments of structures. We next give an example of guard compilation for the procedure:

```
a([a,X], f(X), Y) :- integer(Y), !, ...
...
```

The resulting code is:

```
wait_list            1,wait,L1              % a([
read_constant        a,wait,L1              %    a
read_list            wait,L1                %   ,
read_x_variable      4                      %    4
read_nil             wait,L1                %   ],
wait_structure       f/1,2,wait,L1          %   f(
read_x_value         4,wait,L1              %     X),
wait_x_variable      3,3,wait,L1            %   Y) :-
bip_1                integer,3,wait,L1      % integer(Y), !,
 ...                                        % ...
```

Several forms of `wait` instruction are shown. Notice that the `wait_x_variable` instruction must test if its argument is instantiated, otherwise `bip_1` could call the builtin `integer/1` with an unbound variable as the argument, and the procedure could commit to the clause before the actual value of the third argument was known.

Read-Write Unification Instructions

These unification instructions are called during head unification or when executing the builtin `=/2`. Contrarily to the previous instructions, they are allowed to bind their arguments. The main instructions are:

`get_value`	A_n, A_i	`get_x_value`	X_j, A_i
`get_x_variable`	A_n, A_i		
`get_constant`	`C`, A_i	`get_list`	A_i
`get_structure`	`F`, A_i	`end_structure`	
`get_list_var_var`	X_j, X_k, A_i	`get_list_var_val`	X_j, X_k, A_i
`get_list_val_var`	X_j, X_k, A_i	`get_list_val_var`	X_j, X_k, A_i

There are two versions of `get_value`, one corresponding to unifying with a temporary register, the other with an argument register. The instructions in the two last lines were introduced in JAM and speedup compilation for the common cases of pairs of variables.

The next instructions correspond to subargument unification. The first instructions are similar to the WAM's instructions.

`unify_x_variable`	X_n	`unify_x_value`	X_n
`unify_constant`	`C`	`unify_nil`	
`unify_void`			
`unify_list`		`unify_structure`	`F`
`unify_last_x_variable`	X_n	`unify_last_x_value`	X_n
`unify_last_constant`	`C`	`unify_last_nil`	
`unify_last_void`			

The `unify_list` and `unify_structure` instructions are used when the last subarguments in the term are respectively a list or a compound term. In this case unification carries on directly with the list or the compound term.

An important problem is *locking* of newly created terms. If their argument is an unbound variable, `get_list` or `get_structure` need to create a new term that will be eventually unified to the variable. Andorra-I avoids excessive locking by only actually unifying the newly created structure with the variable when the term has been fully created. The `last` instructions were designed for this purpose (note that the `unify_list` and `unify_structure` instructions are also `last` instructions).

We next show a simple example of unification code for Andorra-I. The example is shown first:

```
h([a,f(X),X], f(X,Y), Y).
```

The code generated by the Andorra-I compiler is shown next:

```
get_list              1           % h([
unify_constant        a           %    a
unify_list                        %   ,
unify_x_variable      4           %    T
unify_list                        %     ,
unify_x_value         5           %    X
unify_last_nil                    %   ],
get_structure         f/1,4       %    T = f(
unify_last_x_variable 5           %           X)
get_structure         f/2,2       %   f(
unify_x_value         5           %     X,
unify_last_x_value    3           %     Y)).
proceed
```

The `unify_list` instructions result in much more compact code. A small disadvantage is that if the first argument is unbound it will only be assigned the list when the instruction `unify_last_nil` is executed.

Write-Only Unification Instructions

The write-only unification instructions are an exact copy of the WAM unification instructions. They are as follows:

`put_x_variable`	X_n, A_i	`put_x_value`	X_n, A_i
`put_constant`	`C`, A_i	`put_nil`	A_i
`put_structure`	`F`, A_i	`put_list`	A_i

The instructions simply place a variable, constant, or pointer to a compound term in the heap in an argument register. Andorra-I tries to store variables in the goal frame, thus the instruction `put_x_variable` will try to initialise a variable in the goal frame.

`write_x_variable`	X_n	`write_x_value`	X_n
`write_structure`	`F`	`write_list`	
`write_constant`	`C`	`write_nil`	

The `write` instructions perform the same operation as the `put` instruction, but they perform these instructions on the subarguments of a compound term.

Goal Management Instructions

In Andorra-I goals may be launched from the body of a clause, or if the clause is a fact, the goal may simply succeed. The several instructions are as follows:

`create`	`F`	`create_one`	`F`
`create_first`	`F`	`create_last`	`F`,N_{Goals},N_{Args}
`neck`	N_{Goals}		
`proceed`			

The several versions of the `create` instruction are useful to determine when a goal frame can be reused, or when the arguments of a goal should be placed in the temporary registers instead of the goal stack. If a goal calls a single goal, the `create_one` instruction is used (note that if the operation is determinate, the new goal can reuse the goal frame of the other one). If several goals exist in the body of a clause, the goals are launched in reverse order. Thus the `create_first` instruction is the last to be called, and the `create_last` the first. Andorra-I tries to optimise register usage by placing the arguments for the leftmost call directly on the temporary registers.

We next show code for a very simple example procedure,

```
a(X) :- b(X), c(X), d(X).
```

The compiler generates the following code:

```
create_last     d/1,3,3     % d(
put_x_value     1,1         %   X)
create          c/1         % c(
put_x_value     1,1         %   X)
create_first    b/1         % b(X)
neck            3
```

For each goal, the code creates a goal frame, and then stores the arguments. The first instruction, `create_last`, has some bookkeeping information on the number of goals that will be launched and on the total number of arguments for these goals. Note that in this clause the compiler assumes that initially A_1 is stored in X_1, and that the goal for `b/1` is launched last but executed first, hence the arguments for `b/1` will be left in the temporary registers. Thus there is no need to do a `put_x_value` for `b/1`. Finally, the instruction `neck` closes down the code for the leftmost goal and results in executing a new goal.

10.4.2 The Andorra-I Compiler

The code shown previously is generated by the Andorra-I clause compiler. The clause compiler is an extension to the determinacy analyser that generates code for the head and body of a clause. The clause compiler originates from the AKL compiler [84], but has been adapted to the Andorra-I Abstract Machine described previously.

The determinacy analyser is responsible for the compilation of a procedure. The determinacy analyser calls the clause compiler once for every clause, receiving back a label with the address of this code. If the leaves of the determinacy code are of the form “commit to a clause” then the code will jump to the clause's label. If the leaf for the determinacy code is of the form “test if the clause can succeed in read-only mode”, the clause compiler is called for this clause with this special mode.

A procedure can also be called from a nondeterminate goal. The determinacy analyser recognises two cases. First, every procedure has a chain of `try`, `retry`, and `trust` instructions, with one instruction for every clause. This chain is the default nondeterminate code. The determinacy analyser also associates a special sequence of `try`, `retry` and `trust` instructions to the points in the determinacy code where the number of remaining matching clauses is significantly less than the total number of clauses. If a nondeterminate goal was suspended at one of these points, this sequence of `try`, `retry` and `trust` instructions is tried instead of the default one.

Compilation of a clause proceeds in several steps:

- The clause is reduced to a canonical form. The ordering of goals in the body is reversed; sequential conjunctions and pruning operators are transformed into special goals; finally, goals are rewritten into a form designed to facilitate compilation.
- A simple register optimiser is called. The register optimiser is only interested in variables appearing in the head of the clause or in the first goal of the body, and it tries to place those variables in registers such that it will avoid generating instructions such as `get_x_variable`. The optimisation algorithm is similar to the one presented by Debray [**?**].
- The actual instructions are generated. Instructions for the head are generated first, followed by instructions for each goal in the body. When generating instructions for a goal the compiler may be in one of three modes. In `head` mode the compiler generates instructions for read-write unification, in `body` mode the compiler generates instructions to place arguments in goals, and in `guard` mode the compiler looks at read-only arguments.

Some special care is taken to generate good code for common builtins such as unification, =/2, and for builtins appearing in the guard of a cut or commit.

- The registers which have not been allocated in the optimising step are allocated by a very simple register allocator.

The current compiler is quite simple but reasonably robust. Better results could be obtained by adding further optimisations to the compiler.

10.5 Andorra-I Performance Analysis

This section presents performance data obtained by Yang et.al. for Andorra-I. All timings were made on a Sequent Symmetry containing twelve processors. Each Andorra-I worker was supported by a separate processor.

10.5.1 Overall Performance

The data was obtained from a series of substantial Andorra-I applications and from some common benchmarks [?]. The applications have been mentioned in the previous chapters and include the aircraft flight scheduling system, the road edge recognising algorithm from the vision group at Bristol University, the deductive database system for reasoning about protein structure, and the natural language question answering system Chat-80. We also include two industrial applications originally developed for British Telecom: a workforce management system and a clustering algorithm.

The benchmarks include the famous naïve reverse benchmark, the 5x4x3_puzzle [161], warplan [176], a scanner originally developed for AKL [84], and a substitution decoding system developed for Andorra-I by Yang [190].

The twelve programs include a mix of Prolog style, committed-choice language style and Andorra style programming. The overall performance of Andorra-I for these benchmarks is shown in Table 10.1. Basis for comparison is the performance of "parallel Andorra-I" running on a single processor. This version is contrasted to the "sequential Andorra-I", that is to Andorra-I without the data-structures necessary to support parallel execution (scheduler, SRI model, locking and chunking scheme). The leftmost table shows the performance of a popular state-of-the-art Prolog system, SICStus Prolog, when meaningful. The tables to the right show the performance of "parallel Andorra-I" using ten processors and the performance (best and worst cases) and the performance of the Parallel PARLOG system (where applicable).

prog name	SICStus 2.1	Sequential Andorra-I	Parallel Andorra-I			JAM
			1 processor	10 processor		1 processor
				best	worst	
nrv400	2.17	1.26	1	8.25	8.07	1.25
bt_cluster	3.27	1.29	1	9.37	9.02	1.81
bt_wms	2.75	1.23	1	3.32	3.30	
road_markings	4.39	1.67	1	6.24	5.31	
chat_80_db5	2.24	1.26	1	7.30	6.59	
5x4x3_puzzle	6.56	1.47	1	9.66	9.56	
warplan	4.43	1.65	1	1.20	1.08	
protein_all	3.90	1.51	1	6.81	5.51	
protein_1st	3.25	1.48	1	2.78	2.02	
fly_pan	0.02	1.25	1	6.88	5.07	
scanner		1.48	1	5.47	4.41	
cipher		1.51	1	5.65	4.67	

(Relative Speed)

Table 10.1: The Overall Performance of Compiler-Based Andorra-I

The performance results from Andorra-I are very positive. Although the system is not optimised, it shows a sequential performance typically 3 slower than SICStus, and 1.5 slower than Parallel PARLOG. Moreover, the system's ability to extract different forms of parallelism allows it to give good speedups for very different applications.

One should notice the flexibility of the Basic Andorra Model. By exploring parallelism Andorra-I can do well for Prolog applications and committed-choice language applications. Andorra-I also gives good results for applications which are hard to write in Prolog or the committed-choice languages but that benefit from the Basic Andorra Model.

We next discuss those results in more detail.

10.5.2 Computation Size

Andorra-I and Prolog use different selection functions, and one can expect different computation sizes. Table 10.2 lists the number of resolution required for the benchmark program in both Prolog and Andorra-I. We count each successful reduction and a call to a builtin predicate as one resolution. The table also shows the number of reductions performed by an Andorra-I system executing with ten processors.

prog name	Prolog	Andorra-I	
		1 processor	10 processors
nrv400	80612	80612	80612
bt_cluster	498617	498617	498617
bt_wms	889183	889183	889183
road_markings	23338	23338	45376
chat_80_db5	6373	6373	6373
5x4x3_puzzle	135108	135108	135108
warplan	48102	47792	429511
protein_all	125898	106132	117902
protein_1st	26334	18846	66685
fly_pan	7976002	41019	42319
scanner	NA	118467	118215
cipher	NA	10324	11096

Table 10.2: Number of resolutions in Prolog and Andorra-I

For determinate benchmarks, such as `nrv400`, the number of resolutions is identical in Prolog and Andorra-I, as one would expect. For some non-determinate benchmarks, such as `chat_80_db5`, `5x4x3_puzzle` and `warplan`, early execution of goals does not change the search space much. But a benchmark such as `fly_pan` shows that early execution of determinate goals can constrain the search-space. The program `fly_pan` was designed for the Basic Andorra Model, and thus running it under Prolog's depth-first left-to-right order is very inefficient. The benchmark `protein`, although originally designed for Prolog, also benefits from the Basic Andorra Model. The benchmarks `scanner` and `cipher` are examples of two applications where it is meaningless executing the program under Prolog's selection function.

The last column in table 10.2 shows that for most applications, and as one would expect, the search space of the sequential and parallel versions is quite similar. The exceptions are due to *speculative or-parallelism*, that is or-branches that will be pruned and thus never taken in the sequential execution, but that might be taken (before being pruned) by idle teams in the parallel system.

10.5.3 LIPS

Traditionally, logic programming systems use *LIPS*, Logical Inferences per Second, as a measure of performance. Table 10.3 gives the performance of the system in terms of thousands of LIPS, or KLIPS.

prog name	SICStus 0.7	Seqential Andorra-I	Parallel Andorra-I		
			1 processor	10 processors	
				best	worst
nrv400	23.3	13.5	10.7	88.6	86.7
bt_cluster	16.3	6.5	5.0	46.9	45.1
bt_wms	13.0	5.8	4.7	15.8	15.7
road_markings	8.9	3.4	2.0	24.8	21.1
chat_80_db5	4.2	2.4	1.9	13.8	12.5
5x4x3_puzzle	8.8	2.0	1.3	13.0	12.9
protein_all	6.2	2.0	1.3	10.2	8.2
protein_1st	6.2	2.0	1.4	13.4	9.7
fly_pan	14.8	3.9	3.1	22.4	16.5
scanner	NA	5.0	3.4	18.4	14.8
cipher	NA	1.9	1.2	7.5	6.2

Table 10.3: Overall performance in KLIPS

Notice that where the search-space is very different (e.g, in `fly_pan`) the LIPS rating is not directly comparable.

In general, a ten processor Andorra-I system can go up to 89 KLIPS on a Symmetry machine. The variation in LIPS can be explained by several factors. Programs where inferences are more expensive are programs that create more choice-points, or programs that suspend often and where goals need to be pushed to the goal stack.

10.5.4 Sequential Performance

Table 10.1 gives a perspective on how well Andorra-I does versus a state-of-the-art Prolog system, SICStus Prolog. Table 10.4 details these results by comparing the sequential version of Andorra-I versus SICStus Prolog.

Table 10.4 shows “sequential Andorra-I” being about 1.7 to 4.5 slower than SICStus. Andorra-I does better on the determinate branches where it is about 2.5 times slower, de-

prog name	SICStus 2.1 time in ms	Seq. Andorra-I time in ms	Ratio
nrv400	3459	5940	1.71
bt_cluster	30490	77320	2.53
bt_wms	68090	152020	2.32
road_markings	2600	6840	2.63
chat_80_db5	1500	2670	1.78
5x4x3_puzzle	15300	68280	4.46
warplan	6849	18370	2.68
protein_all	20189	52110	2.58
protein_1st	4250	9330	2.19
fly_pan	543520	10430	0.02
scanner	NA	23600	
cipher	NA	5510	

Table 10.4: Comparison between SICStus Prolog and Sequential Andorra-I

pending on overheads from goal frame manipulation, and from suspending non-determinate goals. Andorra-I does worse for non-determinate benchmarks, mainly because of inefficiencies in the engine when executing non-determinate goals. Note that Andorra-I does relatively better in some non-determinate examples because it can use the determinacy code for indexing.

It is interesting to compare the compiled and interpreted version. The results are shown in table 10.5.

The differences for the compiled version confirm our previous observations. The compiler will do better where the almost-fixed overheads, such as suspension and choice-point manipulation, will not be significant. Thus, the good results in `nrv400`. The particularly good results in `bt_wms` are due to better compilation of determinacy code in the compiled version, plus some extra optimisations.

10.5.5 Parallel Speedups

Table 10.1 showed the speedups obtained by Andorra-I using ten processors. These speedups originate from and-parallelism, or-parallelism or both forms of parallelism. Table 10.6 gives a more detailed view of the parallelism by comparing Andorra-I with and-parallel only or or-parallel only systems. Table 10.6 gives the best speedup of each system for some

prog name (time in ms)	Par. Andorra-I (1 worker) interpreter	compiler	Ratio
nrv400	33920	7510	4.52
bt_cluster	364283	99720	3.56
bt_wms	1345340	187060	7.19
road_markings	25530	11420	2.24
chat_80_db5	12130	3360	3.61
5x4x3_puzzle	214290	100390	2.13
protein_all	100390	78840	1.27
protein_1st	18420	13840	1.33
warplan	32710	30370	1.08
fly_pan	63150	13020	4.85
scanner	130460	35070	3.72
cipher	11150	8310	1.34

Table 10.5: Interpreted versus Compiled Andorra-I

benchmarks. The or-parallel systems are Aurora and Muse, the and-parallel system is Crammond's Parallel PARLOG.

prog name	Andorra-I	JAM	Aurora	Muse
nrv400	8.25	8.37		
bt_cluster	9.37	9.70		
chat_80_db5	7.30		7.30	5.91
5x4x3_puzzle	9.66		9.51	8.69
warplan	1.20		2.63	1.06
protein_all	6.81		9.49	8.64
protein_1st	2.78		4.10	3.12

Table 10.6: Speedups (10 processors) in Andorra-I, JAM, Aurora, and Muse

The speedups show that Andorra-I does almost as well as the one form of parallelism only systems for most benchmarks. The main exceptions are systems where the or-parallel system Aurora benefits from more advanced schedulers that perform much better for speculative work (such schedulers can be ported to Andorra-I).

It is also of some interest to show how Andorra-I benefits from combining or- and and-parallelism. Table 10.7 shows the results obtained from trying the system with and-parallelism only, or-parallelism only and both forms of parallelism. The selected benchmarks have both forms of parallelism.

prog name	And Parallel Only	Or Parallel Only	And + Or Parallel
fly_pan	6.29	1.16	6.88
scanner	5.07	2.83	5.47
cipher	2.52	5.43	5.65

Table 10.7: Speedups (10 processors) for both and- and or-parallelism

The results were obtained with a top-scheduler [?], that is, by dynamically adapting the configuration of teams in order to extract the best available parallelism. Clearly by combining the several forms of parallelism Andorra-I can obtain better speedups than by exploiting a single form of parallelism.

10.6 Summary

This chapter described Andorra-I's engine and schedulers and analyses its performance. The design of the engine extends previous work on the or-parallel system Aurora and on the implementation of the committed choice languages such as PARLOG, and adds new features such as the chunk scheme, updatable variables and the sideways-linked chain. Andorra-I also uses several schedulers. The or-scheduler was originally designed for Aurora, the and-scheduler uses ideas from Parallel PARLOG and the top-scheduler is responsible from the configuration of teams.

The Andorra-I compiler compiles to a WAM-like abstract machine code that is emulated by the engine. The abstract machine includes instructions to support read-only unification, read-write unification, and write-only unification. Control instructions manipulate goals. Several abstract machine instructions have been developed to support the determinacy code and the manipulation of choice-points.

Finally, the chapter presented performance data for Andorra-I. The data showed the Andorra-I performance to be quite good, both in its sequential performance and in its ability to exploit parallelism. It also showed that Andorra-I can be used to reduce the search space.

Chapter 11

Related Work

In this chapter we present several languages and systems that relate to our work in Andorra-I, and particularly to the Basic Andorra Model. We present the NUA-Prolog system (an evolution of the Parallel NU-Prolog system), Andorra and Flat Andorra Prolog, and Pandora.

We also discuss computational models that try to improve on the Basic Andorra Model. IDIOM [66] adds to the Basic Andorra Model independent and-parallelism, therefore providing a practical framework for the exploitation of all three forms of parallelism. Warren's Extended Andorra Model [182] provides a powerful and elegant computational model for the implicit exploitation of both and- and or-parallelism in logic programs. AKL [70, 84] is a novel programming language that uses the same principles, but where the control constructs are more explicit.

11.1 Andorra-I Related Languages

The Basic Andorra Model can also be used as the basis for new logic programming languages. We briefly comment on a few languages that are based in, or close to the Basic Andorra Model.

11.1.1 Parallel NU-Prolog and NUA-Prolog

Parallel NU-Prolog [118] provides parallelism for a deterministic subset of NU-Prolog. Declarations are used to specify where one can exploit parallelism. The `lazyDet` declarations are used for deterministic procedures in the style of P-Prolog, whereas the `eagerDet` declarations are used for committed-choice style procedures. Naish suggests using a preprocessor to provide the declarations. The style of execution is similar to the Basic Andorra Model: non-deterministic procedures cannot execute while and-deterministic goals are executing in parallel, but only fully deterministic procedures are considered for parallel execution. Naish also proposes an all-solution predicate in the style of PARLOG [25].

The Parallel NU-Prolog was eventually extended by Palmer and Naish to form the NUA-Prolog system [124]. This new system includes Andorra predicates, given by the declaration `andorra`, and that may have modes associated. The marker predicates `$sa` and `$fa` are used to explicitly start and finish an Andorra execution (outside this boundary Andorra goals behave as normal goals). An and-parallel version of the NUA-Prolog system has been implemented and obtained speedups similar to Andorra-I.

11.1.2 Andorra Prolog and Flat Andorra Prolog

The language Andorra Prolog [69] was presented by Haridi and Brand as a new language able to combine Prolog and Committed Choice Languages. Andorra Prolog relies on the Basic Andorra Model. In this language, clauses are of the form $H\colon -G_t, G_b.$, and only G_t, the guard, is used to test for determinacy. Don't care non-determinism is exploited by adding the commit operator "|", as in FGHC (guards to the commit must be flat and every clause in the procedure must have a commit). The commit operator may also appear in the body of the clause. In this case it will not be used for determinacy and it will be implemented as Aurora's commit. Only a limited form of cut is allowed in this language. Finally, `delay` declarations force a goal to be executed only when determinate. Flat Andorra Prolog [68], also known as AP(:,|), includes only the guard operators, commit (now only used in guards) and the *wait* operator that replaces `delay` declarations. The construct $H\colon -G_d : Bp$, means that non-determinate resolution of a goal will wait until H and G_d have succeeded.

The language Andorra Prolog can be seen as a subset of Andorra-I Prolog. Flat Andorra Prolog diverges further from traditional Prolog, and was a stepping stone in the design of AKL [70], discussed later in this chapter.

11.1.3 Pandora

The Pandora language was designed by Bahgat and Gregory [4]. Pandora uses the Basic Andorra Model to extend PARLOG. Pandora relations may be either *and-parallel* or *deadlock*. And-parallel (or don't-care or committed-choice) relations are defined by normal

PARLOG procedures (called and-parallel procedures). Deadlock or don't-know relations may be called from anywhere except from the guard of an and-parallel relation. They are defined by an unmoded procedure, that is without mode declarations, which comprises a sequence of clauses of the form $p(t_1, \ldots, t_n) \leftarrow D : B.$, where D is a *det-guard*. Det-guards are flat guards, that is they only include some predefined primitives, whereas B may be any Pandora conjunction. The definition of deterministic goals (corresponding to the Basic Andorra Model's determinate goals) in Pandora is given using det-guards:

A goal for p is deterministic if and only if at least $k-1$ clauses have false (unsatisfiable) det-guards, where k is the number of clauses in p's definition.

Bahgat presents several applications of Pandora [3]. From the original paper, particular emphasis has been given to problems in constraint logic programming, and the authors show an elegant implementation of domain variables. The Pandora language has been implemented on top of PARLOG (thus, and-parallel procedures may be nonflat), but the example Pandora programs have also been translated to Andorra-I Prolog. In her thesis [3], Bahgat presents an abstract machine for the implementation of Pandora, PAM, based on Crammond's abstract machine for Parallel PARLOG. It was considered that in most applications of Pandora and-parallelism would outweigh or-parallelism, hence PAM is designed to exploit only dependent and-parallelism.

Pandora does not specify which non-determinate goal to select, but Bahgat proposes a meta-level deadlock handler as part of the language. Such a feature would provide better search, but it might result in serious overheads and interfere with and-parallelism.

11.2 IDIOM

IDIOM integrates independent and-, dependent and- and or-parallelism. IDIOM tries to use the concepts that have been demonstrated to be successful for systems exploiting individual forms of parallelism. In this way one can be reasonably confident that a final implementation will be efficient, at least for those programs which exploit only one kind of parallelism. For the programs which exploit more than one kind of parallelism one can still hope that the integrated system would be efficient given that the properties of logic programs which give rise to the three forms of parallelism are largely orthogonal. This principle has been applied before in Andorra-I.

The integrated framework IDIOM (Integrated Dependent- Independent- and Or-parallel Model), is based on the Basic Andorra Model and on Gupta's Extended And-Or Tree Model [62] (although other models for the exploitation of both independent and-parallelism and or-parallelism can be supported [61, 60]). From the former we borrow the principle of eager execution of determinate goals and of or-parallel execution of non-determinate goals; from the latter we borrow techniques for parallel execution of independent (non-determinate) goals, and ways for combining their solutions. Since IDIOM is based on the Basic Andorra Model it supports both Prolog like languages as well as (flat) Committed Choice Languages (such as FGHC). Throughout the design of IDIOM special emphasis was given to the support of Prolog.

11.2.1 The Computational Model

Like the Extended And-Or Tree Model, IDIOM uses Conditional Graph Expressions (CGEs) [73] (see section 3.2.2 for a detailed discussion) to express independent-and parallelism.

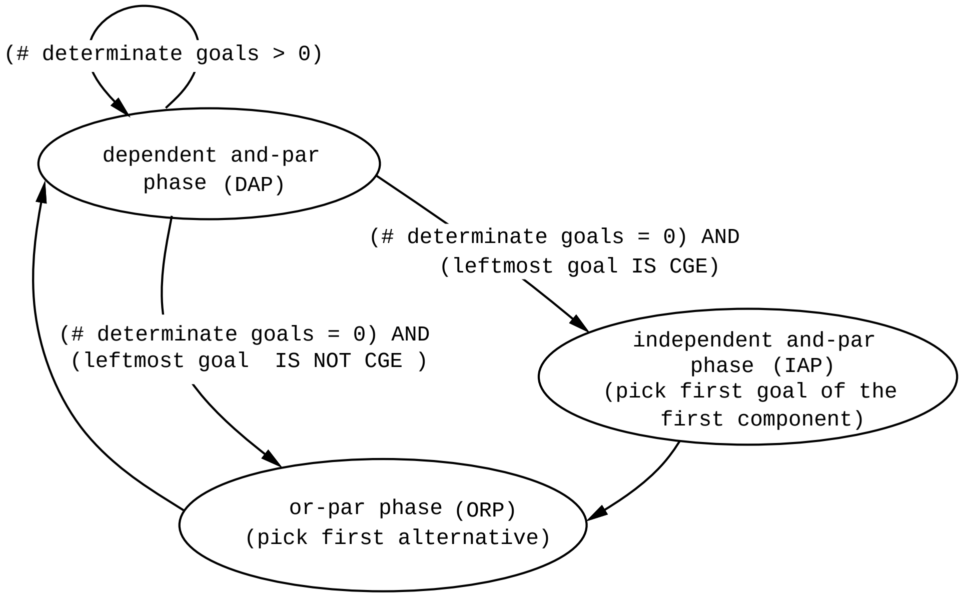

Fig 1: Phases in Parallel Execution

Figure 11.1: Execution Flow in IDIOM

IDIOM executes determinate goals first. When no more determinate goals are available, then either an non-determinate reduction or the independent execution of several goals may be started. Forward execution in IDIOM thus alternates between 3 phases(see figure 11.1): the Dependent And-parallel Phase (DAP), the Independent And-parallel (IAP) phase, and the Or-parallel (ORP) phase. In the DAP phase all goals which can be determinately reduced are evaluated in and-parallel (including those inside the CGEs) until none are left. One goal (say, the leftmost) is then examined to see if it is (i) a simple goal; or (ii) is part of a CGE. In case (i) a non-determinate reduction is tried while in case (ii) independent and-parallelism is initiated. In the ORP phase the first alternative to the goal is selected (other alternatives being available for or-parallel processing), head unification is performed, and the DAP phase is entered again. In the IAP phase, firstly the condition in the CGE is evaluated. If true, the components of the CGE are made available for independent-and parallel processing. In practice an implementation would select one, (possibly the leftmost), component for immediate execution and make the remaining components available as independent-and parallel work. The ORP phase is then entered for that component of the CGE; and one goal of the selected component, say the leftmost, will be reduced non-determinately, thus allowing for or-parallelism inside independent and-parallelism. If the condition in the CGE evaluates to false, the ORP phase is entered immediately to one goal of the CGE (usually the leftmost) in or-parallel. Execution of the continuation of the CGE is possible when a solution for the entire CGE (that is, for all the components of the CGE) is found. The resulting solution is then made available to the environment of the CGE which then starts execution of the remaining goals in the DAP phase. The control algorithm is abstracted in Figure 11.1.

Recomputation: the previous discussion assumes a mechanism for the calculation of the cross-product. The original mechanism for IDIOM [66] was the one proposed in the Extended And-Or Tree model, where a special cross-product node is used to prevent recomputation of solutions. Simpler algorithms such as the one used in &-Prolog [75], later extended in the ACE [61] and the PBA [60] models, also apply. Such mechanisms have the disadvantage that they recompute solutions, and hence they may generate more computation than a model which does not recompute solutions, but they have the important advantages of being simpler and more efficient to implement, and of providing better support for Prolog [65].

11.2.2 An Example

To illustrate execution in IDIOM we take the program which finds "cousins at the same generation who have the same genetic attributes", a modification of a program from Ullman [167]. Given two persons and a list of attributes common to both persons, the program succeeds if it find a common ancestor and returns the set of attributes that are common to the cousins and to all the ancestors of the cousins up to and including the common ancestor.

The top-level procedure `sg(X1, X2, Pi, Pf)` defines the relation "cousins at the same generation with the same genetic attributes". It receives as the first two arguments the names of two persons, `X1` and `X2`, and as a third argument a list of attributes they are know to share, `Pi`. If `X1` and `X2` are the same person `X`, the relation returns the intersection of the attributes `Pi` with the attributes of the person `X`. Otherwise, the same relation is called recursively for a parent of `X1` and for a parent of `X2`. The procedures `set_xion` and `in` define intersection of two sets (note that the two sets do not contain duplicates). Finally, the assertion `parent(X, Y)` means `Y` is the parent of `X`, `attributes(X, Px)` means `Px` is a list of attributes of `X`, and `set_xion(S1, S2, S3)` represents that set `S3` is the result of intersection of sets `S1` and `S2`.

The program is shown next.

```
sg(X, X, Pi, Pf ) :- attributes( X, Px ), set_xion( Px, Pi, Pf ).
sg(X, Y, Pi, Pf ) :- X ≠ Y,
         parent(X, Xp), attributes(Xp, Pxp),
         parent(Y, Yp), attributes(Yp, Pyp)),
         set_xion(Pi, Pxp, Pxi), set_xion(Pxi, Pyp, Pii),
         sg(Xp, Yp, Pii, Pf).

set_xion( S1, S2, S3 ) :- set_xion( S1, S2, [ ], S3 ).
set_xion( [ ], _, In, In ).
set_xion( [X|T], S, In, Out ) :- in( X, S, In, IR ),
                                 set_xion( T, S, IR, Out ).
in(_, [ ], In, In).
in(X, [X|_], In, [X|In]).
in(X, [Y|T], In, Out) :- X ≠ Y, in( X, T, In, Out).

parent(fred, frank).
...

attributes(fred, [brown-hair, green-eyes, large-build]).
...
```

In this example there is independent and-parallelism in the second clause of `sg/4` (that is, when a parent is found for both X and Y), and the CGE annotator would annotate the second clause for `sg`. The corresponding program, where the & operator is used to represent independent and-parallelism, is shown next:

```
sg(X, X, Pi, Pf ) :- attributes( X, Px ), set_xion( Px, Pi, Pf ).
sg(X, Y, Pi, Pf ) :- X ≠ Y,
         ( indep(X, Y) => (parent(X, Xp), attributes(Xp, Pxp)) &
```

```
                    (parent(Y, Yp), attributes(Yp, Pyp)) ),
    set_xion(Pi, Pxp, Pxi), set_xion(Pxi, Pyp, Pii),
    sg(Xp, Yp, Pii, Pf).
```

Consider the query:

```
?- sg(fred, john, [brown-hair, green-eyes], Att).
```

where the third argument is a list of attributes common to `fred` and `john`, and we want to find out if they are cousins of the same generation and if so, the common genetic attributes inherited from their common ancestor.

We start from the query goal. We check if it is determinate, which indeed it is, since only the second clause matches. Head unification is then performed and the body of the second clause is inserted in the goal-chain. Only the `set_xion(Pi, Pxp, Pxi)` subgoal is determinate (since `Pi` is known) so it is reduced next. The call to `in` from within `set_xion` soon suspends; however, the recursive call to `set_xion` can still continue determinate execution as long as there are list-elements available in `Pi`. This will result in a number of calls to `in` all of which would be suspended on `Pxp`. Eventually, no determinate goals are left, and the DAP phase is exited. Since the leftmost subgoal is a CGE, the IAP phase is entered. The independence condition evaluates to true, and parallel execution of the two components is started. The parent goals are executed in or-parallel, and for each alternative of `parent` the corresponding `attributes` goal is executed determinately. As soon as an `attributes` goal is executed, and a binding for `Pxp` is generated, all the suspended `in` goals are rendered determinate and thus awakened; but their execution is delayed until the execution of the CGE is over [66]. Rather, the cross-product is computed and a tuple is selected for executing the continuation of the CGE. The determinate execution of the delayed `in` goals can then be started. (The two alternatives for the two `parent` goals, give rise to four cross-product tuples, which are pursued in (or-) parallel. The execution of the continuation of the CGE is the same for all tuples.)

As soon as the execution of `in` instantiates `Pxi` and causes the first list-element to appear in it, the second call to `set_xion` becomes determinate and begins execution. As more and more list-elements of `Pxi` are generated by the multiple `in` goals, they are determinately consumed by the second `set_xion`. Notice that the recursive call to `sg` was determinate as soon as the CGE was done, since `Xp` and `Yp` were known, and could also start executing. As soon as an element of `Pii` is available, the first `set_xion` call inside recursive call to `sg` becomes determinate and can begin execution. Thus, it is as if a data pipeline has been set up between calls to `set_xion`, calls to `in` and the recursive call to `sg`, which indeed gives rise to dependent-and parallelism. Or-parallelism is exploited in the execution of the parent goals, and independent-and parallelism in the execution of the CGE. Thus, all three forms of parallelism are exploited in the IDIOM-based execution of this program.

11.3 The Extended Andorra Model

The Extended Andorra Model (EAM) was designed by Warren [182] to extract all forms of parallelism available in logic programs while also minimising the number of inferences performed. The EAM extends the Basic Andorra model in that it allows the non-determinate and-parallel execution of goals as long as they do not bind external variables.

The EAM is formally defined using a set of rewrite rules which manipulate and-or trees, represented with the aid of *and-*, *or* and *choice-boxes.*

And-box: $[\exists X_1, \ldots, X_m : \sigma \& C_1 \& \ldots \& Cn]$

Or-box: $\{C_1 \vee \ldots \vee Cn\}$

Choice-box:$\{G_1 \% C_1 \vee \ldots \vee Gn \% Cn\}$

And-boxes are created when launching a new clause, the variables X_1 to X_n represent the variables created in the box and σ represents the constraints on external variables. Or-boxes correspond to several clauses and choice-boxes are used to delimit the scope of the pruning operator % (which may be a cut or commit).

The basic operations of the EAM are:

Reduction: $G \rightarrow \{[\exists Y_1 : \sigma_1 \% C_1] \vee \ldots \vee [\exists Y_n : \sigma_1 \% C_n]\}$;

Promotion: $[\exists X : A \& [\exists Y, W : Z = \theta(W) \& C] \& B]$
$\rightarrow [\exists X, W : X = \theta(W) \& A \% [\exists Y : C] \& B]$;

Substitution: $[X, Z : Z = \theta(W) \& C] \rightarrow [\exists X : C\theta]$;

Forking: $[\exists X : \{C_1 \vee \ldots \vee C_n\} \& G] \rightarrow \{[\exists X : C_1 \& G] \vee \ldots \vee [\exists X : C_n \& G]\}$

Reduction expands a goal G into an or-box, promotion allows the promotion of constraints from an inner and-box to an outer and-box, substitution applies a constraint on a variable and forking distributes a conjunction across a disjunction.

11.3.1 Control in the EAM

Basically, the EAM uses a control scheme where an and-box suspends if it constrains an external variable. Forking is the operation that can lead to duplication of work, hence it is only applied when all the other operations cannot be applied further (this follows the Extended Andorra principle that delays guessing a variable's binding as long as possible). If several goals can produce a non-determinate binding for a variable, the leftmost one will be chosen. Notice that, before forking is started, a binding for variable X has to generated by one of the C_i. After forking this binding can be promoted in the newly created and-box. A forking step followed by a promotion step is called *global forking*.

The EAM allows the parallel early execution of non-determinate goals. In some cases, this may create speculative and-work, as some other goals might fail. This may result in a larger search space or even lead to non-termination. Ideally one would like to provide an implicit control mechanism to prevent this problem. Detecting speculative work depends knowing a priori whether the computation will terminate, which is very difficult to know beforehand (in fact it might be undecidable because the EAM allows any computation to go ahead as long as it does not bind variables, and in general deciding if a goal will fail is undecidable). In practice, an approximation through global analysis should be possible. In the current definition, the sequential conjunction may be used by the programmer to prevent speculative computation.

One of the aims of the Extended Andorra Model is to perform the least number of reductions to obtain the solutions for a goal. To obtain this effect one often wants to do computations for different goals as independently as possible, and eventually only combine the solutions for the different goals as late as possible. This is similar to the issues of cross-product calculation found in independent and-parallelism (see section 3.2.2).

Notice that in order to obtain maximum independence between goals it is necessary to limit the number of goals that can be allowed to delay global forking. That is, global forking should happen as soon as the goals that can bind the variable, and only these goals, cannot apply any other operations. A solution to classify these goals is to give scope to variables, possibly obtained through rewriting clauses in mini-scope form.

11.3.2 Lazy Copying

In the previous description, a goal will suspend when nondeterminately trying to bind an external variable, irrespective of whether it is the producer (the goal that generates bindings for) or the consumer (the goal that reads bindings for) that variable. If the producer, there is no need for it to suspend. The EAM tries to gain more parallelism by letting the producer continue. This is implemented not by creating a physical copy for every alternative of the producer but by creating pointers to the consumer goal. The single copy of the consumer goal is called the *phantom goal.* Physical copying is done only when the consumer is completely suspended. The copy of the consumer is done in the lazy fashion, hence the name *lazy copying.* The forking rule now becomes:

$$[\exists X : \{C_1 \vee \ldots \vee C_n\} \& G] \rightarrow \{[\exists X : C_1 \& G_p] \vee \ldots \vee [\exists X : C_n \& G_p]\}$$

where G_p is a reference to a single goal. One can do even better, by copying parts of G as soon as they suspend.

The EAM now includes a classification scheme for variables as *guessable*, *non-guessable* or others, which provides control for lazy-copying. This classification might be provided by the programmer, or generated automatically.

A variable is annotated as guessable if it is certain it is going to be guessed non-determinately. In this case, as soon as a binding is generated by a producer for a guessable variable global forking with lazy copying is performed. In contrast, global forking on non-guessable variables is only performed if there is no way to restart the computation.

11.3.3 An Example of the EAM

We next discuss a simple example of the operation of the EAM (taken from [67], itself based in an example from [182]). The program and query are:

```
sublist([], _).
sublist([X|L], [X|L1]) :-
        sublist(L, L1).
sublist([X|L], [Y|L1]) :-
        sublist([X|L], L1).

?- sublist([X,Y],[c,a,t,s]), sublist([X,Y],[l,a,s,t]).
```

The predicate `sublist` succeeds if the first argument is a sublist of the second argument.

Without lazy copying, the EAM will operate by first executing the two goals independently. Both `sublist` goals will create branches for the several possible alternatives, but both will eventually suspend when trying to promote bindings for X. Eventually, the entire computation will stop and wait for nondeterminate promotion. The nondeterminate promotions will result in the left goal attributing values to X: after substitution these values will be combined with the values from the right call to `sublist` and eventually only $X = a$ is found to match. The process is then repeated for Y resulting in finding the solutions $Y = t$ and $Y = s$.

In order to use lazy copying, one should provide annotations to specify that the variables X and Y are guessable by the left call to `sublist`. In this case, the first calls to `sublist` is allowed to immediately bind X (that is, to guess X). After they do so, it can copy the parts of the other call to the `sublist` goal that have already suspended. Lazy copying gives more parallelism because the left `sublist` is allowed to proceed earlier. A disadvantage is that eager execution of the producer may generate extra work, because in this case the search space of the producer is not being as tightly constrained by the consumer.

11.3.4 Discussion

Gupta has implemented a Prolog interpreter for the EAM [67]. His examples show that the EAM can lead to a substantial reduction on the number of inferences, even when comparing to the Basic Andorra model. Lazy copying can provide more parallelism. Even considering the overheads caused by the extra complexity of the model, one can expect that the EAM will allow implicit parallelism in applications where the current models are rather limited.

There are currently no implementations of the Extended Andorra Model. Still, we believe that the tools we developed for Andorra-I should be useful for the Extended Andorra Model. The principles of the sequencer should still apply when the Extended Andorra Model is used for parallel execution. Results of abstract interpretation will be of interest to generate scope information for variables. Finally determinacy analysis should still be useful in prevent unnecessary or-boxes.

11.4 The Andorra Kernel Language

Kernel Andorra Prolog, designed by Haridi and Janson [70], is a language framework based on similar ideas to the Extended Andorra Model. The framework adds guard operators to delimit local execution and supports constraint operations [137]. The Andorra Kernel Language (AKL) [70, 84] is a general concurrent logic programming language based on Kernel Andorra Prolog.

Clauses in AKL always have a guard. All the clauses in the same procedure must have the same guard operator, which can be cut, commit or wait. The wait operator is similar to Saraswat's don't-know commit [137]. The computation model for AKL is again defined in terms of and-boxes $and(C)_V$ where V is the set of variables local to the and-box, or-boxes and choice-boxes. And-boxes are said to be quiet if they do not contain bindings (constraints) for external variables. The main execution rules are:

1. Local forking: $A \Rightarrow choice(and(G_1)_{V_1}\%B_1, \ldots, and(G_n)_{V_n}\%B_n)$
2. Determinate promotion:
   $and(R, choice(C_V\%B), S)_W \Rightarrow and(R, C, B, S)_{V \cup W}$
3. Nondeterminate promotion:
   $and(T_1, choice(R, C_V : B, S), T_2)_W \Rightarrow$
   $or(and(T_1, C, B, T_2)_{V \cup W}, and(T_1, choice(R, S), T_2)_{,W}))$

The first rule corresponds to the rewrite of an goal A, the second correspond to the simplification of a box when C_V is solved (and if a pruning operator if it is quiet). The last rule promotes a wait-guarded goal with a solved guard when R or S is non-empty as long as the rewrite is done within a stable and-box. An and-box is *stable* if (i) no other rule is applicable to any of its subgoals, and (ii) the and-box satisfies the constraint satisfiability condition that no possible changes in the environment will lead to a situation where a non-trivial rule is applicable in the and-box. Other rules include rules for the *cut*, *commit*, several synchronisation rules and rules for the aggregation operator *bagof*.

In general, determining if an and-box is stable is undecidable. Janson and Montelius' sequential AKL system [85] uses the simple rule that the all and-boxes must be stable if the entire computation has stopped. A more powerful condition is that an and-box must be stable when there are no constraints on any external variables.

AKL allows local bindings to external variables. That is, the guard of a goal is allowed, to bind external variables which are then only locally visible. Local bindings can sometimes detect failure as in:

```
a(1,2).          a(2,2).

?- a(X,X).
```

Execution of this query will result in committing to the second clause without any need to do nondeterminate promotion, usually a very expensive operation. Note that allowing local bindings in the execution of guards may lead to much wasted work if the goal was not sufficiently instantiated, cf. the case of deep cuts, where if the arguments are not sufficiently instantiated the computation may loop.

11.4.1 Pruning in AKL

The original version of AKL only accepted "quiet" cuts and "quiet" commits, where the definition of quietness in AKL is more restrictive than ours. An occurrence of C_V (that is, a conjunction of goals with local variables V) in G is *quiet* in AKL if $\theta(C)$ does not restrict the environment of C_V outside V [50]. This definition does not include the cases where C_V restricts the environment determinately. In this work, we use the name *silent* pruning to represent this form of quiet pruning.

Comparing both forms of pruning, silent pruning has the advantage that it can be implemented locally, that is, it needs not to test whether alternative clauses can bind a variable. This makes run-time checking of silent pruning much easier (verifying if a binding is determinate is in general as complex as verifying if a goal is deterministic), hence silent pruning is arguably more appropriate for a language that heavily relies on it. On the hand, quietness as we defined it is a more general solution to the problem of whether coroutining is possible in the presence of pruning, and therefore to our task of supporting Prolog.

11.4.2 An Analysis of AKL

A sequential compiled AKL system was implemented by Janson and Montelius [85]. Non-determinate promotion is done on the leftmost goal when the entire computation suspends. For determinate programs performance is about four times slower than SICStus Prolog [84]. This is partly due to the use of copying in the implementation of AKL. Also, better compilation of flat guards (in the style of Andorra-I) is expected to improve performance for many applications.

One issue in AKL programming is the possibility of looping. The language makes control explicit, thus it is the programmer's task to prevent such situations. On the other hand, the sophistication of the execution rule and particularly the fact that local bindings to external variables are allowed can make control quite difficult to understand (particularly in a parallel implementation). A solution proposed by Palmer [123] is to use mode declarations to specify when and-boxes can be executed.

The AKL allows exploitation of independent and-parallelism. It is sufficient to isolate the independent and-parallel guards as guards of newly created goals. Obviously, such scheme depends on an accurate definition of stability, that is stability between and-boxes corresponds to each and-goal not interfering (in other words, they should be *independent*), and the implementation must provide the machinery to calculate the cross-product of solutions (in terms of AKL terminology, run-time machinery must now support parallel nondeterminate promotions).

Janson and Montelius [84] discuss how AKL subsumes the language GHC (although GHC is in practice more similar to the Extended Andorra Model, as goals in the guard cannot even make local bindings to external variables) and can support Prolog. There is already some work in parallel versions of AKL, one by Palmer supporting and-parallelism [123] and another by Van Acker and others supporting AND/OR parallelism [168]. In both cases, the authors' initial claim is that the parallel implementation of AKL will not be too complex.

11.5 Summary

This chapter described three approaches to extending the Basic Andorra Model, IDIOM, the EAM, and AKL. IDIOM aims at adding conventional independent and-parallelism to the Basic Andorra Model. Also, the design of IDIOM has tried to reuse previous experience on parallel systems as much as possible.

The Extended Andorra Model aims at extracting the maximum parallelism, whilst doing the best possible search. The model specifies rewrite rules on and-or boxes. A new technique, lazy copying, is also introduced to maximise parallelism. AKL is a programming language based on ideas similar to the Extended Andorra Model, but where control is more explicit.

Chapter 12

Conclusions and Future Work

The Andorra-I system was designed for the parallel execution of logic programs, and particularly of Prolog programs. Andorra-I obtains and-parallelism by running determinate goals in parallel, and or-parallelism by trying the alternatives to non-determinate goals in parallel. Experience in using Andorra-I has led to the following main conclusions:

- Andorra-I performs well and exploits parallelism successfully for Prolog applications, committed-choice style applications, and new, "Andorra style", applications.
- The coroutining in the Basic Andorra Model can reduce the search space of logic programs. This holds true for some Prolog programs, and has been exploited in the "Andorra style" applications.

These good results depend to a large extent on the operation of the preprocessor:

- The determinacy property of goals can be detected efficiently at run-time by using special determinacy code.
- Compile-time analysis allows the execution of Prolog programs in Andorra-I.

This chapter analyses the main contributions of this work, and discusses two fundamental issues, programming languages for Andorra-I, and directions on how to extract more parallelism from logic programs.

12.1 The Preprocessor

In designing Andorra-I, we believed that it would be more useful to have a system that could support existing logic programming languages, instead of trying to relying on a new, alternative, logic programming language, to support parallel execution in the Basic Andorra Model. We concentrated on Prolog as it is the most popular logic programming language.

The main contribution of this work is the idea of using a *preprocessor* to convert from a logic programming language, such as Prolog, to a logic programming system supporting a different execution mechanism from the one for which Prolog was originally designed. Although the preprocessor was designed for Prolog, other preprocessors could be designed to support different programming languages such as the committed-choice languages or coroutining languages.

We have found that the early execution of determinate goals can interfere with the correct execution of Prolog builtins in an Andorra-I environment. Examples include some uses of cut, the meta-predicate `var/1`, and the side-effects predicates. The preprocessor thus includes a *sequencer* that prevents early execution of goals that can interfere with these builtins.

Results show that although the actual sequencing varies from application to application, so far it does not seem to be a major constraint on parallelism, as it usually only affects a few parts of the program (the exception being programs that heavily change the database and programs that heavily use sensitive meta-predicates).

The sequencer needs precise information on how the cut and some meta-predicates are to be used. We designed an *abstract interpreter* to collect mode patterns that can in turn be used to study the operation of the program. Besides simple groundness or freeness information, the abstract interpreter generates information on the structure of arguments, that is if they are lists, atoms or compound terms.

The current abstract interpreter performs well (and sometimes very well) for quite a few applications, but it can also fail to recognise modes in some applications. Although abstract interpretation is limited by the fact that it must generate correct results for all possible runs of the program, experience shows that most of the problems with the current system result from limitations in the abstract domain and in program representation. Despite this, the abstract interpreter is one of the most powerful currently available, and there is a question of how far can one go in improving the analysis and still obtaining acceptable preprocessing times.

The preprocessor generates the code that is called at run-time to detect when a goal is determinate. This code recognises when a goal is determinate either because a single (or no) clause matches, or due to the operation of a cut or commit. The code was designed to be efficient, and to minimise any overheads Andorra-I would introduce when testing determinacy of goals. Thus some simplifications were introduced in the design of the analyser.

The Andorra-I performance shows that, in practice, the code recognises a very high percentage of all determinate goals. Moreover, the time spent in the determinacy code seems to be only a small percentage of the total execution time.

A compiler-based version of Andorra-I was necessary for a performance comparable to current Prolog systems. A WAM-like abstract instruction set was designed for this purpose, and the preprocessor includes a clause compiler. Although the compiler would benefit from further optimisations, it is now sufficiently robust to process a large number of applications.

12.2 Language Issues

The preprocessor provides independence between the engine and the languages used to program it. The preprocessor has allowed programs originally written for Prolog to be run in Andorra-I.

The main difference between Prolog execution and Andorra-I execution is the coroutining. Is this coroutining important? There is quite a large set of committed-choice programs that demand it. Moreover, programs designed for Andorra-I show that the coroutining can substantially reduce the search-space for many applications. These applications usually also have substantial or- and and-parallelism. In chapter 10 we looked at several "Andorra style" programs, either initially developed for Andorra-I, or developed for similar languages such as Pandora and AKL. The applications given also exemplify how constraint-like programming can be implemented in Andorra-I, by associating constraints to determinate goals. Note that some Prolog programs can also take advantage of the coroutining.

A problem with Prolog is that Prolog programs do not make explicit whether cuts are quiet, or whether builtins are sensitive. This information is needed for Andorra-I, hence the need for sequencing. In the worst cases user intervention may be convenient, especially if the abstract interpreter failed to recognise all the modes.

Our experience with executing Prolog programs in Andorra-I allowed us to reach some conclusions on characteristics of Prolog programs that can perform well in parallel environments:

- There should be a clear separation between what is the logic of the program, and what is the control.
- Pruning should in general be quiet. Extra restrictions (such as the silent commit) can be allowed, but it is very important that pruning will be quiet.
- Finally, a program may in certain circumstances need to perform side-effects or noisy cuts. Their scope should be clear from the context and their effects easy to understand.

The fundamental idea is *locality of control*: ideally, the meaning of a Prolog construct should be clear just from looking at the procedure where it is placed. We believe that such ideas are valid for any logic programming language.

12.3 Areas of Further Research

Throughout, this thesis has mentioned how the individual components of the preprocessor can be improved. Better results from the abstract interpreter should be quite useful, as they would further reduce the cases where human intervention is necessary. Improvements in the compiler (and also in the determinacy analyser) should result in an improved Andorra-I base performance.

Still, the main limitation of Andorra-I is that coroutining and and-parallelism can only be exploited between determinate goals. There are several possible solutions to this problem.

First, one can broaden the definition of "determinate goal" in Andorra-I, to recognise the goals where one clause becomes the solution, either because some Prolog goals fail and exclude the other clauses, or because some Prolog goals succeed and result in pruning. Some restrictions would be needed in order to preserve efficiency. One simple such scheme would be to extend the determinacy analyser to analyse simple goals which would not bind external variables. Note that a different approach (although in the same vein) would be to transform the Prolog program so that more goals would be determinate. It would be a worthy subject of research to find out how much Prolog programs can benefit from these schemes.

The IDIOM model (described in section 11.2) addresses a wider range of and-parallelism. This model combines dependent and-parallelism between determinate goals with independent and-parallelism. As independent and-parallelism between determinate goals is automatically exploited by Andorra-I, the main benefit of IDIOM is exploiting any independent and-parallelism between nondeterminate goals. This form of parallelism does seem to appear widely in Prolog programs.

Although the two previous solutions expand the amount of and-parallelism that Andorra-I can exploit, they only partially address the problem of how to get the most and-parallelism out of a logic program. The Extended Andorra Model (see 11.3) provides a more thorough solution to this problem. This model gives dependent and-parallelism between non-determinate goals and subsumes independent and-parallelism. The drawbacks will be a more complex, and possibly more expensive, implementation. Similar principles are followed by the AKL, for which an early sequential implementation is available. It will be worthwhile to design and implement parallel logic programming systems based on the Extended Andorra Model and of the AKL, and compare them to current systems as Andorra-I.

Early systems such as Aurora and JAM proved that parallel logic programming can be useful and practical, but followed very different paths to do so. Andorra-I shows that one can obtain the best of these two different worlds. The Extended Andorra Model and other models show that parallel logic programming is taking steps to remove any restrictions on what parallelism can be exploited, towards the ultimate goal of maximum parallelism with minimal user intervention.

Appendix A

Basic Operations on the Abstract Domain

In this appendix we present the implementation of the fundamental operations for the abstract domain used in the abstract interpreter. We first present the fundamental implementation issue of how to represent the abstract execution of a clause, and then present the several operations on the abstract domain.

A.1 Environments

In Bruynooghe's framework, execution of a clause is represented by a set of abstract substitutions $\beta_1, \ldots, \beta_n$ corresponding to entry and exit of every subgoal in the clause. Considering the set $\mathcal{V}$ of all program variables appearing in the clause, and a goal Q constructed by having each variable in $\mathcal{V}$ appear as an argument to Q, $\beta_1, \ldots, \beta_n$ can be seen as applying abstract substitutions $\Theta_1, \ldots, \Theta_n$ to the variables in Q.

In our system, for each clause we use an environment $\mathcal{E}$ where the substitutions Θ are explicitly represented by pairs $Label/AbstractTerm$. To obtain an entry substitution β_i we define a function $project(Goal, \mathcal{E})$. Abstract unification between a subgoal $Goal$ and its exit substitution β_j is used to create the corresponding exit environment $\mathcal{E}'$ for the subgoal.

Environments $\mathcal{E}$ are implemented as dictionaries of pairs $L/\overline{T}$, where L is a label and $\overline{T}$ an abstract term. There is an unique value for each label. Dictionaries are updated by abstract unification, and avoid the problems corresponding to the direct use of logic variables [187].

Currently, the preprocessor implements environments as lists of pairs of the form variable to assignment. Labels are simply implemented as Prolog variables, as it is easier in Prolog to create a new variable than a new constant. Moreover, if ever two abstract terms are made to share the same label, one can rely on straightforward unification. Environments are updated by pushing the new substitution to the top of list. An alternative to the use of lists would be binary trees. The main advantage of binary trees would be to provide logarithmic time access to a pair (instead of linear). This should provide faster access, particularly because environments can grow to be quite large.

A.2 Procedure Entry

The process of procedure entry in our system consists of four steps: (a) project the variables from the caller environment $\mathcal{E}$ to obtain β_i; (b) verify if the call should be made; (c) create an or-node with branches for each clause; (d) for each clause do abstract unification with the head and if successful create a corresponding environment $\mathcal{E}'$. We discuss each step in detail.

Step (a) is implemented by the function $project(Goal, \mathcal{E})$. For every variable appearing in *Goal* this function looks up its corresponding abstract substitution from $\mathcal{E}$. Dereferencing must be made for a variable, in case there is a chain of labels, and for all the arguments of a compound term. The system ensures that all the labels (Prolog variables) appearing in the implementation of β_i do not appear in the implementation of $\mathcal{E}$.

Step (b) depends on the iteration algorithm and is discussed in detail later.

Step (c) is implemented by finding all clauses for the corresponding procedure, and executing them one by one. The algorithm tries each clause sequentially.

The most complex step is the implementation of abstract unification.

A.2.1 Abstract Unification

Given two terms $\overline{A}$ and $\overline{B}$, corresponding to the two sets of concrete terms $\gamma(\overline{A})$ and $\gamma(\overline{B})$, abstract unification tries to find $\overline{X} = \alpha(mgu(\gamma(A), \gamma(B)))$, or at least a correct approximation $\overline{X} \sqsubseteq \overline{Y}$.

Our system represents abstract substitutions as a set of substitutions in an environment $\mathcal{E}$. Therefore, successfully unifying two abstract terms A and B results in updating the environment $\mathcal{E}_0$ with the necessary substitutions. The new, updated environment, will be called $\mathcal{E}_f$. The algorithm for abstract unification thus corresponds to the following Prolog procedure:

```
abstract_unify(A, B, E0, EF) :-
        deref(A, E0, T1),
        deref(B, E0, T2),
        abstract_unify_derefd(T1, T2, E0, EF).
```

`deref` is used by the implementation to obtain the current value of a term from the current environment. `abstract_unify_derefd` implements the real unification of two abstract terms. This procedure receives two terms and the original environment, and crates the new environment $\mathcal{E}_f$.

`abstract_unify_derefd` can be described in terms of a set of rules that are given two abstract terms and that update a binding environment. The next tables show the rules for abstract unification between a term of the type in the first column to a term A in the presence of the environment $\mathcal{E}$. The rules return the new abstract bindings that replace previous abstract bindings in the environment.

The simplest case corresponds to unifying a constant with an abstract term. We show this case in table A.1. The table consists of five columns. The first column give the values for the term B in the environment $\mathcal{E}$ we are unifying A with, and the second column gives what conditions are tested by abstract unification. The third and fourth column describe the result, in terms of the value that both A and B will take after abstract unification and of any extra constraints on the environment $\mathcal{E}$.

Clearly, the rules for abstract unification with a defined constant mimic unification closely. Either the term has a main functor, and it is verified if it can unify, or it is bound to the constant. If a variable $var(L)$ is bound to a constant, all variables sharing the same label must take the same value:

$$p(T, L) \rightarrow \{\forall (v'/Var(L)) \in \mathcal{E} : v'/T\}$$

$p(T, L)$ enforces this constraint. A similar operation is not needed if one unifies the constant with a term of the form $\top(L')$, because our domain cannot guarantee that two terms $\top$ sharing the same labels will also always share every variable.

The case for $Constant$, the set of all constants is similar, but now we cannot assume a particular value for the constant. The rules are shown in Table A.2.

$OtherArg.$	$Test$	$Result$	$ExtraConstraints$
$atom'$	$atom' = atom$	$atom$	
	$atom' \neq atom$	$\perp$	
$Constant$		$atom$	
$f(\ldots)$		$\perp$	
$Or(\mathcal{B})$	$Constant \in \mathcal{B}$	$atom$	
	$Constant \notin \mathcal{B}$	$\perp$	
$List(\ldots)$	$atom = []$	$[]$	
	$atom \neq []$	$\perp$	
$Var(L)$		$atom$	$p(atom, L)$

Table A.1: Abstract Unification with `atom`

$OtherArg.$	$Test$	$Result$	$ExtraConstraints$
$atom'$		$atom'$	
$Constant$		$Constant$	
$f(\ldots)$		$\perp$	
$Or(\mathcal{B})$	$Constant \in \mathcal{B}$	$Constant$	
	$Constant \notin \mathcal{B}$	$\perp$	
$List(\ldots)$		$[]$	
$Var(L)$		$Constant$	$p(Constant, L)$

Table A.2: Abstract Unification with any $Constant$

The rules for abstract unification of compound terms are based on the principle that terms that can have the same functor will match, and then abstract unification proceeds for all the arguments of the term. The simplest rules correspond to abstract terms of the form $f(\ldots, a_i, \ldots)$, where the main functor is known. They are shown in table A.3.

The two constraints a and u should be explained. Two compound terms can only unify if their arguments unify. A guarantees that all the arguments unify. The rule u corresponds to the case where we want to unify a pair of the form $[a_1|a_2]$ with a list $L(b)$. The list is the only recursive abstract term. To unify it with a normal term, we need first to unfold the list into a term of the form $[b_1|List(b)]$, where b_1 is constrained to be an instance of the argument of the list.

The rules for abstract unification with the abstract term "set of compound terms" are presented in table A.4.

$OtherArg.$	$Test$	$Result$	$ExtraConstraints$
$f(\ldots,b_i,\ldots)$ $g(\ldots,\ldots)$		$f(\ldots,c_i,\ldots)$ $\perp$	$\forall i, \{c_i : c_i = a(a_i,b_i)\}$
$Or(\mathcal{B})$	$f(\ldots,b_i,\ldots\}) \in \mathcal{B}$ $f(\ldots,b_i,\ldots\}) \notin \mathcal{B}$	$f(\ldots,c_i,\ldots)$ $\perp$	$\{c_i : c_i = a(a_i,b_i)\}$
$List(b)$	$f(\ldots) = [a_1\|a_2]$ $f(\ldots) \neq [\_\|\_]$	$[c_1\|c_2]$ $\perp$	$c_1 = a(a_1,U(b)) \wedge c_2 = a(a_2,List(b))$
$Var(L)$		$f(\ldots,a_i,\ldots)$	$p(f(\ldots,a_i,\ldots),L)$

Table A.3: Abstract Unification with $f(\ldots,a_i,\ldots)$

$OtherArg.$	$Test$	$Result$	$ExtraConstraints$
$Or(\mathcal{B})$		$Or(\mathcal{C})$	$\mathcal{C} = \{f(\ldots,a_i,\ldots):$ $f(\ldots,a_i,\ldots) \in \mathcal{A}$ $f(\ldots,b_i,\ldots) \in \mathcal{B},$ $\{c_i : c_i = a(a_i,b_i)\}\}$
$List(b)$	$Constant \notin \mathcal{A} \wedge [a_1\|a_2] \notin \mathcal{A}$ $Constant \in \mathcal{A} \wedge [a_1\|a_2] \notin \mathcal{A}$ $Constant \notin \mathcal{A} \wedge [a_1\|a_2] \in \mathcal{A}$ $Constant \in \mathcal{A} \wedge [a_1\|a_2] \in \mathcal{A}$	$\perp$ $[]$ $[c_1\|c_2]$ $Or(\{Constant,$ $[c_1\|c_2]\}$	 $c_1 = a(a_1,U(b))\wedge$ $c_2 = a(a_2,List(b))$ $c_1 = a(a_1,U(b))\wedge$ $c_2 = a(a_2,List(b))$
$Var(L)$		$Or(\mathcal{A})$	$p(Or(\mathcal{A}),L)$

Table A.4: Abstract Unification with a set of compound terms $Or(\mathcal{A})$

To unify two Or terms one needs to find the terms whose main functor appears in both Or terms and then construct a new $Or(\mathcal{C})$ term. This term can be simplified if $\mathcal{C}$ consist of a single element (to the element itself) or if $\mathcal{C}$ is empty. The simplification rules are made clear in the case of unifying Or with a $List$. The result can consist either of a list constructor or of []. The table details the necessary operations, which generalise on unifying a list with a term of the form [_|_].

We now consider unification with lists, as presented in table A.5.

Unification of two lists simply corresponds to unifying their arguments.

Table A.6 shows abstract unification to a term of the form $Ground$. Basically, we need to

$OtherArg.$	$Test$	$Result$	$ExtraConstraints$
$List(b)$		$List(c)$	$c = a(a, b)$
$Var(L)$		$List(a)$	$p(List(a), L)$

Table A.5: Abstract Unification with a list $L(a)$

preserve the structure of the term B, plus the fact that all possible variables in the term have now become fully instantiated. For instance, if a sub-term is of the form $V(L)$ this corresponds to the fact that all variables in the term will be instantiated to ground. The rules are shown in table A.6.

$OtherArg.$	$Test$	$Result$	$ExtraConstraints$
$atom$		$atom$	
$Constant$		$Constant$	
$f(\ldots, b_i, \ldots)$		$f(\ldots, c_i, \ldots)$	$\forall i, \{c_i : c_i = a(Ground, b_i)\}$
$Or(\mathcal{B})$		$Or(\mathcal{C})$	$\{c_i \in \mathcal{C} : c_i = a(Ground, b_i \in \mathcal{B})\}$
$List(b)$		$List(c)$	$c = a(Ground, b)$
$Ground$		$Ground$	
$Var(L)$		$Ground$	$p(Ground, L)$

Table A.6: Abstract Unification with $Ground$

Table A.7 shows abstract unification to $\top(L')$. Basically, all noninstantiated arguments in the other term are promoted to $\top(L')$ and made to share with the same values. An important case is when we abstract unify two $\top$ terms. As a result of the abstract unification, the two terms will now share, and the two top terms must have the same label. This is guaranteed by the constraint $j(\top(L'), \top(L''))$.

In general, most abstract unification rules constrain the abstract terms and thus provide more precise information. The notable exception is unification with a term of the form $\top$: unification of abstract terms that can include possibly unbound variables with $\top$ actually increases the sets of elements represented by the term $\top$ (because we do not know if the variables may or not have been bound, and to which values).

$OtherArg.$	$Test$	$Result$	$ExtraConstraints$
$atom$		$atom$	
$Constant$		$Constant$	
$f(\ldots, b_i, \ldots)$		$f(\ldots, c_i, \ldots)$	$\forall i, \{c_i : c_i = a(\top(L'), b_i)\}$
$Or(\mathcal{B})$		$Or(\mathcal{C})$	$\{c_i \in \mathcal{C} : c_i = a(\top(L'), b_i \in \mathcal{B})\}$
$List(b)$		$List(c)$	$c = a(\top(L'), b)$
$Ground$		$Ground$	
$Var(L)$		$top(L')$	$P(\top(L'), L)$
$\top(L'')$		$top(L')$	$j(\top(L'), \top(L''))$

Table A.7: Abstract Unification with $\top(L')$

A.3 Procedure Exit

Procedure exit corresponds to decorating the right side of a node with an abstract substitution. Within the chosen representation of the abstract And/Or-tree, this abstract substitution is calculated by obtaining the least upper bound of the abstract substitutions for each clause. These are themselves obtained by projecting the exit abstract substitution of the last goal to the head variables of the clause.

The new operation is the calculation of an upper bound. For simplicity of the implementation this is usually a worse approximation than the correct least upper bound.

A.3.1 Calculating the Least Upper Bound ($\sqcup$)

The algorithm proceeds in two steps. First, it calculates a first structure of the least upper bound. Second, if verifies if the sharing conditions for any abstract terms of the form $Var(L)$ hold (i.e., it verifies it two terms $Var(L)$ with the same label are guaranteed to be the same variable). If they are not, the terms must be promoted, in our case to $\top(L')$.

Most of the rules of formation of structure are straightforward. Some of the most interesting are shown in table A.8 (these rules presume that the labels of both arguments are named apart).

The rule for the bottom element is trivial: the least upper bound of a term with bottom is the term itself. The two rules for Var say that if the other term is a Var the least upper bound is, at least a Var, otherwise it must be the $\top$ element. The lub of anything with

$\overline{T}_1$	$\overline{T}_2$	$Test$	$lub(\overline{T}_1, \overline{T}_2)$
$\perp$	$\overline{T}_2$		$\overline{T}_2$
$Var(L)$	$Var(L')$		$Var(L'')$
$Var(L)$	$\overline{T}_2$	$\overline{T}_2 \neq Var(L')$	$\top(L'')$
$\top(L)$	$\overline{T}_2$		$\top(L'')$
$Ground$	$\overline{T}_2$	$vars(\gamma(\overline{T}_2)) = \emptyset$	$Ground$
$Ground$	$\overline{T}_2$	$vars(\gamma(\overline{T}_2)) \neq \emptyset$	$\top(L'')$
$[]$	$[]$		$[]$
$[]$	$[b_1, \ldots, b_n]$		$List(b_1 \sqcup \ldots \sqcup b_n)$
$[]$	$List(b)$		$List(b)$
$atom_1$	$atom_2$	$atom_1 = atom_2$	$atom_1$
$atom_1$	$atom_2$	$atom_1 \neq atom_2$	$Constant$
$atom_1$	$f(\ldots)$		$Or(\{f(\ldots), Constant\})$
$atom_1$	$Or(\mathcal{B})$	$Constant \in \mathcal{C}$	$Or(\mathcal{C})$
$atom_1$	$Or(\mathcal{B})$	$Constant \notin \mathcal{C}$	$Or(\mathcal{C} \cup Constant)$
$atom_1$	$List(b)$		$Or(\{[b'\|b], Constant\})$
$f(\ldots, a_i, \ldots)$	$f(\ldots, b_i, \ldots)$		$f(\ldots, a_i \sqcup b_i, \ldots)$
$f(\ldots)$	$g(\ldots)$		$Or(\{f(\ldots), g(\ldots)\})$
$f(\ldots)$	$Or(\mathcal{B})$	$f(\ldots) \notin \mathcal{B}$	$Or(\mathcal{C} \cup f(\ldots))$
$f(\ldots, a_i, \ldots)$	$Or(\mathcal{B})$	$f(\ldots, b_i, \ldots) \in \mathcal{B}$	$Or(\mathcal{B} \backslash \{f(\ldots)\} \cup f(\ldots, a_i \sqcup b_i, \ldots))$
$[a_1, \ldots, a_n]$	$List(b)$		$List(a_1 \sqcup \ldots \sqcup a_n \sqcup b)$
$[a_1\|a_2]$	$List(b)$		$Or(\{[b' \sqcup a_1 \| b \sqcup a_2], Constant\}$
$List(a)$	$f(\ldots)$	$f(\ldots) \neq [b_1\|b_2]$	$Or(\{f(\ldots), [a'\|a], Constant\})$
$List(a)$	$List(b)$		$List(a \sqcup b)$

Table A.8: Examples of rules to calculate the least upper bound

$\top$ is clearly $\top$. The lub of an abstract term $\overline{T}$ with *Ground* is only *Ground* if $\overline{T}$ describes ground terms, otherwise it must be $\top$. A special case corresponds to the atom `[]`. When combined with a list constructor, `[]` is used to generate the abstract type *List*. The rules are simpler for other constants: if combined with a different constant they generate *Constant*; if combined with a compound term they generate *Or*; if combined with an *Or* they imply that *Constant* must be contained in the arguments to the resulting *Or*; if combined with *List* they result in an *Or*. Rules for the *Constant* abstract type are very similar.

Concerning the rules for compound terms. The least upper bound of two compound terms with the same functor has the same functor and the arguments are combined recursively. If the compound terms have different functors, an *Or* term is built. When combining a compound term with an *Or*, the algorithm verifies if a term with the same functor exists below the *Or*. If so, the arguments of these two terms are combined. Otherwise, the term is added to the Or. When combining a compound term with a list, particular care is given to terms of the form $[a_1|a_2]$. If they are true lists, the least upper bound is also a true list. If they are not true lists (e.g., open lists) an Or is created. An Or is also created when considering the general case of least upper bound of a compound term and a list.

The least upper bound of two terms *List* is a *List* whose argument is the abstract upper bound of both lists. The calculation of upper bounds between a *List* and a term of the form Or, or between terms of the form Or are not described here, but can be obtained as a simple generalisation of the previous rules.

These rules are used recursively up to a certain depth. When this limit is reached, much simpler rules is used: the result of the least abstract bound operation can only be a constant, *Ground*, Var, or $\top$, but can never be a compound term, a *List* or an Or. This corresponds to the depth limit on abstract terms, which guarantees finiteness of the domain domain, and hence termination.

Sharing The previous rules are not complete. They do not describe how to calculate the new labels for the terms of the form $Var(L'')$ or $\top(L'')$.

To compute labels for $\top$ one has simply to guarantee that all labels in abstract terms that have been updated to the same $\top$ term, share a single label.

To compute labels for Var one has two problems. The first one concerns the rule $lub(Var(L), \overline{T}_2) = \top(L'')$. If this rule is applied to one argument of the form $Var(L)$, then, in all other places where a term of the form $Var(L)$ appears, the least upper bound must *always* be $\top(L'')$.

Problems may also arise with the rule $lub(Var(L), Var(L')) = Var(L'')$. The reason is that we *must* guarantee that if one subterm of the least upper bound if of the form $Var(L'')$, whenever $Var(L'')$ is bound to some value, all other subterms of the form $Var(L'')$ should be bound to the same value. This situation is particularly subtle when constructing recursive terms, in this case the *List*.

Table A.9 shows some examples of this problem (V is used instead of Var and i is used instead of L_i for readability).

The first two columns show the arguments to lub. The third column shows the *correct* least upper bound. In the first example it is necessary to promote both arguments because binding one argument will not necessarily result in binding the other (therefore an abstract term of the form Var cannot be used, although we know the least upper bound is a variable).

$\overline{T}_1$	$\overline{T}_2$	$lub(\overline{T_1}, \overline{T_2})$
$a(V(1), V(2))$	$a(V(1'), V(1'))$	$a(\top(1''), \top(1''))$
$a(a(V(1)), V(2))$	$a(b(V(1')), V(1'))$	$a(Or(\{a(V(1''), b(\top(2''))\}, \top(2''))$
$a([V(1)], [V(1)])$	$a([], [])$	$a(List(\top(1'')), List(\top(1'')))$

Table A.9: Examples of promotion to $\top$ due to sharing

The second example demonstrates a more subtle problem, where a variable appears both inside and outside an Or term. In this case, binding the occurrence outside the Or should result in binding the inner occurrence. Unfortunately, the opposite is not possible and we have to promote the variable to $\top$.

Finally, the last example concerns occurrences of variables in lists. Variables are allowed inside lists, but in this case their meaning is not a set of variables that always share, but a set of sets of variables that always share, one for each element of the list. Because the use of *rename* may force the creation of new variables during abstract unification, it is not allowed for a Var term appearing in a list to have a scope larger than the $List$. The alternative would be during abstract unification and projection to unfold both lists, and this cannot be guaranteed to be correct in our domain.

These transformation rules lose some information. A simple solution would be to extend the abstract domain with a new type to represent variables which might or not share. A disadvantage of this new abstract type is that unification of one element could result in promoting the other elements sharing the same label to $\top$.

A.4 Interpretation of Builtins

Interpretation of builtins consists of (a) verifying that the arguments to the builtin are correct and (b) simulating the execution of the builtin.

The first operation usually consists of a type check applied to the arguments of the builtin. In some cases, say for the builtin `is`, if it is found that the second argument is Var, the interpreter generates an error. In other cases, say for the builtin `atom`, if the argument is Var, the builtin is simply made to return the empty substitution, i.e., to fail.

Simulating the execution of a builtin can be more complex. In general, the operation of a builtin may result in (i) changing the instantiation of its arguments, (ii) providing extra control to the program, (iii) changing the external environment to the program, (iv) changing the program database, or (v) generating a new computation for some predicate P.

Consequence (i) can be simulated by performing abstract unification with a skeleton representing the consequences of the program. For instance, the builtin `=..`, can be simulated by performing abstract unification of the input arguments with the term $\top(L) = ..[Cons|List(\top(L))]$. In some cases, more precise analysis is possible. For `=..`, if the first argument is $f(\ldots)$ then the builtin itself could be executed. In practice, separate rules are generated for each builtin, and the amount of effort devoted to each builtin depends on its importance in real programs.

Consequence (ii) concerns the pruning operators. Unfortunately, the system is in general not precise enough to know when pruning is performed, and in practice ignores pruning. Notice that this does not affect the correctness of the system, only that more solutions are considered.

Consequence (iii) results from builtins such as `write` or `read`. External actions are of no interest to the abstract interpreter, which is only interested in the internal operation of the program.

Consequence (iv) in Prolog relates to the use of `assert`, `retract` or similar predicates. Debray gives a detailed discussion of the problem [43]. Basically, he defines predicates as *static* if they are not affected by asserts or retracts, and tries to determine which predicates can be asserted or retracted (this is trivial if the Prolog system forces `dynamic` declarations). The most important conclusion concerns *stable programs*, i.e., programs where the use of assert does not create new uses of assert, and which can be analysed. An important case of stable programs are programs where only facts are asserted. Our system tries to verify this condition, if it it not guaranteed to hold it generates warnings but continues analysis regardless.

Finally, consequence (v) concerns mainly the use of `call`. In this case the abstract types $f(\ldots)$ and Or are very useful in reducing most metacalls to normal calls. Otherwise, the system again generates warnings, as abstract interpretation may not provide a correct approximation to the program.

Bibliography

[1] H. Aït-Kaci. *Warren's Abstract Machine — A Tutorial Reconstruction.* MIT Press, 1991.

[2] K. A. M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *Proceedings of the North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[3] R. Bahgat. *Non-Deterministic Concurrent Logic Programming in Pandora.* World Scientific, 1993.

[4] R. Bahgat and S. Gregory. Pandora: Non-deterministic Parallel Logic Programming. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 471–486. MIT Press, June 1989.

[5] M. G. d. l. Banda and M. V. Hermenegildo. A Practical Approach to the Global Analysis of CLP Programs. In *ILPS93*, pages 437–455, 1993.

[6] A. K. Bansal and L. Sterling. An Abstract Interpretation Scheme for Identifying Inherent Parallelism in Logic Programs. *New Generation Computing*, 7(2,3):273–324, 1990.

[7] G. Battani and H. Meloni. Interpréteur de Langage de Programmation Prolog. Internal report, Groupe Intelligence Artificielle, Université Aix-Marseille II, September 1973.

[8] A. Beaumont, S. M. Raman, P. Szeredi, and D. H. D. Warren. Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler. In *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 403–420. Springer Verlag, June 1991.

[9] G. Bell. Ultracomputers: a Teraflop Before its Time. *Communications of the ACM*, 35(8):26–47, 1992.

[10] P. Boizumault. A General Model to Implement Dif and Freeze. In E. Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 585–592. Springer-Verlag, 1986.

[11] K. A. Bowen, K. A. Buettner, I. Cicekli, and A. K. Turk. The Design of a High-Speed Incremental Portable Prolog Compiler. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 650–656. Imperial College, Springer-Verlag, July 1986.

[12] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *The Journal of Logic Programming*, 10(2), February 1991.

[13] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference (Extended abstract). In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 669–683, August 1988.

[14] M. Bruynooghe, G. Janssens, A. Callebault, and B. Demoen. Abstract Interpretation: Towards the Global Optimisation of Prolog Programs. In *Proceedings 1987 Symposium on Logic Programming*, pages 192–204. IEEE Computer Society, September 1987.

[15] A. Calderwood and P. Szeredi. Scheduling or-parallelism in Aurora – the Manchester scheduler. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 419–435. MIT Press, June 1989.

[16] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 40–58. University of Melbourne, "MIT Press", May 1987.

[17] M. Carlsson. On the efficiency of optimised shallow backtracking in Compiled Prolog. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 3–15. MIT Press, June 1989.

[18] M. Carlsson and P. Szeredi. The Aurora abstract machine and its emulator. SICS Research Report R90005, Swedish Institute of Computer Science, 1990.

[19] M. Carlsson and J. Widen. SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science, 1988. SICS Research Report R88007B.

[20] J. Chassin de Kergommeaux. Measures of the PEPSys Implementation on the MX500. Technical Report CA-44, ECRC, January 1989.

[21] J. Chassin de Kergommeaux and P. Robert. An Abstract Machine to Implement Or-And Parallel Prolog Efficiently. *The Journal of Logic Programming*, 8(3), May 1990.

[22] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 276–293. University of Melbourne, "MIT Press", May 1987.

[23] K. L. Clark and S. Gregory. A Relational Language for Parallel Programming. In Arvind, editor, *ACM Conference on Functional Programming Languages and Computer Architecture*, pages 171–178. Acm, October 1981.

[24] K. L. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM TOPLAS*, 8:1–49, January 1986.

[25] K. L. Clark and S. Gregory. PARLOG and Prolog United. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 927–961, May 1987.

[26] K. L. Clark, F. G. McCabe, and S. Gregory. IC-PROLOG – language features. In K. L. Clark and S. A. Tärnlund, editors, *Logic Programming*, pages 253–266. Academic Press, London, 1982.

[27] W. F. Clocksin. Principles of the DelPhi parallel inference machine. *Computer Journal*, 30(5):386–392, 1987.

[28] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1986.

[29] M. Codish, M. Falaschi, and K. Marriott. Suspension analysis for concurrent logic programs. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 331–345, Cambridge, Massachusetts London, England, 1991. MIT Press.

[30] M. Codish and E. Shapiro. Compiling OR-parallelism into AND-parallelism. In E. Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 283–297. Springer-Verlag, 1986.

[31] C. Codognet and P. Codognet. Non-deterministic Stream And-Parallelism Based on Intelligent Backtracking. In G. Levi and M. Martelli, editors, *Logic Programming: Proceedings of the Sixth International Conference*, pages 63–79. The MIT Press, 1989.

[32] C. Codognet, P. Codognet, and M.-M. Corsini. Abstract Interpretation for Concurrent Logic Languages. In *Logic Programming Proceedings of the 1990 North American Conference*, pages 215–232. MIT Press, October 1990.

[33] A. Colmerauer. Theoretical Model of Prolog II. In M. van Caneghen and D. H. D. Warren, editors, *Logic Programming and its Applications*, pages 3–31. Ablex Publishing Corporation, 1986.

[34] A. Colmerauer. An Introduction to Prolog-III. *Communications of the ACM*, 33(7):69–90, July 1990.

[35] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un système de communication homme–machine en francais. Technical report cri 72-18, Groupe Intelligence Artificielle, Université Aix-Marseille II, October 1973.

[36] J. S. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Norwell, Ma 02061, 1987.

[37] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[38] P. Cousot and R. Cousot. Comparison of the Galois Connexion and Widening/Narrowing Approaches to Abstract Interpretation. In *ICLP'91 Pre-Conference Workshop on Semantics-Based Analysis of Logic Programs*, Università di Pisa, Dipartimento di Informatica, June 1991.

[39] J. A. Crammond. A Garbage Collection Algorithm for Shared Memory Parallel Processors. *International Journal of Parallel Processing*, 17(6), December 1988.

[40] J. A. Crammond. *Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors.* PhD thesis, Heriot-Watt University, Edinburgh, May 1988. Research Report PAR 88/4, Dept. of Computing, Imperial College, London.

[41] J. A. Crammond. The Abstract Machine and Implementation of Parallel Prolog. Technical report, Dept. of Computing, Imperial College, London, June 1990.

[42] J. A. Crammond. Scheduling and Variable Assignment in the Parallel Parlog Implementation. In *1990 North American Conference on Logic Programming*, pages 642–657. MIT Press, October 1990.

[43] S. K. Debray. A Simple Code Improvement Scheme for Prolog. In *Sixth International Conference on Logic Programming*, pages 17–32. MIT Press, June 1989.

[44] S. K. Debray. Flow Analysis of Dynamic Logic Programs. *The Journal of Logic Programming*, 7(2):149–176, September 1989.

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

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

[47] D. DeGroot. Restricted and-parallelism. In H. Aiso, editor, *International Conference on Fifth Generation Computer Systems 1984*, pages 471–478. Institute for New Generation Computing, Tokyo, 1984.

[48] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *International Conference on Fifth Generation Computer Systems 1988*, pages 693–702. ICOT, Tokyo, Japan, Nov. 1988.

[49] V. Dumortier, G. Janssens, M. Bruynooghe, and M. Codish. Freeness analysis in the presence of numerical constraints. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 100–115, Budapest, Hungary, 1993. The MIT Press.

[50] I. C. Dutra. A Flexible Scheduler for the Andorra-I System. In *LNCS 569, ICLP'91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, pages 70–82. Springer-Verlag, June 1991.

[51] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A New Declarative Semantics for Logic Languages. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 993–1006. MIT Press, August 1988.

[52] T. Franzén. Logical Aspects of AKL. Sics research report r91:12, Swedish Institute of Computer Science, October 1991.

[53] J. Gabriel, T. Lindholm, E. L. Lusk, and R. A. Overbeek. A Tutorial on the Warren Abstract Machine for Computational Logic. Research Paper ANL-84-84, Argonne National Laboratory, June 1985.

[54] R. P. Gabriel. *Performance and evaluation of Lisp systems.* MIT Press, 1985.

[55] J. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialization. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 732–746. MIT Press, June 1990.

[56] J. Gallagher, M. Codish, and E. Shapiro. Specialization of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing*, 6(2,3):159–186, 1988.

[57] R. Giacobazzi, S. K. Debray, and G. Levi. A generalized semantics for constraint logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 581–591, ICOT, Japan, 1992. Association for Computing Machinery.

[58] F. Gianotti and M. V. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *LNCS 528, Programming Language Implementation and Logic Programming 1991*, pages 323–334. Springer-Verlag, August 1991.

[59] R. Giocabazzi and L. Ricci. Pipeline Optimizations in AND-Parallelism by Abstract Interpretation. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 291–305. MIT Press, June 1990.

[60] S. Gregory. *Parallel Logic Programming in PARLOG*. Addison–Wesley, 1987.

[61] S. Gregory and R. Yang. Parallel Constraint Solving in Andorra-I. In *International Conference on Fifth Generation Computer Systems 1992*, pages 843–850. ICOT, Tokyo, Japan, June 1992.

[62] G. Gupta. Paged Binding Array: Environment Representation for And-Or Parallel Prolog. Technical Report TR-91-24, University of Bristol, Computer Science Department, October 1991.

[63] G. Gupta and M. V. Hermenegildo. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *LNCS 569, ICLP'91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, pages 146–158. Springer-Verlag, June 1991.

[64] G. Gupta and B. Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *Proceedings of the North American Conference on Logic Programming*, pages 332–349. MIT Press, October 1989.

[65] G. Gupta and B. Jayaraman. On Criteria for Or-Parallel Execution Models of Logic Programs. In *Proceedings of the North American Conference on Logic Programming*, pages 604–623. MIT Press, October 1989.

[66] G. Gupta and V. Santos Costa. And-Or Parallelism in Full Prolog with Paged Binding Arrays. In *LNCS 605, PARLE'92 Parallel Architectures and Languages Europe*, pages 617–632. Springer-Verlag, June 1992.

[67] G. Gupta and V. Santos Costa. Complete and Efficient Methods for Supporting Side-Effects and Cuts in And-Or Parallel Prolog. In *PDP '92*, pages 288–295. IEEE, November 1992.

[68] G. Gupta, V. Santos Costa, R. Yang, and M. V. Hermenegildo. IDIOM: Integrating Dependent and-, Independent and-, Or-parallelism. In *Logic Programming: Proceedings of the International Logic Programming Symposium*, pages 152–166. MIT Press, October 1991.

[69] G. Gupta and D. H. D. Warren. An Interpreter for the Extended Andorra Model. Preliminary report, Department of Computer Science, University of Bristol, November 1991.

[70] S. Haridi. A Logic Programming Language based on the Andorra Model. *New Generation Computing*, 7(2,3):109–125, 1990.

[71] S. Haridi and P. Brand. Andorra Prolog–an integration of Prolog and committed choice languages. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, 1988.

[72] S. Haridi and S. Jansson. Kernel Andorra Prolog and its Computational Model. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 31–46. MIT Press, 1990.

[73] B. Hausman, A. Ciepielewski, and A. Calderwood. Cut and Side-Effects in Or-Parallel Prolog. In *International Conference on Fifth Generation Computer Systems 1988*, pages 831–840. ICOT, 1988.

[74] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[75] M. V. Hermenegildo. An Abstract Machine for Restricted And-Parallel Execution of Logic Programs. In E. Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 25–39. Springer-Verlag, July 1986.

[76] M. V. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[77] M. V. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 40–54. Imperial College, Springer-Verlag, July 1986.

[78] M. V. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[79] M. V. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *The Journal of Logic Programming*, 13(1, 2, 3 and 4):349–366, 1992.

[80] T. Hickey and S. Mudambi. Global compilation of Prolog. *The Journal of Logic Programming*, pages 193–230, November 1989.

[81] R. Hill. LUSH-Resolution and its Completeness. Dcl memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.

[82] K. Horiuchi. Less Abstract Semantics for Abstract Interpretation of FGHC Programs. In *International Conference on Fifth Generation Computer Systems 1992*, pages 897–906. ICOT, Tokyo, Japan, June 1992.

[83] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *Logic Programming Proceedings of the North American Conference, 1989*, pages 154–165. MIT Press, October 1989.

[84] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings Fourtheenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119. "ACM", 1987.

[85] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 196–218. University of Melbourne, "MIT Press", May 1987.

[86] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the International Logic Programming Symposium*, pages 167–186. MIT Press, October 1991.

[87] S. Janson and J. Montelius. Design of a Sequential Prototype Implementation of the Andorra Kernel Language. Sics research report, in preparation, Swedish Institute of Computer Science, 1992.

[88] G. Janssens. *Deriving Run Time Properties of Logic Programs by Means of Abstract Interpretation*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, March 1990.

[89] N. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of PROLOG. In *Abstract Interpretation of Declarative Languages*, chapter 6, pages 124–142. Ellis-Horwood, September 1987.

[90] L. V. Kalé. The REDUCE OR process model for parallel execution of logic programming. *The Journal of Logic Programming*, 11(1), July 1991.

[91] T. Kanamori. Abstract interpretation based on Alexander templates. *The Journal of Logic Programming*, 15(1 & 2):31–54, January 1993.

[92] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *International Symposium on Logic Programming*, pages 468–477. San Francisco, IEEE Computer Society, August 1987.

[93] S. Kliger and E. Shapiro. A Decision Tree Compilation Algorithm for FCP(|,:,?). In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1315–1336. MIT Press, August 1988.

[94] S. Kliger and E. Shapiro. From Decision Trees to Decision Graphs. In *Proceedings of the North American Conference on Logic Programming*, pages 97–116. MIT Press, October 1990.

[95] F. Kluźniak and S. Szpakowicz. *Prolog for Programmers.* Academic Press, 1985.

[96] M. Korsloot and E. Tick. Compilation Techniques for Nondeterminate Flat Concurrent Logic Programming Languages. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming.* MIT Press, 1991.

[97] R. A. Kowalski. *Logic for Problem Solving.* Elsevier North-Holland Inc., 1979.

[98] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A Generic Abstract Interpretation Algorithm and its Complexity Analysis (extended abstract). In *Proceedings of the Eighth International Conference on Logic Programming*, pages 64–78. MIT Press, June 1991.

[99] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[100] E. Lusk, R. Butler, T. Disz, R. Olson, R. A. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[101] M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 858–876. University of Melbourne, "MIT Press", May 1987.

[102] H. Mannila and E. Ukkonen. Flow Analysis of Prolog Programs (extended abstract). In *Proceedings 1987 Symposium on Logic Programming*, pages 205–214. IEEE Computer Society, September 1987.

[103] A. Mariën, G. Janssens, A. Mulkers, and M. Bruynooghe. The impact of abstract interpretation: an experiment in code generation. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 33–47. MIT Press, June 1989.

[104] K. Marriot and H. Søndergaard. Bottom-up Abstract Interpretation of Logic Programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 733–748. MIT Press, August 1988.

[105] K. Marriot and H. Søndergaard. Analysis of Constraint Logic Programs. In *Logic Programming Proceedings of the 1990 North American Conference*, pages 531–547. MIT Press, October 1990.

[106] H. Matsumoto. A Static Analysis of Prolog Programs. *SIGPLAN Notices*, 20(10):48–59, October 1985.

[107] M. Meier. Benchmarking of Prolog Procedures for Indexing purposes. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 800–807. Institute for New Generation Computer Technology, November 1988.

[108] C. S. Mellish. The Automatic Generation of Mode Declarations for Prolog Programs. DAI Research Paper 163, Department of Artificial Intelligence, Univ. of Edinburgh, August 1981.

[109] C. S. Mellish. Some Global Optimizations for a Prolog Compiler. *The Journal of Logic Programming*, 2(1), April 1985.

[110] C. S. Mellish. Abstract interpretation of PROLOG programs. In *Abstract Interpretation of Declarative Languages*, chapter 8, pages 181–198. Ellis-Horwood, September 1987.

[111] K. Muthukumar and M. V. Hermenegildo. Determination of Variable Dependence Information at Compile–Time Through Abstract Interpretation. In *Logic Programming Proceedings of the North American Conference, 1989*, pages 166–185. MIT Press, October 1989.

[112] K. Muthukumar and M. V. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 80–97. MIT Press, June 1989.

[113] K. Muthukumar and M. V. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[114] K. Muthukumar and M. V. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[115] K. Muthukumar and M. V. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *The Journal of Logic Programming*, 13(1, 2, 3 and 4):315–347, 1992.

[116] A. Mycroft and R. A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23(6):295–307, 1984.

[117] L. Naish. All Solutions Predicates in Prolog. In *International Symposium on Logic Programming*, pages 73–78. IEEE Computer Society, July 1985.

[118] L. Naish. Automating Control of Logic Programs. *The Journal of Logic Programming*, 2(3), October 1985.

[119] L. Naish. *Negation and Control in Prolog.* Lecture notes in Computer Science 238. Springer–Verlag, 1985.

[120] L. Naish. Parallelizing NU-Prolog. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1546–1564. MIT Press, August 1988.

[121] U. Nilsson. Abstract Interpretation: A Kind of Magic. In *LNCS 528, Programming Language Implementation and Logic Programming 1991*, pages 299–309. Springer-Verlag, August 1991.

[122] R. A. O'Keefe. Finite Fixed-Point Problems. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 729–743. University of Melbourne, "MIT Press", May 1987.

[123] I. W. Olthof. An Optimistic AND-Parallel Prolog Implementation. Master's thesis, Department of Computer Science, University of Calgary, 1991.

[124] T. Ozawa, A. Hosoi, and A. Hattori. Generation Type Garbage Collection for Parallel Logic Languages. In *Proceedings of the North American Conference on Logic Programming*, pages 291–305. MIT Press, October 1990.

[125] D. Palmer. The DAM: A Parallel Implementation of the AKL. Presented at the ILPS workshop on Parallel Logic Programming, October 1991.

[126] D. Palmer and L. Naish. NUA-Prolog: an Extension to the WAM for Parallel Andorra. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming.* MIT Press, 1991.

[127] L. M. Pereira. Logic control with logic. In J. Campbell, editor, *Implementations of Prolog*, pages 177–193. Ellis Horwood, 1984.

[128] L. M. Pereira, L. Monteiro, J. Cunha, and J. N. Aparício. Delta Prolog: a distributed backtracking extension with events. In E. Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 69–83. Springer-Verlag, 1986.

[129] L. M. Pereira and A. Porto. Intelligent Backtracking and Sidetracking in Horn Clause Programs - the Theory. Report 2/79, Departamento de Informatica, Universidade Nova de Lisboa, October 1979.

[130] A. Porto. Epilog: A language for extended programming in logic. In J. Campbell, editor, *Implementations of Prolog*, pages 268–278. Ellis Horwood, 1984.

[131] B. Ramkumar and L. Kalé. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *Proceedings of the North American Conference on Logic Programming*, pages 313–331. MIT Press, October 1989.

[132] L. Ricci. *Compilation of Logic Programs for Massively Parallel Systems*. PhD thesis, dipartimento di informatica, università di pisa, March 1990.

[133] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(23):23–41, January 1965.

[134] V. Santos Costa. Implementação de Prolog. Provas de aptidão pedagógica e capacidade científica, Universidade do Porto, Dezembro 1988.

[135] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[136] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 825–839. MIT Press, June 1991.

[137] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting full Prolog on the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

[138] V. A. Saraswat. Partial Correctness Semantics for $\mathbf{CP}[\downarrow,|,\&,;]$. In *Proceedings of the Foundations of Software Technology and Theoretical Computer Science Conference*, pages 347–368, December 1985.

[139] V. A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989. Available as Technical Report CMU-CS-89-108.

[140] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *IFIP Working Conference on Parallel Processing*, pages 305–318. Pisa, North Holland, May 1988.

[141] M. Sato, H. Shimizu, A. Matsumoto, K. Rokusawa, and A. Goto. KL1 Execution Model for PIM Cluster with Shared Memory. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 338–355. University of Melbourne, "MIT Press", May 1987.

[142] T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34(1,2):227–240, November 1984.

[143] E. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 27–83. MIT Press, Cambridge MA, 1987.

[144] E. Shapiro. *Concurrent Prolog: Collected Papers*. MIT Press, 1987.

[145] E. Shapiro. An Or-Parallel Execution Algorithm for Prolog and its FCP implementation. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 311–337. MIT Press, August 1988.

[146] E. Shapiro. The family of Concurrent Logic Programming Languages. *ACM computing surveys*, 21(3):412–510, 1989.

[147] K. Shen. *Studies of AND/OR Parallelism in Prolog*. PhD thesis, University of Cambridge, 1992.

[148] R. Sindaha. The Dharma Scheduler – Definitive Scheduling in Aurora on Multiprocessors Architecture. In *PDP '92*, pages 296–303. IEEE, November 1992.

[149] Z. Somogyi, K. Ramamohanarao, and J. Vaghani. A Stream AND-Parallel Execution Algorithm with Backtracking. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 2*, pages 1142–1159. The MIT Press, 1988.

[150] H. Søndergaard. An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction. In *LNCS 213, Proceedings 1st. European Symposium on Programming (ESOP 86)*, pages 327–338. Springer-Verlag, 1986.

[151] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[152] P. Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732. MIT Press, October 1989.

[153] P. Szeredi, M. Carlsson, and R. Yang. Interfacing Engines and Schedulers in OR-Parallel Prolog Systems. In *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 439–453. Springer Verlag, June 1991.

[154] A. Takeuchi. *Parallel Logic Programming*. PhD thesis, University of Tokyo, July 1990.

[155] J. Tan and L. I-Peng. Compiling Dataflow Analysis of Logic programs. In *Proceedings of the ACM SIGPlAN'92 Conference on Programming Language Design and Implementation*, pages 106–115, June 1992.

[156] J. Tanaka, K. Ueda, T. Miyazaki, A. Takeuchi, Y. Matsumoto, and K. Furukawa. Guarded Horn Clauses and Experiences with Parallel Programming. In *1986 Proceedings Fall Joint Computer Conference*, pages 948–954. IEEE Computer Society Press, November 1986.

[157] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[158] A. Taylor. Removal of Dereferencing and Trailing in Prolog Compilation. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 49–60. MIT Press, June 1989.

[159] A. Taylor. LIPS on a MIPS: Results from a Prolog Compiler for a RISC. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 174–185. MIT Press, June 1990.

[160] H. Tebra. Optimistic And-Parallelism in Prolog. In *PARLE: Parallel Architectures and Languages Europe, Volume II*, pages 420–431. Springer-Verlag, 1987. Published as Lecture Notes in Computer Science 259.

[161] J. Thom and J. Zobel. NU-Prolog reference manual, version 1.3. Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1988.

[162] E. Tick. Prolog Memory-Referencing Behavior. Technical Report CSL-TR-85-281, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, September 1985.

[163] E. Tick. *Parallel Logic Programming.* MIT Press, 1991.

[164] E. Tick and J. A. Crammond. Comparison of Two Shared-Memory Emulators for Flat Committed–Choice Logic Programs. In *International Conference on Parallel Processing*, volume 2, pages 236–242, Penn State, August 1990.

[165] H. Touati and A. Despain. An Empirical Study of the Warren Abstract Machine. In *International Symposium on Logic Programming*, pages 114–124. San Francisco, IEEE Computer Society, August 1987.

[166] K. Ueda. Making Exhaustive Search Programs Deterministic. In E. Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 270–282. Springer-Verlag, 1986.

[167] K. Ueda. Guarded Horn Clauses. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.

[168] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 3–17. MIT Press, June 1990.

[169] J. D. Ullman. *Database and Knowledge-Base Systems.* Computer Science Press, Maryland, 1988.

[170] H. Van Acker, R. Moolenaar, and B. Demoen. A parallel implementation of AKL. Presented at the ILPS workshop on Parallel Logic Programming, October 1991.

[171] M. H. van Emdem and G. J. de Lucena Filho. Predicate Logic as a Language for Parallel Programming. In K. L. Clark and S. A. Tärnlund, editors, *Logic Programming*, pages 189–198. Academic Press, London, 1982.

[172] M. H. van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, October 1976.

[173] P. Van Hentenryck. *Constraint Satisfation in Logic programming.* MIT Press, 1989.

[174] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.

[175] P. Van Roy, B. Demoen, and Y. D. Willems. Improving the execution speed of compiled Prolog with modes, clause selection and determinism. In *TAPSOFT'87*, pages 111–125. Springer Verlag, 1987.

[176] P. Van Roy and A. M. Despain. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. In *Logic Programming Proceedings of the 1990 North American Conference*, pages 501–515. MIT Press, October 1990.

[177] A. Waern. An Implementation Technique for the Abstract Interpretation of Prolog. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 700–710. MIT Press, August 1988.

[178] D. H. D. Warren. WARPLAN: a System for Generating Plans. DCL Memo 76, University of Edinburgh, June 1974.

[179] D. H. D. Warren. Implementing Prolog - Compiling Predicate Logic Programs. Technical Report 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[180] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[181] D. H. D. Warren. Prolog Engine. Technical report, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, April 1983. Unpublished draft.

[182] D. H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

[183] D. H. D. Warren. The Andorra model. Presented at Gigalips Project workshop, University of Manchester, March 1988.

[184] D. H. D. Warren. Extended Andorra model. PEPMA Project workshop, University of Bristol, October 1989.

[185] D. H. D. Warren and S. Haridi. Data Diffusion Machine–a scalable shared virtual memory multiprocessor. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, 1988.

[186] D. H. D. Warren and F. C. N. Pereira. An Efficient, Easily Adaptable System For Interpreting Natural Language Queries. *American Journal of Computational Linguistics*, 8(3-4):110–122, 1982.

[187] D. H. D. Warren, L. M. Pereira, and F. C. N. Pereira. Prolog—The Language and its Implementation Compared with Lisp. *ACM SIGPLAN Notices*, 12(8):109–115, 1977.

[188] D. S. Warren. Efficient Prolog Memory Management for Flexible Control Strategies. *New Generation Computing*, 2:361–369, 1984.

[189] R. Warren and M. V. Hermenegildo. On the Practicality of Global Flow Analysis of Logic Programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

[190] W. H. Winsborough. Path-Dependent Reachability Analysis for Multiple Specialization. In *Logic Programming Proceedings of the North American Conference, 1989*, pages 133–153. MIT Press, October 1989.

[191] R. Yang. *P-Prolog a Parallel Logic Programming Language*. World Scientific, 1987.

[192] R. Yang. Solving Simple Substitution Ciphers in Andorra-I. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 113–128. MIT Press, June 1989.

[193] R. Yang and H. Aiso. P-Prolog: a Parallel Logic Language Based on Exclusive Relation. In E. Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 255–269. Springer-Verlag, July 1986.

[194] R. Yang, A. Beaumont, I. C. Dutra, V. Santos Costa, and D. H. D. Warren. Performance of the Compiler-based Andorra-I System. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 150–166. MIT Press, June 1993.

[195] N.-F. Zhou, T. Takagi, and U. Kazuo. A Matching Tree Oriented Abstract Machine for Prolog. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 158–173. MIT Press, 1990.