Ricardo Nuno de Sousa Lopes

An Implementation of the Extended Andorra Model

Tese submetida à Faculdade de Ciências da Universidade do Porto para obtenção do grau de Doutor em Ciência de Computadores

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2001

In memory of my beloved niece Eduarda.

You will always hold a very special place in my heart.

>> – <<

A Loved One is not gone, until they are forgotten.

And to live in the hearts of those left behind is to

Live Forever!

Acknowledgments

First of all, I would like to thank my supervisors, Vítor Santos Costa and Fernando Silva. I am very grateful for their constant support, encouragement and guidance. A special thank you to Fernando Silva for doing the first review of this thesis and for making it readable. To Vítor Santos Costa I thank for the in-depth revisions, comments and suggestions towards the improvement of this thesis.

I would also like to testify that I have been very fortunate to work with Vítor and Fernando. I have learned much from their vast experience and knowledge in logic programming. The discussions we had were definitely a major influence to this work.

I also thank Ricardo Rocha, Michel Ferreira, Álvaro Figueira, Luís Lopes, Eduardo Correia, and many others at LIACC for our many discussions and for their friendship.

My research has also benefited from the ideas and discussions with Salvador Abreu, Gopal Gupta and Enrico Pontelli. So, I am also thankful to them.

This work has been supported by Fundação da Ciência e Tecnologia and PRAXIS (Praxis Grant BD/9277/96). Thank you for supporting this research.

Last, but not least, I would like to thank my wife for always being there, to my parents for always believing in me, to my brother, my sister and their families for all the support they gave me and finally, to my daughter for making my life doubly happy.

Abstract

Logic programming provides a high-level view of programming, giving implementers a vast latitude in what techniques to research towards obtaining the best performance for logic programs. Progress in coroutining, constraints, and tabulation allow for substantial reductions in search space over the traditional Prolog selection function, and provide a basis on which to build advanced features and support novel applications. Parallelism further allows logic programs to transparently exploit the advantages of the new parallel machines that are more than ever gaining in popularity.

Towards obtaining maximum performance, one of the holy grails of logic programming has been to design computational models that could be executed efficiently and would allow both for a reduction of the search space and for exploiting all the available parallelism in the application. These goals have motivated the design of the Extended Andorra Model, a novel model where goals that do not constrain non-deterministic goals can execute first.

In this work we present the Basic design for Extended Andorra Model (BEAM), a design that builds upon David H. D. Warren's original EAM with Implicit Control. We provide a complete description of an EAM kernel as a set of rewrite and control rules, and we evaluate these rules through a prototype implementation. We present the major data structures and execution algorithms that are required for efficient execution. We discuss the basic data structures required to implement the model and some issues on extending the EAM to avoid infinite computations. A detailed study of our prototype performance and memory usage is also included. Lastly, we propose a novel parallel execution model for the BEAM.

Resumo

O Prolog é uma linguagem de programação lógica de alto-nível que tem sido muito investigada tendo em vista conseguir-se obter um elevado desempenho na execução das aplicações. O progresso em *coroutining*, programação por restrições e tabelação permitiram reduções substanciais no espaço de procura da função de procura do Prolog, e são a base para a construção de aplicações mais ambiciosas. O uso de paralelismo, por outro lado, permite que as aplicações lógicas explorem de forma transparente as vantagens das arquitecturas paralelas, de forma a melhorar ainda mais esse desempenho.

Obter o máximo desempenho nas aplicações foi desde sempre um dos maiores objectivos da investigação na programação lógica. Este objectivo só poderá ser conseguido através de novos modelos de computação que sejam capazes de executar eficientemente os programas lógicos através da redução do espaço de procura e através da extracção do paralelismo existente nesses mesmos programas. Estes objectivos motivaram o desenvolvimento do Extended Andorra Model (EAM), um novo modelo que permite que os predicados sem restrições não determinísticas possam ser executados primeiro.

Neste trabalho apresentamos um sistema criado com base na EAM original de David H. D. Warren com controle implícito que designamos por Basic design for Extended Andorra Model (ou BEAM). Apresenta-se uma descrição completa do modelo, das regras de execução e do controle de execução. São referidos os principais algoritmos de execução do sistema bem como todas as estruturas de dados associadas que permitem uma execução eficiente das aplicações. São também discutidas algumas extensões ao modelo EAM que permitem um melhor controle da execução e que servem para evitar computações infinitas. Incluímos igualmente um estudo detalhado sobre o desempenho e comportamento da gestão de memória do sistema. Por fim, apresenta-se um novo modelo de execução paralelo que poderá ser implementado sobre a BEAM.

Résumé

La programmation logique permet un grand niveau de programmation, donnant aux programmateurs une vaste latitude de techniques pour obtenir la meilleure exécution pour les programmes logiques. Le progrès dans le *Co-routining*, *contraints*, et la *tabulation* permettent des réductions substantielles de l'espace de recherche au-dessus de la fonction traditionnelle de sélection de Prolog et fournissent une base pour la construction d'applications plus ambitieuses. Le parallélisme permet que les programmes logiques exploitent d'une manière transparente les avantages des nouvelles machines parallèles qui gagnent de plus en plus de popularité.

Pour obtenir la meilleure performance, un des plus grands objectifs de la programmation logique c'est de concevoir les modèles de calcul qui peuvent être exécutés efficacement et permettent encore une réduction de l'espace de recherche et exploiter tout le parallélisme disponible de l'application. Ces buts ont motivé la conception du Extended Andorra Model, un nouveau modèle où les buts qui ne contraignent pas des buts non déterministes peuvent exécuter d'abord.

Dans ce travail nous présentons la conception pour le Basic design for Extended Andorra Model (BEAM), une conception qui est construite sur David H. D. Warren EAM initiale avec le contrôle implicite. Nous fournissons une description complète d'un modèle EAM comme un ensemble de règles de réécriture et de contrôle, et nous évaluons ces règles par une mise en place d'un prototype. Nous présentons les structures de données et les algorithmes principaux d'exécution qui sont exigés pour l'exécution efficace. Nous discutons les structures de données de base nécessaires pour mettre en application le modèle et quelques points pour éviter des calculs infinis en EAM. Une étude détaillée de notre utilisation d'exécution et de mémoire de prototype est également incluse. Bref, nous proposons un nouveau modèle d'exécution parallèle pour le BEAM.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Logic programming [76] is based on the idea that computation is controlled inference. It provides a high-level view of programming where programs are fundamentally seen as a collection of statements that define a model of the intended problem, and questions may be asked against this model. Prolog [33] is the most popular example of a logic programming language, and has been successfully used in applications such as artificial intelligence, database programming, circuit design, genetic sequencing, expert systems, compilers, simulation, and natural language processing.

Prolog relies on SLD resolution [65] and uses a straightforward left-to-right selection function and depth-first search rule. This computation rule is simple to understand and efficient to implement but, unfortunately, it is not the ideal rule for every logic program. In some cases the limitations of Prolog lead programmers to convoluted and non-declarative programs. It is well known that for many programs the left-to-right selection function is not effective at constraining the search space and may in fact lead to looping.

Research on improving the performance of logic programs has focused on three main alternatives: coroutining, tabling, and parallelism. Essentially, coroutining allows goals to execute only when sufficient data is available. The idea has been discussed since the initial days of logic programming [34, 30, 102, 93] and most modern Prolog systems support some form of coroutining [23, 118, 6]. Work on tabling is more recent: the idea is to store intermediate solutions to a query. The XSB-Prolog system [114] has shown that tabling can be quite useful in opening up novel applications for logic programming. Lastly, several different forms of parallelism [117, 49, 50, 24] can be exploited in logic programs with good results [84, 9, 63, 125, 37, 120, 89].

Ideally, we would like novel computational models for Prolog to achieve the following goals, presented by David H. D. Warren [151] as, in order of priority:

- *Minimum number of inferences*: this is achieved by trying never to repeat the same execution step (inference) in different locations of the execution tree.
- *Maximum parallelism*: this is achieved by allowing goals to execute as independently as possible, and combining all solutions as late as feasible.

The Basic Andorra Model (BAM) [149] is a known proposal where coroutining between deterministic goals constrains the search space and generates and-parallelism. Or-parallelism is achieved through exploiting alternatives of non-deterministic goals. The Andorra-I prototype [120] demonstrated that the BAM can successfully exploit the two forms of parallelism and that it can reduce the search space for a large category of programs. On the other hand, Andorra-I depends on finding deterministic applications. If determinacy cannot be found efficiently, there is no improvement over Prolog.

The limitations of the BAM motivated the development of the Extended Andorra Model [150], or the EAM. The key ideas for this model can be described as:

- Goals can execute (in parallel) as long as they are deterministic or *they do not need to bind external variables*;
- If a goal must bind external variables non-deterministically, the computation of this goal will *split*.

The Extended Andorra Model addresses the major problems of the Basic Andorra Model, and approximates the goals of minimizing the search space and maximizing the parallelism. The original proposal was based in the concept of rewriting and- and or-boxes. Research then evolved in two directions.

One approach was followed by Haridi, Janson and researchers at SICS who concentrated on the AKL [69], the Andorra Kernel Language, based on the principle that the advantages of the Extended Andorra Model justified a new programming paradigm that could subsume both traditional Prolog and the committed choice languages. AKL programs were constructed from guarded clauses, where the guard could be separated with a sequential conjunction, cut, or commit. AKL also included novel features such as ports [67], an elegant approach towards manipulating streams. AKL was shown to obtain acceptable performance in sequential and parallel implementations, such as Penny [89] [90]. Eventually, the AKL researchers have shifted their interest to Oz [134]. Oz provides some of the features of AKL such as the logical variable and encapsulation, but thread programming and search control must be programmed explicitly.

In contrast, David H. D. Warren and researchers at Bristol concentrated on the *Extended Andorra Model with Implicit Control* [151], where the goal was to obtain the advantages of the Extended Andorra Model with the least effort from the programmer. Towards this goal Gupta designed a proof-of-concept interpreter [56]. Work in the interpreter allowed experimentation with novel ideas for the EAM, such as lazy copying and eager producers to give finer control over search and improve parallelism.

Gupta and Pontelli later experimented with an extension of dependent and-parallelism that provide some of the functionality of the EAM through parallelism [55]. The insight was that goals that share variables may execute in parallel and mutually constrain their search-space. EDDAP shows how the EAM ideas are important in parallel logic programming systems, although in their case they do require parallelism in order to achieve some of the features of the EAM. Interestingly, in EDDAP the search space actually depends on how much parallelism is exploited.

Concluding, the EAM offers great potential both for improving performance of existing applications and for obtaining more declarative formulations of search and database applications.

1.1 Motivation

In this work we return to Warren's original goal, that is, to improving the efficiency of controlled inference. Towards this goal the objective of this thesis is the design and implementation of an Extended Andorra Model with implicit control.

Our goal is to have a simple design that will make it possible to support execution of Prolog programs effectively and efficiently, ideally without relying on user annotations. On the other hand, we do accept that in some cases programmer annotations will be necessary. Instead of forcing the programmer into a straight-jacket from the very beginning, we propose that several techniques for control should be experimented with, and eventually made available to programmers. Examples of techniques we would like to experiment with include declarations on predicates, pruning, sequencing operators, stability, and tabling.

Notwithstanding Gupta's excellent work on the EAM prototype, the EAM with implicit control was never fully implemented as a logic programming engine. The question therefore remained of whether it could achieve performance comparable to a version of the EAM with well defined control, such as the AKL. One very important motivation of our work was to achieve a sequential prototype that could serve as an evaluation of the model. We would like to prove that the EAM with implicit (or flexible) control is feasible. Ideally, we would be satisfied if our design could achieve performance comparable to other implementations of the Andorra Model.

Our interest in the EAM with implicit control has been reinforced by our experience with Prolog applications. It is often the case that Prolog (and CLP) programs are generated automatically. Often, these programs will not perform well with the Prolog execution strategy, but programmers will have hints towards what makes these programs perform well. Our goal can thus be resumed in a single sentence:

To build a system that will complement, not replace, Prolog, by providing a very flexible efficient execution mechanism that can support very flexible control mechanisms.

1.2 Contributions

Towards the goals of this dissertation we have designed the LIGHT-BEAM [77], a novel language framework that contributes upon the original EAM work by:

- Providing a complete description of an EAM kernel as a set of rewrite and control rules, and experiment with these rules through an implementation. We call this kernel design the BEAM [1], Basic design for Extended Andorra Model [78, 79]. We have designed the BEAM over a traditional Prolog system, YAP [38]. Initially, this reduces the amount of work we have to do in implementing our

[1] Not to be confused with the Erlang BEAM virtual machine [61]

prototype. Ultimately, we aim at making the BEAM an extension of this Prolog system that the user can exploit towards maximum performance in declarative applications where Prolog performs badly.

- Addressing three major issues over the original design: when to allow splitting, how to perform pruning, and how to obtain better termination properties. We show that the model offers scope for both implicit and explicit pruning and that tabulation can be naturally integrated into the EAM. We call this model the LIGHT-BEAM, that is, the Leftmost Innermost Goal Hopping with Tabulation over the Basic design for the Extended Andorra Model.

- Propose the RAINBOW [82], a parallel execution model for the BEAM. We present a general overview of how to distribute work, propose alternative approaches towards addressing the binding problem for the EAM, and present a scheduling strategy.

The BEAM is a main contribution of our work since is the first implementation of an Extended Andorra Model with implicit control. It serves to prove the feasibility of Warren's design for the EAM. Moreover the implementation shows that implicit control can be in fact quite effective for a sizeable number of applications, and that simple annotations can contribute to further improvements with little programmer effort.

One of the arguments against the EAM is that it would have a very expensive memory footprint. We also contribute with a in-depth study of the EAM memory usage [80, 81] that focus on the main data structures and algorithms used for memory management in our implementation, with focus on the garbage collector.

1.3 Thesis Outline

This thesis has ten chapters. On next chapter, chapter 2, logic programming and in particular Prolog, are discussed in more detail. Alternative logic programming languages, such as the committed-choice languages are also presented.

Chapter 3 gives an overview on the Andorra-based Languages and their implementations. We start with the Basic Andorra Model. Next we discuss the Extended Andorra

Model and the Andorra Kernel Language (AGENTS).

Chapter 4 presents a new model, the Boxed EAM[79], that fully specifies an Extended Andorra Model with implicit control. It details the main BEAM rewrite rules and adds rules to simplify and optimize the And-Or Tree. It also describes the control model used for the execution of Prolog programs.

Chapter 5 extends the BEAM model towards the LIGHT-BEAM [77] to support more flexible search, implicit and explicit pruning of the search space, and a mechanism to deal with the non-termination problem of the EAM.

Chapter 6 details the execution model of BEAM [77]. It describes in detail the And-Or Tree Manager and the Abstract Machine.

On chapter 7 we report on the memory management used in BEAM [78, 80, 81]. We address questions such as how effective are the techniques the BEAM uses to recover space, reuse memory and garbage collection and how garbage collection affects performance.

Chapter 8 presents the performance results of the BEAM system and compares them with state of the art Prolog implementations, AKL and Andorra-I.

On chapter 9 we propose RAINBOW[82], a parallel execution model for the BEAM. The chapter presents a general overview of how to distribute work, propose alternative approaches towards addressing the binding problem for the Extended Andorra Model and presents a preliminary scheduling strategy.

Chapter 10 presents the main conclusions of this work and suggests lines of research for future work.

The thesis includes an appendix containing the Prolog code for the benchmarks used during this thesis.

Chapter 2

Logic Programming

In this chapter we present the main issues on logic programming with a focus on the Prolog language and on its sequential implementation. We further discuss parallel programming and the committed-choice languages.

2.1 Logic Programming

Logic programming [76] is a programming paradigm based on a subset of a First Order Logic named Horn Clause Logic. A Horn clause is a clause where at most one of the literals is positive:

$$A \vee \sim B_1 \vee \sim B_2 \vee \ldots \vee \sim B_n \qquad \text{or} \qquad \sim B_1 \vee \sim B_2 \vee \ldots \vee \sim B_n$$

which is equivalent to a logical implication of a conjunction of literals:

$$A \Leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n \qquad \text{or} \qquad \Leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n$$

A more usual form of writing Horn clauses is:

$$A \leftarrow B_1, B_2, \ldots, B_n \qquad \text{or} \qquad \leftarrow B_1, B_2, \ldots, B_n$$

All the previous representations implicitly assume all variables are universally quantified. Note that the last form omits the $false$ in the case of clauses with no positive literals. Each atom A and B_i has the form $p(t_1, \ldots, t_m)$, where p is a predicate symbol and all t_i's are *terms*.

A term is either a constant (also called *atom*), a logical variable, or a compound term. A variable represents an unspecified term to which values can be assigned. This process is known as *instantiation* or *binding*. A variable can be bound only once, to a non-variable term or to a different variable. Compound terms are structured data objects of the form $f(t_1, \ldots, t_n)$ where f is a functor with arity n and each of the t_i is a term. A term is said to be *ground*, or *fully instantiated*, if it does not contain any variables. *Unification* is the process of verifying if there exists a set of variable instantiations that make two terms identical. A formal definition of unification is given by Lloyd [76].

The term A is denoted as the *head* of the clause, while the conjunction $B_1, \ldots, B_m$ represents the *body* of the clause. Each one of the B_i is also called a subgoal. If the head of the clause is empty the clause is called a *query*. If the body is empty, the clause is named a *fact*. Otherwise, if both head and body are non-empty, the clause is called a *rule*. A sequence of clauses whose head goals have the same functor form a *procedure* (or predicate). A procedure can be formed with facts and/or rules.

A logic program is composed by a set of rules or facts. Given a program P and a query $\leftarrow B_1, \ldots, B_n$, a program execution consists in finding for which assignments of values to variables the conjunction $B_1 \wedge \ldots \wedge B_n$ is a logical consequence of the program P, that is, $P \models B_1 \wedge \ldots \wedge B_n$.

The computation process is based on two mechanisms presented by Robinson [106]: *Resolution* and *Unification*. There are several strategies for applying the resolution rule. Prolog uses a top-down form of resolution designated as SLD-resolution [65]. In this form of resolution, a query is matched against a clause generating a new query called *resolvent*. The *resolvent* is compared with the remainder of the initial query and the body of the matching goal. This process continues recursively until either a goal fails at finding a matching clause, or until an empty query is generated.

We may have several clauses matching a goal and thus have several ways to search for a solution. It is up to the *selection function* to specify which goal of the query should be selected. The *search-space* is the SLD-tree that represents all the possible ways to solve a query from a particular program. Changing the selection function may change the search-space. Different languages and different semantics can thus be obtained by choosing different *selection functions*. This was expressed by Kowalski in the equation [75]:

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

Meaning that the same program can have different behavior if a different *control* is used during the evaluation of the program.

2.2 Prolog

Prolog was invented in Marseille by Colmerauer and his team [33] and is currently the most popular logic programming language.

A Prolog program is a set of Horn Clauses [76]. A group of clauses with the same name and arity form a predicate. Clauses can be rules or facts. Rules are divided into *head* and *body* separated by the symbol ':-' (read as *if*). The body is composed with goals separated by the symbol ',' (read as *and*). Prolog programs manipulate logical *terms*. In Prolog, variables name start with a capital letter and names for constants (or atoms) start with a lower case letter.

```
fib(1,1).
fib(2,1).
fib(N,R):- N>2, N1 is N1-1, N2 is N2-2,
           fib(N1,R1),
           fib(N2,R2),
           R is R1+R2.
```

Figure 2.1: An example of a Prolog program.

Figure 2.1 shows a small Prolog program that implements the Fibonacci recursive function. The first and the second clauses state that if the first argument of `fib` is `1` or `2` then the result is `1`. The third clause state that if the first argument (`N`) is a integer greater than two, then fibonacci will be called recursively with the first argument set to `N-1` and `N-2`. The sum of the outcome obtained in these two calls is the result of the clause. Note that the program relies on two *builtin* predicates, `is/2` and `</2`, which we will discuss in the sequence.

Prolog systems apply SLD-resolution with a fixed selection function: the leftmost goal is always selected first. Whenever several alternatives for a goal are available, Prolog follows the order of the clauses in the program. When an alternative fails, Prolog backtracks to the last previous state that has at least one unexplored alternative. The execution tree is thus explored in a depth-first left-to-right manner. Prolog also has some control operators and several builtin predicates not found within first order logic. These were introduced to make Prolog a practical language capable of solving real world problems. Some of these features may vary slightly for different Prolog systems. The most relevant extensions are:

- **Meta-logical predicates**: manipulate terms and query the state of the proof. The most well known are the `var/1` and `atom/1` family of builtins, that test the state of their arguments.

- **Extra-logical predicates**: perform I/O and dynamically change the Prolog database. The most important are `assert`, that adds a clause to the Prolog database, `retract`, that deletes a clause from the Prolog database, input builtins such as `read`, and output builtins such as `write`.

- **Control builtins**: perform control operations. `Cut` is the control operator most popular in Prolog programs. It appears as a goal in a clause. When executed it discards all alternatives created since the start of the current procedure.

- **Other builtins**: perform arithmetic operations (e.g. `is/2`), obtain all solutions for a goal (e.g. `findall/3`), support debugging or and allow interaction with the program's environment (e.g. `consult`).

An important characteristic of many of these predicates is that their behavior is order-sensitive. When using these predicates the programmer is usually expecting the standard Prolog order execution. Hence, different results may be attained with a different selection function.

Modern Prolog systems include substantial libraries of builtin predicates. For a more detailed description of the Prolog builtins readers should refer to Prolog manuals, such as SICStus Prolog´s [23] or YAP's [118]. An introduction to the Prolog language can be attained consulting [31, 135, 40, 73].

2.3 The Warren Abstract Machine

The first Prolog system was an interpreter written by Roussel and Colmerauer in the early 70's [33]. In 1977, David Warren developed the first Prolog compiler, the DEC-10 Prolog [146]. This system showed good performance, comparable to the best Lisp implementations of that time [153]. The DEC-10 Prolog system became very popular and was influential in the Japanese decision to use logic programming for their Fifth Generation Project [1].

Research on Prolog implementations lead David H. D. Warren to create the Warren Abstract Machine (WAM) [147] in 1983. The WAM is an "abstract machine" designed to be the target for the compilation of Prolog. It offers several advantages as easy compilation, portability, good performance, and compact code. For these reasons, the WAM has become the de facto standard for Prolog implementations. Next we give a brief description of the WAM. The reader should refer to the several tutorials for a more detailed information [7, 47, 74].

Prolog systems usually compile Prolog to WAM code, and then use an emulator to execute the WAM code and therefore the initial Prolog program execution. Examples of such Prolog systems include Quintus [104], YAP [118], and SICStus Prolog [21].

The WAM is a stack-based abstract machine. Its memory organization is illustrated in figure 2.2.

The computation state of the WAM is defined by the contents of the WAM stacks, and by the values WAM registers. We describe briefly the WAM data areas first:

- **Heap** or **global stack**: this stack holds the compound terms, list and structures, plus originally unbound variables.
- **Local stack**: holds environments and choice points. The environments, also known as local frames or activation records, store the bindings for variables local to a clause. Choice points contain all the necessary pointers and variables so that it can be possible to backtrack to a previous goal G, that is, so that we can restore the state of the computation in order to execute the next alternative clause for goal G.
- **Trail**: saves the addresses of bound variables that have to be reset during

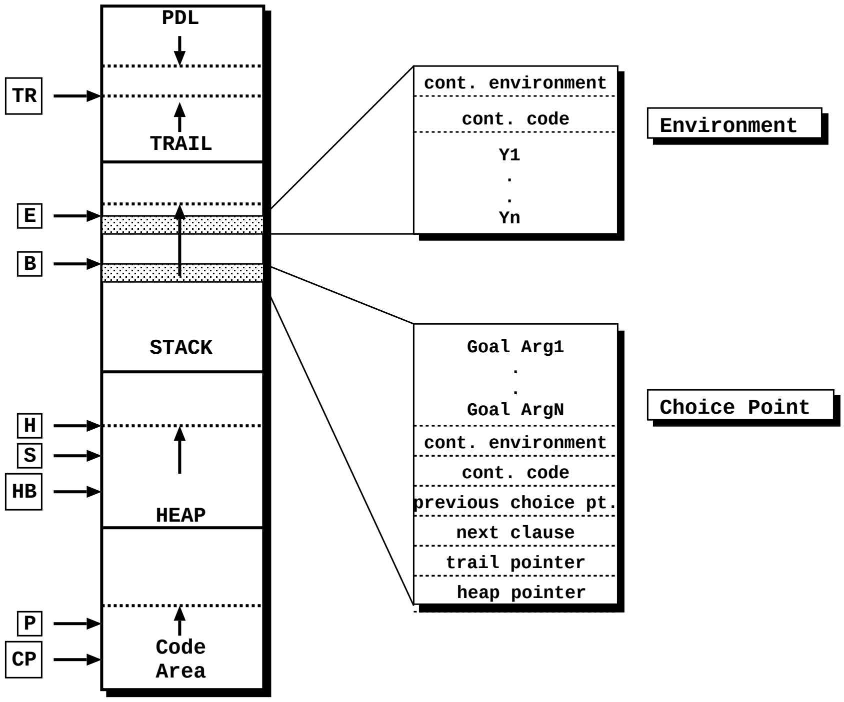

Figure 2.2: WAM Memory Areas

backtracking. The process of saving the addresses of variables is called *trailing*, and the process of restoring them to the unbound state is called *untrailing*.

- **PDL(Push Down List)**: auxiliary structure for the unification of compound terms.
- **Code Area**: holds the Prolog data-base, including the compiled coded for the Prolog programs.

Note that it is possible to change the organization of the memory areas without affecting the execution scheme.

The internal state of the WAM is stored in the following registers:

- `P`: Program Counter, points to the current WAM instruction being executed.
- `CP`: Continuation Pointer identifies the instruction to follow in the case of a successful return from the current call.
- `E`: pointer to the current environment.

- `B`: pointer to the most recent choice point.
- `TR`: top of trail.
- `H`: top of heap.
- `HB`: value of `H` in the most recent choice point.
- `S`: is used for unification of compound terms. Each argument can be accessed by successively incrementing the value of `S`.
- `Mode`: identifies the execution mode of WAM instructions that unify compound terms (read or write).
- `A1,A2,...`: argument registers.
- `X1,X2,...`: temporary variables.

It is common to use `Y[i]` to represent the i-th permanent variable of the current clause. The notation does not refer to abstract machine registers, instead it refers to variables whose values are recorded in environment slot i.

2.3.1 Representing different objects in memory

Prolog variables may represent any object. Hence, at runtime one must be able to detect which type of objects the variable actually represents. The WAM thus represents terms as tagged words, containing a tag field and a value field (see figure 2.3). The tag field describes the type of the term: atom, list or structure. The purpose of the value field depends on the type field. It can be a small integer, an atom identifier, the address of an unbound variable, or a compound term.

The WAM implements unbound variables as self-referential pointers, that is, they point to themselves. As Prolog allows two variables to unify, it may be necessary to follow a chain of pointers to access a variable value. This process is called *deferencing* the variable. During the unification of two variables care must be taken to avoid references to unsafe memory positions, that is, to avoid pointers to space that will later be discarded. The problem arises only in the local stack, hence correct bindings can be ensured by obeying the following rules:

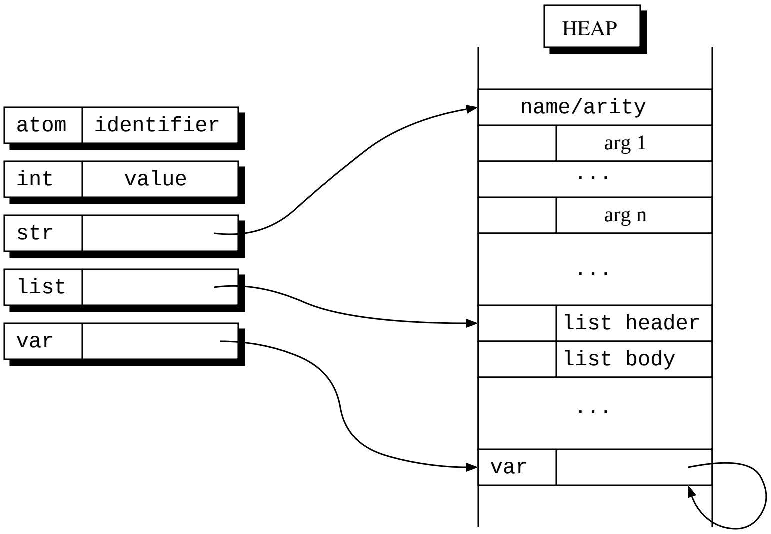

Figure 2.3: Terms representation in WAM.

1. Heap variables should never reference the local stack.
2. Older variables (less recently created) should never reference younger variables (more recently created) in the same stack.

Structures are represented in $n+1$ contiguous words. The first word stores the functor of the term, while the others store the arguments.

Lists are structures with a compact representation, they are simply represented as pairs of words. The head of the list is store on the first word and the tail on the second one.

2.3.2 Abstract machine instructions

The WAM instruction set can be divided into six different groups bearing on their functionality (note we use $\texttt{V}_\texttt{i}$ as shorthand for $\texttt{X}_\texttt{i}$ or $\texttt{Y}_\texttt{i}$):

- **Procedural instructions** launch new goals and terminate the current goal.
  - `call P,N`: call predicate `P` with arity `N`;
  - `execute P`: jump to predicate `P`;
  - `proceed`: return from a predicate;

  - `allocate`: creates an environment;
  - `deallocate`: removes the last environment;

- **Put instructions** set the argument registers with the arguments of the next goal to be executed.
  - `put_variable` $\mathtt{V_n}, \mathtt{A_i}$: creates a new variable and sets $\mathtt{V_n}$ and $\mathtt{A_i}$ to refer it;
  - `put_value` $\mathtt{V_n}, \mathtt{A_i}$: copies $\mathtt{V_n}$ to $\mathtt{A_n}$;
  - `put_constant` $\mathtt{c}, \mathtt{A_i}$: copies the constant `c` to $\mathtt{A_i}$;
  - `put_structure` $\mathtt{f/n}, \mathtt{A_i}$: initialises a compound term with functor `f/n` and sets $\mathtt{A_i}$ to refer it;
  - `put_list` $\mathtt{A_i}$: reserves space for a new list, and sets $\mathtt{A_i}$ to refer to the list;

- **Get Instructions** verify whether the arguments of a clause unify with the current values in the argument registers.
  - `get_variable` $\mathtt{V_n}, \mathtt{A_i}$: copies $\mathtt{A_i}$ to $\mathtt{V_n}$;
  - `get_value` $\mathtt{V_n}, \mathtt{A_i}$: unifies $\mathtt{V_n}$ with $\mathtt{A_i}$
  - `get_constant` $\mathtt{c}, \mathtt{A_i}$: unifies the constant `c` with $\mathtt{A_i}$;
  - `get_nil` $\mathtt{A_i}$: unifies the constant `nil` with $\mathtt{A_i}$;
  - `get_structure` $\mathtt{f/n}, \mathtt{A_i}$: if $\mathtt{A_I}$ bound, checks whether A_i points to a compound term with functor, `f/n`, sets the register `Mode` to *read*, and `S` to point at the first argument of the term; otherwise, that is if $\mathtt{A_i}$ points to a variable, allocates space to a new term, sets the register `Mode` to
    write, and initializes space for a new term with functor `f/n`;
  - `get_list` $\mathtt{A_i}$: unifies a list pointer with $\mathtt{A_i}$ and sets the register `Mode` to *write* if $\mathtt{A_i}$ points to a variable or to *read* otherwise;

- **Unify instructions** unifies arguments of compound terms.
  - `unify_variable` $\mathtt{V_n}$: copies the next structure argument to $\mathtt{V_n}$;
  - `unify_value` $\mathtt{V_n}$: unifies $\mathtt{V_n}$ with the next structure argument;
  - `unify_constant` `c`: unifies the constant `c` with the next structure argument;
  - `unify_nil`: unifies the constant `nil` with the next structure argument;

- `unify_void n`: skip the next `n` structure arguments;

Throughout, the register `S` points at the current argument and the register `Write` indicates whether we are unifying with a term or building a new term.

- **Choice point instructions**: allow backtracking through the several alternative clauses of a predicate.
  - `try Label`: creates a new choice point and jumps to `Label`;
  - `retry Label`: changes the retry address in the last choice point and jumps to `Label`;
  - `trust Label`: removes the last choice point and jumps to `Label`;
  - `try_me_else Label`: creates a new choice point with retry address `Label` and continues to the next instruction;
  - `retry_me_else Label`: change the retry address in the last choice point to `Label` and continues to the next instruction;
  - `trust_me`: removes the last choice point and continues to the next instruction;
- **Index instructions**: optimize the decision of which clauses unify with a given subgoal call. The idea is to use the first argument in order to skip clauses that may never unify.
  - `switch_on_term V, C, L, S`: a four-way jump on the type of the term in the first argument. It jumps to `V` if A_1 is a variable, to `C` if a constant, to `L` if a list or to `S` if a structure.
  - `switch_on_constant N, Tbl`: if A_1 is a constant jump to an address in a hash table with size `n`;
  - `switch_on_structure N, Tbl`: if A_1 is a structure jump to an address in a hash table with size `n`.

2.3.3 Compiling Prolog to WAM

Converting Prolog to non optimized WAM code is straightforward. Figure 2.4 presents a possible scheme to transform a Prolog predicate into WAM instructions. This code is further improved by using the *indexing* instructions described previously.

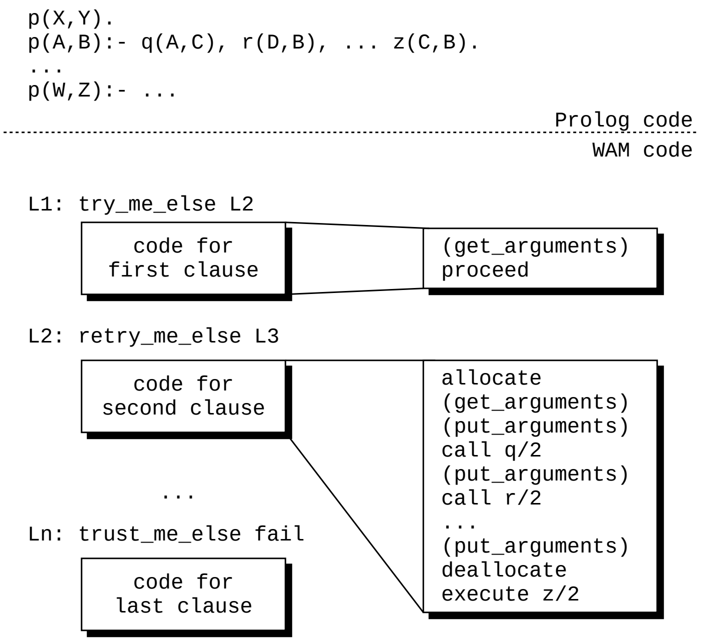

Figure 2.4: Compiling Prolog to WAM.

Predicate clauses are compiled individually as separate blocks of code. These blocks are then linked with `try_me_else` style instructions. The algorithm to compile clauses is as follows.

Facts are compiled into **(i)** a sequence of `get` instructions that unify the argument registers with the arguments of the fact and **(ii)** a `proceed` instruction.

Notice that facts and rules with a single goal in the body do not need an environment. Rules with several goals in the body require an environment and thus their corresponding WAM code must start with an `allocate` instruction, that creates the environment. Immediately next follows **(i)** a sequence of `get` instructions that unify the arguments with the head of the clause. Last, a rule has instructions per subgoal. Code for a subgoal always starts with a sequence of `put` instructions to prepare the arguments for the subgoal. If the subgoal is not the last one, the subgoal is invoked via the `call` instruction. Otherwise, the last subgoal ends with a `deallocate` instruction to release the environment and an `execute` instruction to transfer control.

Figure 2.5 show the Prolog source code and the compiled WAM code for the `append/3` predicate.

The WAM code starts with the indexing instruction `switch_on_term`, that jumps to the

```
append([],L,L).
append([X|L1],L2,[X|L3]):- append(L1,L2,L3).


append/3: switch_on_term  V1,C1,C2,fail
      V1: try_me_else     V2
      C1: get_atom        [], A1
          get_variable    X2,A2   >   Not needed
          get_value       X2,A3
          procceed

      V2: trust_me_else   fail
      C2: get_list   A1
          unify_variable  X4
          unify_variable  X1
          get_variable    X2,A2   >   Not needed
          get_list        A3
          unify_value     X4
          unify_variable  X3
          put_value       X1,A1   >
          put_value       X2,A2   >   Not needed
          put_value       X3,A3   >
          execute         append/3
```

Figure 2.5: The WAM code for `append/3`.

correct clause or set of clauses depending on the first argument. If the first argument is a variable it jumps to `V1`, that is the code that handle the choice point to execute both clauses. If the first argument is a constant, it jumps to `C1`, that is the code for the first clause. If the first argument is a list, it jumps to `C2`, that is the code for the second clause. Otherwise, if the first argument is a structure, it fails. The `try`'s instructions handle the choice points. The `try_me_else` instruction creates a choice point before executing the code for the first clause. The `trust_me_else` instruction removes the choice point before executing the code for the second (and last) clause. The WAM code for the first clause is a sequence of `get` instructions that unify the arguments registers with the arguments of the fact. Notice that the instruction `get_variable X2,A2` is really not needed since it unifies the variable `X2` with the argument `A2`, that in standard WAM implementations are the same. The code for the first clause ends with the `proceed` instruction. The code for the second clause also starts with a sequence of `get` and `unify` instructions that unify the arguments registers with the arguments of the clause. These instructions are followed by the `put` instructions that prepare the argument registers for the subgoal. Notice that in this example, none of the `put` instructions is really needed since `Xn` and `An` are the same. The code ends with the `execute` instruction that transfers control to the beginning of the `append` code.

The WAM includes several optimizations that improve both speed and memory usage. The most important ones are:

- *Last Call Optimization* [155]: this is an instance of the well-known *tail-recursive optimization* [48, 103] commonly found in the implementation of imperative and functional programming languages. This optimization is based on the observation that the environment space may be released *before* we transfer control to the last call, since it is of no use to any further computation. In this case, execution never needs to return to the procedure, so the last call can be simplified into a jump.

  Note that Prolog can only actually recover space if the computation of the clause's body was deterministic, that is, if we have no unexplored branches for subgoals in the body. Otherwise, we may retry a subgoal in the clause, and thus need the environment again.

- *Environment Trimming* [7]: it allows environment size to be progressively reduced during the execution of a clause by trimming permanent variables (variables stored in the environment) that are not needed by the rest of the computation.

- *Shallow Backtraking* [22]: the allocation of the choice point is delayed until success of head unification. This optimization is based on the fact that it is useless to create a choice point before the head unification succeeds.

2.4 Other Logic Programming Languages

The Prolog selection function has been the target of much criticism. Several authors have showed that applying different orders to the different goals of a logic program can significantly improve the search space to find the solutions of a problem. Among these authors were Kowalski [75], Warren and Pereira in the Chat system [152], and Porto and Pereira in the sidetracking execution principle [97].

Coroutining allows for a more flexible execution than the fixed selection used by Prolog. Coroutining allows goals to cooperatively produce and consume data, as shown in the following example:

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

In Prolog, the `validate` goal must wait for `value` goal to create a solution before executing. On the other hand, if the two goals act as coroutines, then `validate` can fail as soon as a partial and incompatible solution is found by `value`. Even better, it can itself help to reduce the search space by partially (or fully) instantiating `Solution`.

Explicit coroutining was introduced by several logic languages, such as Prolog-II [34], IC-Prolog [30], Epilog [102], MU-Prolog [93] and NU-Prolog [139]. Most modern Prolog systems such as SICStus Prolog [23], YAP [118] and Eclipse [6] include builtin support for coroutining.

The implementation of coroutining in Prolog requires new features to allow suspension and resumption of goals. These features have been discussed in detail by authors such as Boizumault [14], Carlsson [20] and Naish [93]. In a nutshell, coroutining requires delaying a goal until variables are bound. Following Colmerauer and Carlsson, this correspond to the freeze operation, `freeze(X,G)`. This operation sets up the system to invoke the goal `G` only when the variable `X` is bound [20]. Freezing a goal can be implemented over a traditional Prolog by adding support to:

- storing information about the suspended goals, namely, the arguments of the goal plus extra control information. This information can be stored as *goal frames* in the Heap [20] or in a separate stack [14].

- determine which goals are suspended on variables. This can be implemented by associating *suspension records* to the variables. The records point to the goal frames of the goals that have suspended on them. Some bookkeeping information may be necessary since a goal may suspend on a variable that already has goals suspended on it. One constant-time implementation is to let the new suspension record always refer to the older suspension records, while the variable refers to this new record. The drawback is that variables lose their single assignment property when using this scheme, and thus trailing such variables is more complex.

2.5 Parallelism in Logic Programs

Logic programming provides ample opportunities for parallelism and is therefore well suited for execution on multiprocessor architectures. There are two different approaches for the exploitation of parallelism from logic programs:

- **explicit**: parallelization is achieved through the use of extra constructs explicitly introduced in the source language, that aim at describing how the parallel computation should occur. The task of managing and extracting parallelism is left to the programmer. The advantage is that the programmer is free to choose what should or not run in parallel and how it should run. On the other hand, the programmer has the very complex task of detecting parallelism, managing parallelism, and guaranteeing the proper synchronization of the parallel tasks. Furthermore, sequential programs have to be annotated to be executed in parallel.

- **implicit**: parallelization of the execution can (potentially) occur without any extra annotations from the programmer. The exploitation of parallelism is automatically performed by the compiler and/or the runtime system. The advantage is that the development of the parallel software becomes as easy as developing a standard sequential program. On the other hand, extracting parallelism implicitly is not easy because it is hard to guarantee beforehand that a piece of work is worthy of being exploited in parallel.

One of the goals is this thesis is the implicit exploitation of parallelism from logic programs. We would like to use implicit control because logic programs are often complex. Moreover, we would like our framework to be close to Prolog, currently the language most used in logic programming. On the other hand, explicit model transfer the complexity of detecting and managing parallelism to the programmer. We believe this reduces the attractiveness of languages that require this model.

There are two main forms of implicit parallelism in logic programs: *or-parallelism* and *and-parallelism*. Or-parallelism results from evaluating several alternatives for a goal in parallel. And-parallelism results from evaluating several goals in a query concurrently. And-parallelism further subdivides into two different cases: independent and-parallelism and dependent and-parallelism. In the first case, parallel goals must not share or communicate via logical variables. In the second case, goals may collaborate executing in parallel.

Both forms of parallelism exist and can be exploited in most logic programming languages. Still, the semantics of the logic programming language that we want to support may affect which forms of parallelism are most promising. An example we describe later are the committed-choice languages where or-parallelism is limited because only one solution will eventually be considered. On the other hand, dependent and-parallelism is very hard to implement with the leftmost goal selection function used by Prolog.

There are a good number of successful systems that exploit a single form of parallelism. They include the or-parallel systems Aurora [83], Muse [9], and YapOr [110], the independent and-parallel system &-Prolog [62], DASWAM [133, 132] and ACE [100, 98, 51, 52, 53, 55, 99], and dependent and-parallel systems such as KL1 [126, 72, 26] and PARLOG [36].

2.6 The Committed-Choice Languages

The committed-choice, or concurrent, logic programming languages [131] evolved from the efforts to introduce coroutining and and-parallelism to Prolog. In these languages all goals that can commit may run in parallel and generate new goals in parallel.

Unlike Prolog's operational semantics, the committed-choice languages use the *process reading* of logic programs [144]. In the *process reading* of logic, each goal is viewed as a process and computations as a network of concurrent processes, with interconnections specified by the shared logical variables. The *process reading* of logic programs is most useful to build *reactive* systems [57], which contrast with *transformational* systems [105]. A *transformational* (closed) system receives an input at the beginning of its operation and yields an output at its end. On the other hand, the purpose of a *reactive* (open) system is not necessarily to obtain a final result, but to maintain some interaction with its environment. Some reactive systems, such as operating systems, database management systems, etc., do not obtain a final result at all since ideally they never terminate.

Concurrent logic languages employ don't-care non-determinism, meaning that once a transition has been taken the computation is committed to it and cannot backtrack or explore in parallel other alternatives. This contrasts with Prolog's don't-know non-determinism, where queries can have several solutions corresponding to selecting different clauses.

Another important characteristic of these languages is that goals can only choose a clause, or commit to a clause, when certain conditions on the goal are fulfilled. These conditions are expressed as guarded Horn clauses of the following form:

```
Head:- Guard | Body.
```

If the goal "satisfies" the `Head` and the `Guard` of the clause, the goal can commit to that clause. Both the `Guard` and the `Body` consist of a (possibly empty) conjunction of goals.

The commit operator to control don't-care non-determinism was introduced on the first committed-choice language proposal, "A Relational Language for Parallel Programming" by Clark and Gregory [28] in the mid eighties. In practice, the most well-known committed-choice languages are Shapiro's Concurrent Prolog [129], PARLOG [29] and GHC [130]. All these languages later evolved into several dialects, e.g., the KL1 dialect of GHC [138].

Shapiro's Concurrent Prolog [129] uses read-only variables: a goal must suspend until its read-only variables are instantiated. Concurrent Prolog evolved into the CP family of languages [131]. 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, resulting in quite expensive operations. Several CP languages address this problem in different ways, as explained on Shapiro's survey on the family of concurrent logic programming languages [131].

Clark and Gregory's PARLOG [29] family of languages rely on the notion of safe guards: it is up to the programmer to guarantee that no external variables will be bound. The programmer must also provide mode declarations to classify the arguments of a procedure as input or output. The execution of a clause will wait till the input arguments are sufficiently instantiated, and the output arguments are not allowed to be instantiated before commitment.

Ueda's GHC [130] (Guarded Horn Clauses) is a language where a process will suspend during the head unification or the execution of the guard if the process attempts to bind any of the caller's variables. No extra explicit annotations are necessary.

All the main committed-choice languages have *flat* versions. In flat languages the guard is possibly empty conjunction of pre-defined builtins such as equality, inequality and arithmetic predicates. Flat languages are simpler to implement and understand, but they do lose some expressiveness. One example of a flat language is KL1 [72, 126], the language used in the Japanese Fifth Generation Project [1]. The core of KL1 is the flat subset of GHC, the FGHC [142]. KL1 added meta-call predicates to control computations, a pragma language to control parallel execution plus other extensions for practical programming. KL1 was implemented in the high-performance KLIC system that runs on sequential, parallel, and distributed systems [26, 111].

Most committed choice languages implementations use abstract machines similar to the WAM but with strong differences on how goals are manipulated. We next mention some of the more important concepts used on the implementation of the committed choice languages.

Manipulation of goals : in contrast to Prolog, where the leftmost goal can always be executed, goals in the committed choice languages can be in several states. The KL1-B [72, 126, 125] for example, classifies these states as, or available for execution - *ready*, waiting for some variable to be instantiated - *suspended*, and being executed - *current*.

Suspension on variables : in the committed choice languages a goal may suspend on several variables. Several goals may also suspend on the same variable. Thus, these languages associate with each variable a linked list of *suspension records* with information on the suspended goals.

Organization of clauses : clauses are divided into the guard, where unification is read-only and can suspend, and the body, where, as in Prolog, unification can bind external variables.

Backtracking : in the committed choice languages, there is no true backtracking. Thus, the trail and choice points are not used. The disadvantage of not having backtracking, is that it is impossible to recover space. Thus the system must rely on garbage collection to recover unused memory.

2.6.1 Supporting don't-known non-determinism

The decision to support don't-care non-determinism solution simplifies the design and implementation of these languages. On the other hand, they lose the advantages of don't-know non-determinism. For example, search programs that are easily coded in Prolog are harder to write with the committed-choice languages [140]. This limits the usefulness of these languages for several important AI-style applications, for example. To address this problem, several authors proposed languages that allowed non-determinism in a committed choice environment. We next mention some of those proposals: CP[↓,|,&,;] [123], P-Prolog [157] and ANDOR-II [137].

CP[↓,|,&,;]

One of the first proposals that introduced don't-known non-determinism in the committed choice languages was Saraswat's CP[↓,|,&,;]. The language includes the Blocking Ask, ↓, that when applied to variable forces the computation to wait until the variable is instantiated. It also includes the don't-care commitment and don't-know commitment, &. If a clause with a don't-know commit & is selected during a goal execution the system must behave as if the current resolvent has been replaced by two (disjuntive) copies. Both copies can execute in parallel but without influencing each other.

P-Prolog

Yang's P-Prolog [157] extends the committed-choice languages with two fundamental concepts: an *exclusive relation* for clauses and don't-know non-determinism. Guarded Horn clauses are exclusive if only one clause matches the goal. If a group of clauses is not exclusive, all the alternative clauses can be tried. It is up to the programmer to specify which clauses are mutually exclusive and to divide exclusive clauses into different subsets.

The notion of exclusive clauses in P-Prolog was an important influence in the design of the Andorra model, which is presented in the next chapter.

ANDOR-II

Takeuchi and others proposed the ANDOR-II language [137], The ANDOR-II language is an AND- OR- computation model where predicates are divided into AND-predicates and OR-predicates. An AND-predicate is defined by a set of guarded clauses that behave like 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. The ANDOR-II language needs mode declarations for both AND- and OR-predicates. These modes are similar to PARLOG's and are used to classify the arguments of a procedure as read or write.

The AND- OR- parallel computation allows for non-determinate goals to reduce eagerly, that is, even when determinate goals are available. The eager evaluation of non-determinate goals may in some cases generate a search space larger than the one generated by selection functions that delay non-determinate goals as much as possible, such as Andorra's.

2.7 Final Remarks

This chapter presented a survey of logic programming languages with emphasis on the implementation of the Prolog language and on the committed-choice languages. In the committed-choice languages computation always commits to a single clause in contrast to Prolog where alternative clauses can represent different solutions. This condition restricts the number of Prolog applications that can be easily ported to these languages. However committing to a single alternative simplifies the implementation of coroutining. In the next chapter we present the Andorra model which has the advantages of the committed choice languages while still supporting Prolog.

Chapter 3

The Andorra Principle

Much research on Logic Programming has been performed on improving the performance of Prolog through And/Or parallelism and concurrent execution. The work on these areas is substantial and already mature, however the combination of this work has introduced new difficulties affecting both language design and implementation. In order to tackle this problem, several languages were proposed more recently, namely the Andorra-based languages.

This chapter gives an overview on the Andorra-based Languages and their implementations. We start by introducing the original work on the Basic Andorra Model, and then focus on the Extended Andorra Model. Lastly, including Gopal Gupta's EAM interpreter, the Agents Kernel Language, and related work.

3.1 Basic Andorra Model

The Basic Andorra Model [149], BAM, is a model for the execution of logic programs. The model classifies goals as *deterministic* if only one clause matches a goal, or *non-determistic* otherwise. The essential idea of the Basic Andorra Model is that, instead of executing goals according to the simple left-to-right selection rule used in standard Prolog implementations, if deterministic goals exist then one should execute them first and in (and-) parallel, otherwise one can select a non-deterministic goal to execute. The different branches of the choice point can be explored in (or-) parallel. The first advantage of the BAM is that deterministic goals need to be tried only once, rather than re-executed at different branches of the search space. The second advantage is that constraints from the deterministic goals may reduce the number of alternatives for other goals, and even make them deterministic as well (this process was the key idea in the concurrent logic programming languages [131]). Thus, the model supports implicit coroutining in the style of the flat committed-choice languages since goals can make other goals immediately executable by making them deterministic. The Basic Andorra Model also supports goals with several matching clauses and thus several solutions.

The main advantages of the Basic Andorra Model:

- Programs are less execution-order sensitive than in Prolog. For example, the query `?- member(X,L), L=[1,2,3].` has a number of infinite number of backtrack points in Prolog, while in the BAM it is finite.
- The model supports two forms of parallelism extracted implicitly from the program:
  - Dependent And-Parallelism, by running deterministic goals in parallel;
  - Or-Parallelism, by exploring different alternative clauses to a goal in parallel.
- The lazy style of non-deterministic execution has a good effect on search programs since the binding of variables by deterministic goals can turn other non-deterministic goals into deterministic ones.
- The model can adopt a reuse approach. For example, consider the query `?- a, b.` that invokes two goals. If the goal `b` is deterministic, it is computed only once, independently of the number of different solutions for goal `a`. Thus, the model differs from the approach taken in most of the current implementations of Prolog, which perform recomputation of subgoals, that is, each subgoal is completely recomputed for each solution of the subgoals in its left.

An Andorra implementation has to deal with novel problems inherent to the model:

- To determine which goals are deterministic can be hard. For example, consider the facts:

```
parent(mary,john).  parent(mary,mary).
```

and the query `?- parent(X,X).` Although the query unifies with a single fact, it is onerous for the system to predict these cases as deterministic in realtime execution.

- Concurrency can break Prolog semantics. As a trivial example consider the query `write(Solutions), fail.` Executing `fail` first can be dangerous since it can inhibit the display of solutions.

- Programs may do more work. Consider the program:

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

  and the query `?- a(X),b(X), lots_of_deterministic_work.`
  In standard Prolog the query `lots_of_deterministic_work` would never be executed because `a` and `b` do not have a common solution for `X`. However in the BAM model, as `a` and `b` are non-deterministic, the query `lots_of_deterministic_work` would be executed first, leading to the execution of unnecessary work. In the worst case, the BAM may enter an infinite loop that Prolog would never enter.

Languages based on the BAM include Andorra-I [122, 116], Pandora, and NUA-Prolog. Pandora [10, 11] extends Parlog to take advantage of the don't-know non-determinism of the BAM. NUA-Prolog [95] extended NU-Prolog [139] with the BAM, and included sophisticated code for determinacy checking. Arguably, Andorra-I is the most well-known of these systems, and the one we discuss next.

3.1.1 Andorra-I

The Andorra-I [122, 116] is an implementation of the Basic Andorra Model that exploits or-parallelism and determinate dependent and-parallelism while fully supporting Prolog. This system implements the or-parallelism based on Binding Arrays [148], in similar fashion to the or-parallel Aurora [83] system. Thanks to its similarity to Aurora in the or-parallelism component Andorra-I is able to use schedulers built for the Aurora. Indeed the Bristol scheduler [13], developed for Aurora, was plugged with little work into Andorra-I.

Andorra-I is designed to have several workers performing computations at the same time. A worker can be classified as master or slave. Each master is responsible for a team of workers that may or not include slaves. The master is responsible for creating new choice points and for managing and synchronizing their slaves. Workers in a team cooperate with others to share available and-work. Different teams of workers cooperate to share or-work [43, 42].

The Andorra-I system consists of three main components:

1. *Preprocessor*: [122, 115] responsible for generating the determinacy code and the sequencing information necessary to maintain the correct execution of Prolog programs. It consists of four components: a mode analyzer, a sequencer, a determinacy code generator, and a compiler.

2. *Engine*: [121] responsible for executing the Andorra-I programs. The engine consists of teams of workers, where each worker normally corresponds to a physical processor.

3. *Scheduler*: [44] responsible for finding new tasks (work) for workers that have completed their tasks. It is further subdivided into:

   - *AND-scheduler*: responsible for finding eligible and-work (goals) in a list of tasks that were not yet executed by the workers in the same team.

   - *OR-scheduler*: responsible for finding or-work, that is, an unexplored alternative [13].

   - *Reconfigurer*: responsible for arranging the workers into teams so that both and- and or-parallelism can be exploited when available.

Andorra-I results [116, 158] showed that the system is very effective at exploring both forms of parallelism and that it compares well with exclusive or-parallel and exclusive and-parallel systems. The system also proved that the extra complexity of the model is manageable since the basic performance compares quite well with other compiler-based systems such as SICStus Prolog [23].

There is also available a sequential version of Andorra-I in which the machinery that supports parallel execution such as binding arrays, variable locking, stack chunking and the scheduler interface are removed.

The Andorra-I system largely depends on the preprocessor for successfully running Prolog programs [122]. When converting Prolog programs into the code executable by the engine, the preprocessor must:

- guarantee correct execution of side-effects, cuts and other Prolog builtins;
- generate the determinacy code that will be used at runtime to find when a clause is determinate.

We next discuss these two issues in some more detail, as they are quite important for further work in the Andorra based models.

3.1.2 Supporting Prolog

The main functions of the preprocessor are to perform a mode analysis of the program through abstract interpretation, to use this information to recognize procedures containing pruning operators that are potentially "noisy", to protect calls to such "sensitive" predicates by restricting certain conjunctions to be executed sequentially, and finally to generate for each predicate a routine which will determine when calls to that predicate become executable through becoming clause determinate.

Side-effects The early evaluation of goals in the Andorra Model may interfere with side-effects builtins, such as `write`, or meta-predicates, such as `var`. Consider the following example:

```
test(X,Sol):- find_sol(X,Sol), write(Sol).
```

The `write` builtin assume left-to-right ordering, and executing it before the goal `find_sol` will clearly produce incorrect output.

To avoid such problems, Andorra-I associates to each builtin specialize determinacy code that tells when the builtin can execute early. In the case of side-effects builtins, this code simply states that the builtin can never be executed early.

There may be also some problems arising from the interaction between builtin execution and or-parallelism. Andorra-I solves these problems by only allowing a side-effect goal to be executed if the alternative branch is currently the leftmost.

Andorra-I allows calls to meta-predicates to be executed early if their arguments are sufficiently instantiated for their behavior to be determined; for example, `atom(X)` or `var(X)` can be executed as soon as `X` is instantiated. Similarly, calls to other builtins, such as arithmetic, can be executed as soon as their arguments are sufficiently instantiated.

Pruning operators A procedure contains a *noisy* pruning operator if it is possible for the pruning operator to restrict an external variable by imposing a certain binding and preventing other possible bindings. Pruning operators that do not restrict external variables are said to be *quiet*.

Andorra-I allows the commit pruning operator to be executed early if the commit is quiet, that is, if everything prior to the commit can succeed without binding any external variables non-uniquely. Similarly, Andorra-I allows the cut pruning operator to be executed early if all previous alternative clauses have failed and the cut is quiet, that is, if everything prior to the cut can succeed without binding any external variables non-uniquely.

Andorra-I will always prevent execution of goals in the body of a clause following a pruning operator until the pruning operator itself has been executed.

3.1.3 Detecting determinacy

The Andorra-I preprocessor includes an algorithm for finding determinacy. The basic idea of the algorithm is to generate at compile-time a decision graph, formed by a tree with “or” decisions corresponding to each argument of a goal. For each argument (tree), if all the decisions correspond to committing to a clause or failing, then the argument provides a path to show that the call is determinate, therefore the tree is satisfactory and the algorithm may terminate. Otherwise, the algorithm will try to expand other arguments until the tree is satisfactory or no more arguments are available. There are cases where the arguments itself are not sufficient for deterministically choose one of the alternatives. In these cases, the algorithm may also use builtins to achieve determinacy (if they exist in the beginning of the body in the goal). Consider for example the following Prolog code:

```
partition([X|L],Y,[X|L1],L2) :- X =< Y,
             partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :- X > Y,
           partition(L,Y,L1,L2).
partition([],_,[],[]).
```

The algorithm is unable to generate determinate code using the arguments alone. Instead, the test comparing `X` and `Y` is used to complete the determinacy detection.

There are cases, of course, where the algorithm is unable to identify determinacy. First, Andorra-I never uses user-defined builtins to test determinacy, even if their definition is quite simple. To explain the second problem, consider the following example:

```
father(john, paul).     father(patrick, paul).
```

The goal `father(X,X)` fails, but the preprocessor determinacy algorithm is unable to generate deterministic code for it. The execution will always have to wait until one of the arguments becomes instantiated. This example shows a second important limitation of Andorra-I: not being able to bind external variables to detect determinacy.

Despite the excellent results that the Andorra-I attained, the system is limited by the fact that coroutining and and-parallelism can only be exploited between determinate goals. Thus Andorra-I performance largely depends on finding deterministic work in the applications. This problem has been the motivation for the creation of a new model that allow dependent and-parallelism between non-determinate goals, the Extended Andorra Model.

3.2 The Extended Andorra Model

The Extended Andorra Model, EAM, was defined by D. H. Warren [151] as a set of computing rules that perform refutation of goals over Horn Clause programs. Compared to the traditional Prolog strategy, the EAM uses two strategies to reduce search:

1. The EAM follows the Andorra Principle and selects deterministic goals first.

2. The computation is represented as nested conjunctions and disjunctions, that is, as an *And-Or Tree*, not as a conjunction of clauses. This allows the scope of non-deterministic operations to be reduced.

The EAM extends the Basic Andorra Model in that it allows non-deterministic goals to execute in and-parallel, *as long as they do not bind external variables.* It thus allows the three main forms of parallelism:

- *Or-parallelism*, between alternatives in the search.
- *Independent and-parallelism*, between goals that do not share variables.
- *Dependent and-parallelism*, between goals that communicate via variables, say producer-consumers.

Next, we describe the basic ideas in the EAM based on Warren's original definitions [151].

3.2.1 David Warren's execution principles for the EAM

The EAM is formally defined by rewrite rules that manipulate And-Or Trees. Each node of the tree may either be an *and-box*, an *or-box*, or a *choice-box.* We will represent goals as $G_1 \ldots G_n$, variables as $X = \{X_1 \ldots X_m\}$, and a set of bindings as σ.

- **And-box** (figure 3.1): corresponds to a clause with subgoals G_1 to G_n. The variables X_1 to X_m represent the variables created in the box and σ represents the constraints on external variables imposed by the and-box.

$$[\exists X_1, \ldots, X_m : \sigma \& G_1 \& \ldots \& G_n]$$

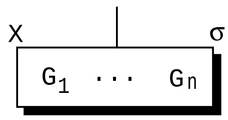

Figure 3.1: And-box graphical representation.

- **Or-box** (figure 3.2): Or-boxes correspond to alternatives to an unfolded goal. Each C_i represents an alternative clause for the goal (see figure

$$\{C_1 \vee \ldots \vee C_n\}$$

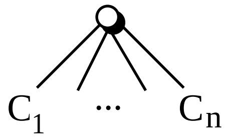

Figure 3.2: Or-box graphical representation.

- **Choice-box:** Choice-boxes are created when the clauses defining a procedure include a pruning operator, %. Each G_i represents the guard of the goal and each C_i represents the remainder of the clause. The original EAM supports two pruning operators, cut, !, and commit, |.

$$\{G_1\%C_1 \vee ... \vee G_n\%C_n\}$$

Next, we present the main EAM rules for Horn clauses, although we do not yet specify when can they be applied. David Warren originally proposed three basic operations for the Extended Andorra Model:

1. **Reduction** (figure 3.3) : Given a goal G, reduction replaces G by a new or-box containing at least one and-box. Each and-box is associated with the body of a clause whose head unifies with the goal.

$$G \rightarrow \{[\exists Y_1 : \ \sigma_1 \ \& \ C_1] \vee \ldots \vee [\exists Y_n : \ \sigma_n \ \& \ C_n]\}$$

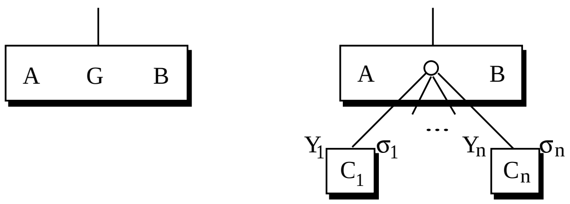

Figure 3.3: EAM reduction rule.

2. **Promotion** (figure 3.4) : In an or-box with a single alternative the lower and-box can be moved to the upper and-box. Therefore, this rule is responsible for propagating results from a local computation to the parent level.

$$[\exists X, Z : A \;\&\; [\exists W : Z = \theta(W) \;\&\; C] \;\&\; B] \rightarrow$$

$$[\exists X, Z, W : Z = \theta(W) \;\&\; A \;\&\; C \;\&\; B]$$

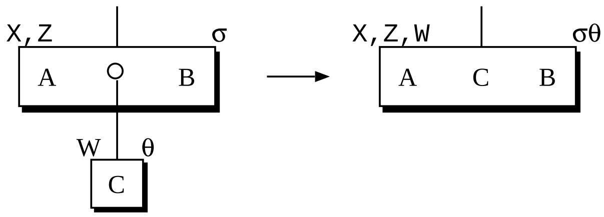

Figure 3.4: EAM promotion rule.

3. **Forking** (figure 3.5) : Forking distributes a conjunction across a disjunction. We say that this operation is non-deterministic because it allows more computation by extending the scope of disjunctions.

$$[\exists X : \sigma \;\&\; A \;\&\; \{C_1 \vee \ldots \vee [\exists Y : \theta \;\&\; C_i] \vee \ldots \vee C_n\} \;\&\; B] \rightarrow$$

$$\{[\exists X, Y : \theta\sigma \;\&\; A \;\&\; C_i \;\&\; B] \vee [\exists X : \sigma \;\&\; A \;\&\; \{C_1 \vee \ldots \vee C_{i-1} \vee C_{i+1} \vee \ldots \vee C_n\} \;\&\; B]\}$$

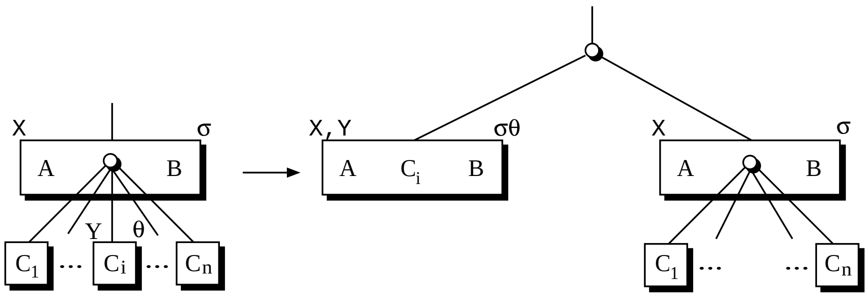

Figure 3.5: EAM forking rule.

3.2.2 Pruning

In the original EAM proposal [151], David H. Warren presented the pruning operations for the Extended Andorra Model as illustrated on figure 3.6. These pruning operations

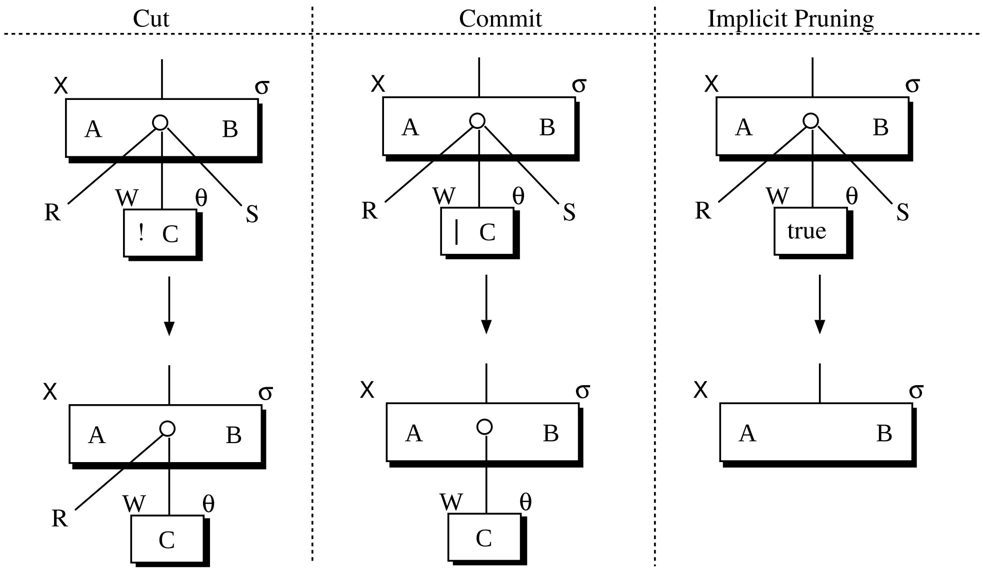

Figure 3.6: EAM pruning rules.

should only be applied for *quiet* cut and commit, and for implicit pruning when θ does not constrain external variables. A pruning operator is *quiet* if it does not impose any binding on external variables in the process of pruning. Otherwise the pruning operator is said to be *noisy*.

The definitions presented by David Warren for the pruning operations were not complete since it did not discuss questions relative to the use of the promotion and the forking rule involving boxes containing pruning operators or calls to side-effects builtins. Note that although choice-boxes were introduced to deal with the pruning operators, no special rules where ever presented to consider these boxes.

3.2.3 Control in the EAM

The Extended Andorra Model gives us a set of basic operations which are logically valid, and which can be controlled in a variety of ways. On the other hand, the unrestricted application of the described rules will surely lead to a completely unpredictable and undesirable computation.

The original EAM design tries to keep the control implicit, as much as possible, that is, it does not rely on information supplied by the programmer. The control decisions are based exclusively on information implicitly extracted from the program.

One of the aims of the Extended Andorra Model is to perform the least number of reductions to obtain the solutions for a goal. In [151], David Warren presents a set of intuitions on how to optimally evaluate $[\exists X : (P_1 \ \& \ldots \& \ P_n)]$:

- if we have no solution for X, we only want to work on goals which will lead to failure;
- if we have one or more solutions to X, it is safe to work on all goals;
- if there are multiple solutions for X, we want to perform all work that does not depend on X before installing different instantiations for X;
- if one goal can generate a unique value for X, allow that goal to be the *producer* and others to be *consumers*. Otherwise we need to select one goal to be a *non-determinate producer* and let other goals be consumers;
- use *non-determinate promotion* (forking) solely to transmit non-determinate bindings from producer to consumers, making copies of consumers in the process.

As long as it does not bind variables, 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 (see section 4.7). Ideally, one would like to provide an implicit control mechanism to prevent this problem. Unfortunately detecting speculative work depends on knowing a priori whether the computation will terminate, which is in general undecidable.

One often wants to perform computations for different goals as independently as possible. Although in order to obtain maximum independence between goals it is necessary to limit the number of goals that are allowed to delay forking. That is, forking should happen as soon as the goals that bind the variable, and only these goals, cannot apply any other operations. A solution as used in Gupta's interpreter is to give scope to variables rewriting clauses to mini-scope form [56].

3.2.4 An Example of EAM execution

To better understand the essential ideas of how the computation of a logic program proceeds in the EAM, we next present a very simple example:

```
g(X):- a(X), b(X).      b(4).           a(1).    c(1).
                        b(X):- c(X).    a(2).    c(3).
```

Figures 3.7 to 3.10 show several execution steps for this program and the query `?- g(X)`. Or-boxes are represented as small circles. And-boxes are represented as rectangles.

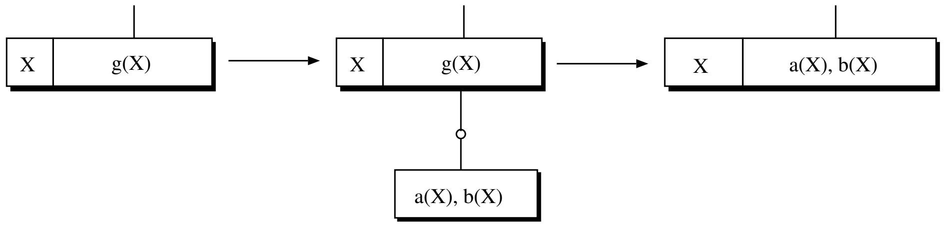

Figure 3.7: Reduction followed by a promotion.

On the first step (fig. 3.7), after starting execution, the EAM performs a reduction on `g(X)` followed immediately by a promotion since `g(X)` only has one alternative.

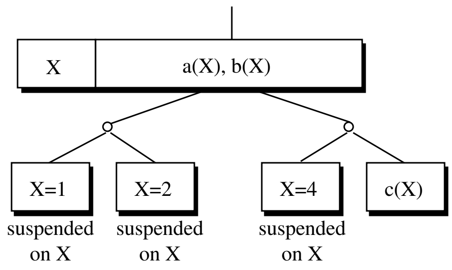

Figure 3.8: Reduction.

On the second step (fig. 3.8), both `a(X)` and `b(X)` are reduced. Notice that there are now two alternatives to `a(X)`. The first one tries to perform $X = 1$, leading to suspension because `X` is an external variable. The second alternative will also suspend when trying to make $X = 2$. There are also two alternatives to `b(X)`. The first one suspends when trying to make $X = 4$, but the second one can continue.

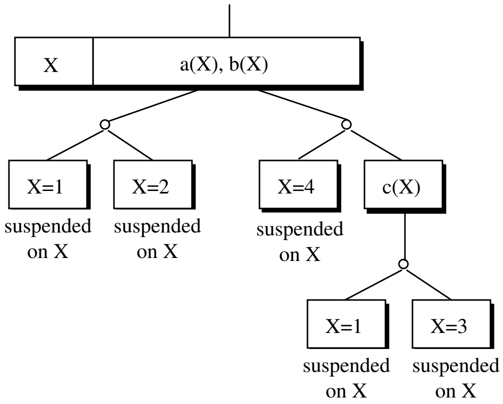

Figure 3.9: Reduction.

The next step (fig. 3.9) can only work on the alternative clause `c(X)` because all the others are suspended. A reduction is applied, creating two more alternatives. Both of them suspend immediately because they also try to assign a value to the external variable `X`.

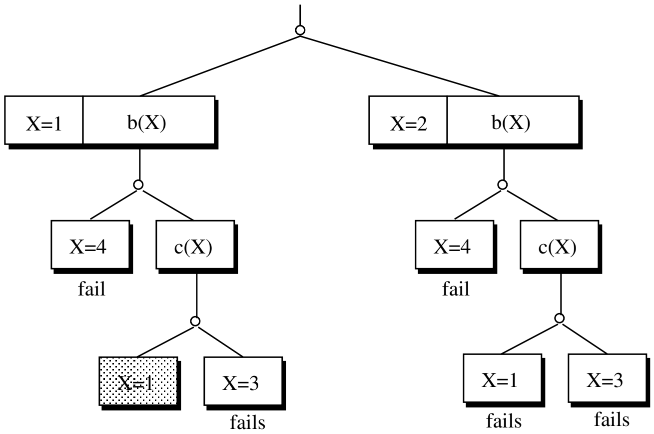

Figure 3.10: Forking (non-determinate promotion).

At this point, all possible alternatives are suspended, only the forking rule can be applied. We decided to choose the or-box that is the leftmost on the tree. After forking (fig. 3.10), we can see the tree divided in two, with an or-box at the root. The left side has `X` bound to 1. The right side has `X` bound to 2. After reactivating all the suspended and-boxes, most of them fail immediately, leaving just one solution, which is $X = 1$ on the left side of the tree.

3.3 Gupta's EAM interpreter

Gupta developed an experimental interpreter written in Prolog that implements the Extended Andorra Model with implicit control [56]. The objective of the interpreter was to study how the EAM rewrite rules could be used to minimize inferences and maximize parallelism and, to extract general principles which could approximate ideal behavior. The interpreter was never intended to be a system for general usage.

During the development of Gupta's EAM interpreter, a fourth rule was added [56]:

- **Substitution:** Substitution applies a constraint on a variable.

  $$[\exists X, Z : Z = \theta(W) \& C] \rightarrow [\exists X : C\theta]$$

  This rule allows all occurrences of a variable to be substituted by the current variable's value.

Gupta's interpreter also changed the EAM forking rule to be the *Global Forking* operation (figure 3.11) :

$$[\exists X : A \ \& \ \{C_1 \vee \ldots \vee C_n\} \ \& \ B] \rightarrow$$

$$\{[\exists X : A \ \& \ C_1 \ \& \ B] \vee \ldots \vee [\exists X : A \ \& \ C_n \ \& \ B]\}$$

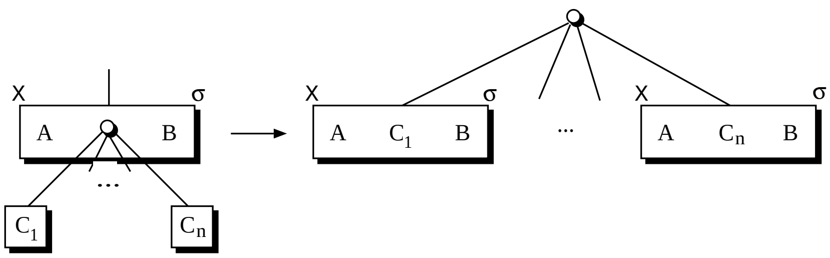

Figure 3.11: EAM global Forking rule.

Global forking is an extension to normal forking rule and is used to apply forking on all alternatives in a single step.

A first control scheme for the EAM was proposed by Gupta's [56] during the development of its interpreter, and it is illustrated in figure 3.12. The control scheme consists

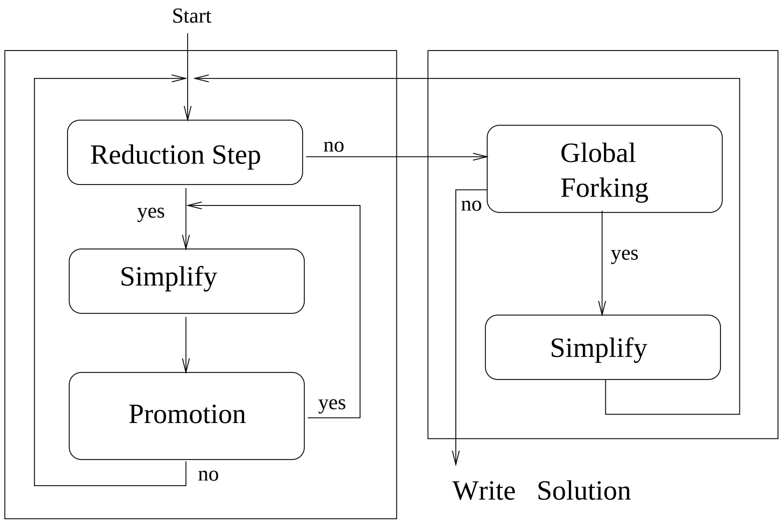

Figure 3.12: Control flow in the EAM.

of applying reduction and promotion steps as much as possible. Whenever it happens that an and-box constrains an external variable, it is suspended. *Simplification* steps are also used in the interpreter to maintain the data structures. Global forking is only applied when no other operation (reduction and promotion) can. It can lead to duplication of work, but it is necessary to open up the computation. In the EAM interpreter, if several goals can produce a non-determinate binding for a variable, the leftmost one will be chosen.

3.3.1 Eager forking and Lazy copying

In the original EAM a goal will always suspend when non-determinately 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. One important observation is that if the goal is the producer, there is no need for it to suspend. In fact, as Gupta proposed [56], the EAM can obtain extra parallelism by letting the producer continue, even through the variable is external to that goal. The computation may be forked immediately and the binding generated promoted. This operation is called *eager forking*. However as the forking is done eagerly, the consumer goals may not have reached the point of suspension. Therefore, to avoid duplication of work, instead of copying the consumer goals physically for every alternative of the producer, a pointer to the consumer goal is used. The physical copying is done only when the consumer goal reaches a suspend state.

Thus, the copy of the consumer is now done in a lazy fashion, hence the name *lazy copying*. The global forking rule 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 $C_1 \ldots C_n$ are producers of bindings of X, and G the consumer. G_p is a reference to a single copy of G. One can do even better, by copying parts of G_p as soon as they suspend.

To support this scheme the EAM should include a classification scheme for variables as guessable or non-guessable. This classification might be provided by the programmer, or generated automatically. A variable is annotated as *guessable* if the annotator 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, forking with lazy copying is performed. In contrast, forking on non-guessable variables is only performed if there is no other way to restart the computation.

3.4 Andorra Kernel Language

The Agents Kernel Language, AKL [68, 69] (originally called Andorra Kernel Language), is a general concurrent logic programming language based on Kernel Andorra Prolog, designed by Haridi and Janson[58]. Kernel Andorra Prolog is a language framework based on the same ideas as the Extended Andorra Model. AKL differs from the EAM in that it gives the programmer complete control over the execution through the use of wait guards. The kernel language clauses are definite clauses augmented with exactly one guard operator. The guarded clauses are assumed to be normalized in the sense that all user defined atomic goals will have the variable-pure form $p(v_1, \ldots, v_n)$ (v_i are different variables). All guarded clauses defining the same predicate must have the same guard operator.

Similar to the EAM, the computation model for AKL is represented as a series of rewrites rules that manipulate And-Or Trees. Each node of the tree may be either an *and-box*, an *or-box* or *choice-box*.

The BAM strategy of performing determinate goals in preference to non-determinate goals is a necessary condition in AKL. Another issue in AKL, similar to what can happen in the EAM, is the possibility of looping. Since the AKL language makes control explicit, it is up to the programmer to prevent such situations.

3.4.1 Rewrite rules

The rewrite rules below are to be understood as being applicable to any subgoal of a goal. That is, AKL defines the rewrite relation on goals $G \rightarrow G'$ as meaning that G' is obtainable from G by rewriting one occurrence of some subgoal X of G as X' in accordance with one of the rewrite rules $X \Rightarrow X'$ defined below.

For a sequence R of goals, $\sigma(R)$ denotes the conjunction of the constraint atoms in the sequence, or *true* if there are no constraint atoms in R. V and W represent the set of variables $\{x_1, \ldots, x_n\}$. They can be used as $and(R)_V$ or R_V meaning that V is the set of local variables to the box or to the goal respectively.

The AKL main execution rules are:

Local Forking

A program atom subgoal A may be rewritten by

$A \Rightarrow choice(and(G_1)_{V1}\%A_1, \ldots, and(G_n)_{Vn}\%A_n)$

using the sequence of clauses $H_i\ :-\ G_i\%A_i$ defining the predicate of A, and where V_i is the set of variables local to the box.

Failure Propagation

$and(R, fail, S)_V \Rightarrow fail$

Choice Elimination

$choice(R, fail\%A, S) \Rightarrow choice(R, S)$

Environment Synchronization

$and(R)_V \Rightarrow fail$

if $\sigma(R)$ is incompatible with the environment of the and-box.

Guard Distribution

$$choice(R, or(G, S)\%A, T) \Rightarrow choice(R, G\%A, or(S)\%A, T)$$

Deterministic Promotion

$$and(R, choice(C_V\%A), S)_W \Rightarrow and(R, C, A, S)_{V \bigcup W}$$

if C_V is satisfiable, and in case % is cut or commit, if % is also *quiet*.

Cut

$$choice(R, C_V!A, S) \Rightarrow choice(R, C_V!A)$$

if S is non-empty and C_V is satisfiable and quiet.

Commit

$$choice(R, C_V|A, S) \Rightarrow choice(C_V|A)$$

if R or S is non-empty and C_V is satisfiable and quiet.

Non-Deterministic Promotion

$$and(T_1, choice(R, C_V : A, S), T_2)_W \Rightarrow$$
$$or(and(T_1, C, A, T_2)_{V \bigcup W}, and(T_1, choice(R, S), T_2)_W)$$

if R or S is non-empty and as long as the rewrite is done within a *stable* and-box (Stability is explained below).

A sequence of applications of the transition rules described constitutes the AKL computational model. The control of AKL was described in [58]. The non-deterministic promotion and pruning are called *guessing* rules, while all the others all called *guess-free* rules. An application of a guess-free rule is always admissible. The non-deterministic promotion is allowed iff it is applied to a innermost *stable* subgoal. An application of pruning is admissible iff it is applied to a *stable* goal or if the solution on which the pruning is based is quiet.

The *stability* condition is the generalization in AKL of the basic Andorra principle of performing deterministic promotion before non-deterministic promotion. An application of environment synchronization, commit promotion, or cut promotion is *non-trivial* if the environment of the goal to which the operation is applied is satisfiable. 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. The main problem with the notion of stability as given is that it is very expensive to compute. For this reason the AKL implementations have replaced the definition of stability by a more manageable and sufficient condition for stability. AKL implementations approximate stability by marking and-boxes that suspend on external variables as not stable. These marks are never cleared, so some stable boxes may never be detected.

Both the BAM and the EAM default strategy only allow the non-determinate rule to be applied to goals whenever no determinate goals are available. The AKL is more aggressive since it allows the non-determinate rule to be applied immediately to *stable* goals, even if there are determinate goals available.

3.4.2 Implementations

AGENTS [70] is a sequential compiled AKL system that was implemented by Janson and Montelius. It provides a complete implementation of AKL with records, finite domain constraints (over integers), and port-based communication. It also provides assorted builtin agents, libraries, and other support.

The AGENTS implementation follows an WAM style stack based approach. The different objects are represented in memory using tags. The AKL program is compiled into a WAM style abstract machine. The abstract machine uses the indexing instructions to reduce the number of different alternatives and to build the choice-boxes. The procedural instructions control the flow of the execution in the and-boxes. `Get`, `put` and `unify` instructions are used in a similar way as in the WAM. A new group of instructions called the *guard instructions* support the AKL guard operators (*cut, commit, wait* and *halt*).

An AGENTS configuration consists of a tree of choice-boxes and and-boxes. The root of the tree is a choice-box called *root*. Or-boxes were not implemented in AGENTS, as every AKL predicate should exactly have one guard operator.

Choice splitting (or non-determinate promotion), is allowed if the current and-box is stable. A flag within the and-boxes detects stability. When constraining an and-box, all and-boxes in the path from (but not including) the current and-box to (but not including) the and-box at which the variable is local, are marked as unstable. An and-box that has been marked as unstable will never be marked as stable again. A *task stack* that contains a sequence of entries to choice-boxes with available work is used to guide execution after a choice splitting.

All run-time terms are allocated on the heap stack. The heap is also used for the allocation of the structures associated with the choice-boxes, and-boxes and suspensions. The heap depends on a stop-and-copy garbage collector to recover free space.

Penny[88, 87] is a parallel implementation of AKL developed by Montelius. It uses both and- and or-parallelism without user annotations. Penny uses a copy based scheme for bindings [89]. Montelius and Magnussson have used the SimICS [85, 86] simulator as a tool for performance debugging and for explaining the behavior of their system [90]. Their study allowed them to improve the performance of Penny and highlighted the importance of the system caches.

ParAKL[143, 91, 92] is a different AND/OR parallel implementation of AKL developed by Remco Moolenaar. ParAKL exploits parallelism by executing goals in an and-box in parallel , and by having a parallel implementation of the choice splitting rule. This implementation is different from Penny's, where both and- and or-parallelism can be exploited but the copying procedure is a sequential operation. A second difference from the Penny system is that ParAKL uses the PEPSys [12, 156] hashing scheme for bindings. The workers maintain a shared tree of hash tables that are updated to reflect the bindings environments of the different and-boxes.

There is another implementation by Doug Palmer, the DAM [94] that uses a binding scheme similar to the scheme used in the Penny System. DAM and Penny differ in the form the locality of variables is determined. In the DAM all variables that are local to an and-box are linked in a list. This implementation introduces a sequential component in the promotion operation.

Some interesting work based on AKL includes AKL(FD) - a concurrent language for Finite Domains programming [124, 19, 128], and Abreu's work on improving backward execution in the Andorra family of languages [3, 2, 5].

In [19] Carlson and others presented the language AKL(FD), an extension to AKL of the finite domain constraint system FD [96, 32], including antimonotone constrains [18]. Monotone FD rules are used as programming primitives when defining and propagating finite domain constraints. AGENTS offers a full implementation of AKL(FD), and is comparable in speed with state-of-the-art finite domain CLP languages. Their work show that the Extended Andorra Model can naturally support several domains of constraints, and that the issues in implementing solvers are in fact orthogonal to the main machinery for the EAM. Therefore, in the remainder of this thesis we will concentrate on the basic machinery to implement the EAM and, without loss of generality, use the Herbrand domain as our example domain.

Abreu's work avoids useless deduction/backtracking work by using the heuristic *a failed goal is likely to fail again*. The basic idea is to upon failure reschedule goals in such a way that goals that have already failed, execute first. They have implemented a prototype [2] over AGENTS with good experimental results, with speedups up to a factor of two or three. Their work is interesting because it can be applied to other designs of the Extended Andorra Model as well.

AKL differs from Prolog, both in syntax and semantics. Automatic translators that transform Prolog programs into AKL have been proposed [16]. However, this work was not sufficient to attract interest to AKL. Development on the language has since been discontinued. However, many of the ideas explored in the AKL have been used in the development of Oz, a concurrent constraint language.

3.5 Oz

The AKL group has since abandoned this language contributed the multiparadigm language Oz [127], including the Mozart distributed systems [145]. The Oz language provides the logical variable and encapsulated search, but thread programming and search is quite explicit, thus leading to a very different approach. The Oz language, instead of Horn clause syntax, has a fully compositional, higher-order syntax. For example, the deterministic append predicate can be written as follows in Oz:

```
proc {Append Xs Ys Zs}
   case Xs
   of nil then Ys=Zs
   [] X|Xr then Zr in
      Zs=X|Zr {Append Xr Ys Zr}
   end
end
```

This syntax follows a very different approach from the Prolog language. In fact, this code looks like functional programming [66] with dynamic typing [59].

The Oz abstract machine was inspired in the AKL work. The basic abstractions are names, logical variables, procedural abstraction and cells. Constraints are built over logical variables by first-order logic equations, using a set of predefined predicates. Cells are primitive entities that maintain state and provide atomic read-write operations.

The Oz abstract machine executes programs made up of threads. Concurrency in the initial Oz language, Oz 1, is implicit and based on lazy thread creation. When a statement suspends, a new thread is created that contains only the suspended statement. The main thread is not suspended and continues with the next statement. The successors of Oz 1, namely Oz 2 and Oz 3, abandoned implicit concurrency in favor of an explicit thread creation construct.

Oz procedures have encapsulated state so that objects can be defined directly as sets of methods (procedures) acting over their state (represented as a set of logical variables). The basic message passing mechanism in Oz is the *port*. Ports hold queues of incoming messages that are explicitly manipulated by interested processes.

Oz is a multiparadigm language designed to support different programming paradigms. Logic programming is one of its major paradigms, but not the only one. Oz combines in a single system the functional, object-oriented and constraint logic programming. On the other hand, programming in Oz is very different from programming in logic, which is the main objective of our work.

3.6 Extended Dynamic Dependent And-Parallelism

In producer-consumer and-parallelism systems, such as the DASWAM [132, 133] or ACE [100, 98, 53], two subgoals that share a dependent variable are executed in parallel independently from each other until one of them accesses that variable. If the variable is unbound and if the goal that accessed the variable is a consumer goal, then it suspends immediately. The consumer is later woken when the producer goal binds the variable to a non-variable value.

Gupta and Pontelli developed an interesting extension of dependent and-parallelism that provides some of the functionality of the EAM through parallelism [55, 98]. This model has been developed as an extension to the ACE system [53, 100, 52]. The insight is that goals that share variables may execute in parallel and mutually constrain their search-space. In EDDAP, a non-leftmost consumer subgoal for the `X` variable is allowed to bind `X` if the binding is deterministic.

The authors have been able to achieve good results with real life applications, such as natural language processing and expert systems, where and-parallelism alone is not sufficient. Their work is quite important in that it shows how the EAM ideas can be quite important for parallel logic programming systems. Note that EDDAP does require parallelism in order to constraint search space. Thus, in their case, the search space actually depends on how much parallelism is exploited.

3.7 Final Remarks

It is interesting to contrast the Basic Andorra Model, the Extended Andorra Model and the Andorra Kernel Language.

The EAM extends the BAM by allowing non-deterministic goals to execute in and-parallel, as long as they do not bind external variables. This has several advantages, consider the simple example:

```
belongs(El,S1,S2,S1) :- in(El,S1).
belongs(El,S1,S2,S2) :- in(El,S2).
```

Andorra-I [120] would consider `belongs/4` non-deterministic when executing the query

`?- belongs(2,[3,2],[3,4],S).` With the EAM, `belongs/4` may execute deterministically. The EAM will first create an or-box for the two clauses. Assuming that `in/2` executes deterministically and that the two sets `S1` and `S2` do not intersect, the EAM will only succeed for a box, and the goal `belongs/4` will be considered deterministic. This in turn allows the constraint on the fourth argument to be exported earlier.

The Andorra Kernel Language is a different evolution of the Extended Andorra Model. The AKL stems from the original EAM work, and thus shares a common set of rewrite rules that are applied to an And-Or Tree representing a program execution. On the other hand, the two languages differ in the language paradigm. The original EAM supports traditional logic programs, seen as a collection of Horn clauses. AKL provides a new programming paradigm where control is embedded in the fabric of the language. Each clause includes a guard and a body. The body can only execute when the guard is entailed by the current environment. Goals in the body or in the guard can execute concurrently. This contrast with David H. D. Warren's approach where control is optional: the system must work even if it has minimal knowledge on the program that is being generated. The control decisions on the EAM are based mostly on information implicitly extracted from the program. One would expect an EAM implementation to be more complex, and maybe less efficient, than an AKL implementation. On the other hand, the EAM provides for more flexibility: we can try pure logic programs, or we can exploit several different types of control strategies, depending on performance. In the next chapter we discuss the BEAM, a novel system that aims at exploiting the flexibility of the EAM with implicit control as a novel tool for logic programming.

Chapter 4

The BEAM Model

David H. D. Warren presented the main issues to be addressed in the EAM model, but he never presented a full specification, namely including the issues required to obtain an efficient implementation. Therefore several problems were left open, thus giving implementers the freedom to try different approaches. As we have seen, Gupta created the first interpreter for the EAM that served as a proof of concept for the EAM model. However, his implementation was not intended to be a system for general usage. An alternative approach was followed at SICS in the AKL work. AKL has explicit control built in the language, thus contrasting with the initial EAM model proposal.

In this chapter, we present a new model, the BEAM model (Basic design for the EAM), that fully specifies the Extended Andorra Model towards an efficient implementation. The BEAM model implements the main functionalities of the EAM with Implicit Control in the context of Horn Clauses. We start by detailing the main BEAM rewrite rules that manipulate the And-Or Tree representing a program's execution. Then, we describe further rules that allow the simplification and optimization of the And-Or Tree. Last, we describe the control model that is used during the execution of Prolog programs in order to decide when and which rules are to be applied.

4.1 BEAM Concepts

A BEAM *computation* is a series of rewriting operations, performed on an And-Or Tree. A BEAM *And-Or Tree* contains two kinds of nodes: *and-boxes*, that represent a conjunction of positive literals, and *or-boxes*, that represent alternative clauses for a selected literal.

A *configuration* is a And-Or Tree, describing a state of the computation. A *computation* is a sequence of configurations obtained by successive applications of rewrite rules that define valid state transitions.

The *initial configuration* is an and-box containing the initial sequence of atomic goals that represent the query and the empty set of constraints. The constraint of a final configuration is called an answer. An *answer* describes a set of assignments for variables for which the initial configuration holds.

An atomic goal is said to be *deterministic* when there is at most one candidate that succeeds for the goal. Otherwise it is said to be *non-deterministic.*

For the reminder of this thesis we define an And-Or Tree as *compact* when all children of and-boxes are or-boxes and when all children of or-boxes are and-boxes. A variable is said to be *external* to an and-box when not defined in the current and-box, and *local* otherwise. Moreover, a box is said to be *suspended* if the computation on it can not progress deterministically and the box is currently waiting for an event that will allow it to resume computation.

4.2 Rewrite Rules

In this section we present the main operations implemented in BEAM to manipulate the And-Or Tree. Our set of rewrite rules are based on the rules initially proposed by David Warren (please see section 3.2.1). Unlike in the original EAM, the BEAM does not support choice-boxes. Instead, the BEAM rewrite rules were adapted to support what we believe is a more flexible implementation of pruning as we discuss later.

The BEAM rewrite rules are:

1. **Reduction** (figure 4.1): in the BEAM, reduction proceeds by resolving the goal G against the heads of all clauses defining the procedure for G.

$$G \rightarrow \{[\exists Y_1 : \sigma_1 \ \& \ C_1] \vee \ldots \vee [\exists Y_n : \sigma_n \ \& \ C_n]\}$$

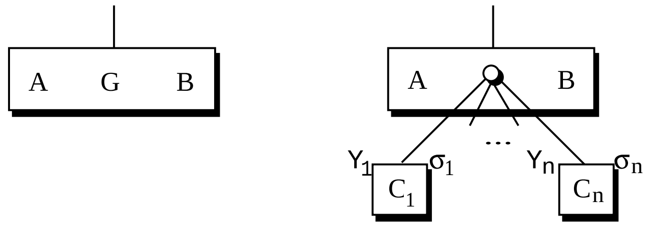

Figure 4.1: BEAM reduction rule.

Similar to the EAM original reduction rule, given a goal G, reduction replaces G by a new or-box with several alternatives to be explored.

2. **Promotion**: an and-box that is a single alternative to an or-box can promote its variables and its bindings to the parent and-box (see figure 4.2):

   (a) $[\exists X : \sigma \ \& \ A \ \& \ \{[\exists Y : \theta \ \& \ W]\} \ \& \ B]\} \rightarrow [\exists X, Y : \sigma\theta \ \& \ A \ \& \ \{[W]\} \ \& \ B]$

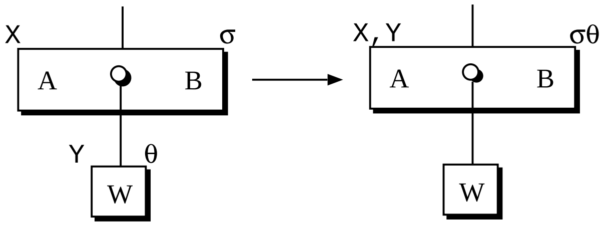

Figure 4.2: BEAM promotion rule.

Note that this rule is similar to the EAM promotion rule as it propagates results from a local computation to the level above. However, it differs from the original rule in that the two and-boxes are not merged. The BEAM does not perform merging because the and-box to be promoted may include pruning operators and thus lead to an incorrect state.

3. **Propagation**: this rule allows us to propagate the constraints from the node where the variables were created to the subtree below. This rule is thus symmetrical to the promotion rule (see figure 4.3).

$$[\exists X, Z : Z = \sigma(X) \ \& \ldots \& \ \{\ldots \vee [\exists Y : G] \vee \ldots\} \ \& \ldots] \rightarrow$$

$$[\exists X, Z : Z = \sigma(X) \ \& \ldots \& \ \{\ldots \vee [\exists Y : Z = \sigma(X) \ \& \ G] \vee \ldots\} \ \& \ldots]$$

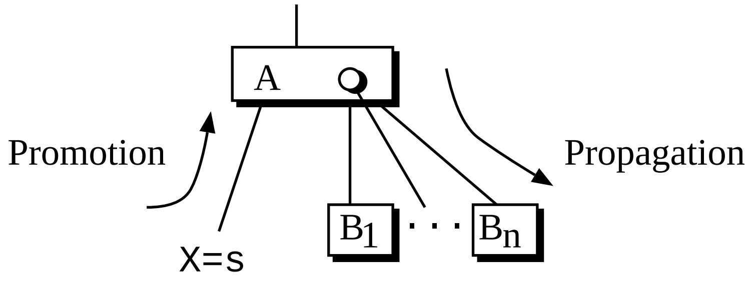

Figure 4.3: BEAM propagation rule.

The propagation rule is expensive and therefore only applied on demand from nodes that consume a binding for the variable. Whenever a variable is bound, all nodes suspended on the variable are signaled to indicate that they can now have the binding propagated to their own box.

4. **Splitting** (non-determinate promotion): splitting distributes a conjunction across a disjunction, in a way similar to the forking rule of the original EAM.

$$[\exists X : \sigma \ \& \ A \ \& \ \{C_1 \vee \ldots \vee [\exists Y : \theta \ \& \ C_i] \vee \ldots \vee C_n\} \ \& \ B] \rightarrow$$

$$\{[\exists X : \sigma \ \& \ A \ \& \ \{[\exists Y : \theta \ \& \ C_i]\} \ \& \ B] \ \vee$$
$$[\exists X : \sigma \ \& \ A \ \& \ \{C_1 \vee \ldots \vee C_{i-1} \vee C_{i+1} \vee \ldots \vee C_n\} \ \& \ B]\}$$

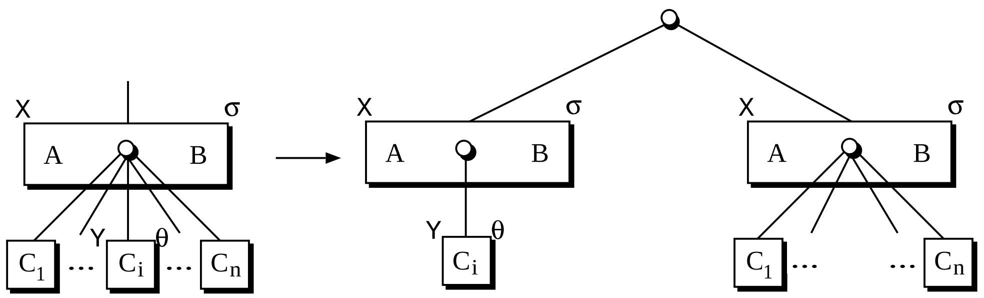

Figure 4.4: BEAM splitting rule.

We have changed Warren's original forking rule in order to improve support for pruning. The BEAM splitting rule does not merge the split and-box with the parent and-box (represented as C_i in figure 4.4). The merge will be performed later by an application of the promotion rule to the and-box C_i.

4.3 Correctness and Completeness of the BEAM Rules

We next give a short argument on the correctness of these rules for the Herbrand domain. Given an initial program P and query $\mathcal{T}_0$ an EAM sequential computation is defined as a sequence of rule applications that generates the trees $\mathcal{T}_1, \ldots, \mathcal{T}_i, \mathcal{T}_{i+1}, \ldots$. To use induction, we need to show that:

$$P, \mathcal{T}_i \vdash \mathcal{T}_{i+1} \Longrightarrow P, \mathcal{T}_i \models \mathcal{T}_{i+1}$$

Assume P and $\mathcal{T}_i$ hold. If $\mathcal{T}_{i+1}$ was created by reduction, $\mathcal{T}_{i+1}$ holds because reduction is a direct application of the resolution rule. If $\mathcal{T}_{i+1}$ was created by promotion, $\mathcal{T}_{i+1}$ holds because conjunction is transitive, associative, and commutative and because all quantifiers are existential and all literals positive. If $\mathcal{T}_{i+1}$ was created by propagation, $\mathcal{T}_{i+1}$ holds because $\sigma(W)$ must always be a valid constraint. If $\mathcal{T}_{i+1}$ was created by splitting, $\mathcal{T}_{i+1}$ holds because we can distribute disjunctions over conjunctions.

The four rules just presented can be used to implement refutation complete strategies. In fact, as we show next, a standard resolution step can be described as a sequence of reduction, splitting, promotion, and finally substitution steps until we return to clausal form.

Consider $:-\ G_1, \ldots, G_n.$ as the conjunction of subgoals to be resolved. Let us assume that the literal G_i is selected for a resolution step. Thus, the program is searched for a set of clauses (that may be empty) whose head unifies with G_i. Consider that such clauses exist, and they are:

$A_1\ :-\ B_{11}, \ldots, B_{1k_1}.$
$\ldots$
$A_m\ :-\ B_{m1}, \ldots, B_{mk_m}.$

Consider also that the unification process has determined the substitutions σ_j to the variables of A_j and G_i, such that $A_j\sigma_j = G_i\sigma_j$ for $j = 1, \ldots, m$.

$:-\ G_1, \ldots, G_n.$ is equivalent to $\sim G_1 \vee \ldots \vee \sim G_n$ and can also be written as $\sim (G_1 \wedge \ldots \wedge G_n)$. We will use this last form to apply the EAM rules. Also for simplicity and coherence with our previous notation, we use {} and [] to represent or-boxes and and-boxes respectively.

Let us start with the and-box: $\sim [G_1 \wedge \ldots \wedge G_n]$.

1st Step: *Reduction* on G_i

$\sim [G_1 \wedge \ldots \wedge G_{i-1} \wedge \{[\sigma_1 \wedge B_{11} \wedge \ldots \wedge B_{1k_1}] \vee \ldots \vee [\sigma_m \wedge B_{m1} \wedge \ldots \wedge B_{mk_1}]\} \wedge G_{i+1} \wedge \ldots \wedge G_n]$

2st Step: *Splitting*

$\sim \{[G_1 \wedge \ldots \wedge G_{i-1} \wedge \{[\sigma_1 \wedge B_{11} \wedge \ldots \wedge B_{1k_1}]\} \wedge G_{i+1} \wedge \ldots \wedge G_n] \vee$
$[G_1 \wedge \ldots \wedge G_{i-1} \wedge \{[\sigma_2 \wedge B_{21} \wedge \ldots \wedge B_{2k_2}] \vee \ldots \vee [\sigma_m \wedge B_{m1} \wedge \ldots \wedge B_{mk_m}]\} \wedge G_{i+1} \wedge \ldots \wedge G_n]\}$

that is the same as having:

$\sim [G_1 \wedge \ldots \wedge G_{i-1} \wedge \{[\sigma_1 \wedge B_{11} \wedge \ldots \wedge B_{1k_1}]\} \wedge G_{i+1} \wedge \ldots \wedge G_n] \wedge$
$\sim [G_1 \wedge \ldots \wedge G_{i-1} \wedge \{[\sigma_2 \wedge B_{21} \wedge \ldots \wedge B_{2k_2}] \vee \ldots \vee [\sigma_m \wedge B_{m1} \wedge \ldots \wedge B_{mk_m}]\} \wedge G_{i+1} \wedge \ldots \wedge G_n]$

For this step forward, we will only proceed with the first alternative (first line). The other alternatives can be reduced similarly through $m-2$ applications of the splitting rule. Thus we will only continue the simplification of :

$\sim [G_1 \wedge \ldots \wedge G_{i-1} \wedge \{[\sigma_1 \wedge B_{11} \wedge \ldots \wedge B_{1k_1}]\} \wedge G_{i+1} \wedge \ldots \wedge G_n]$

that is, we will only explore the first alternative for G_i.

3st Step: *Promotion*

$\sim [G_1 \wedge \ldots \wedge G_{i-1} \wedge (\sigma_1 \wedge B_{11} \wedge \ldots \wedge B_{1k}) \wedge G_{i+1} \wedge \ldots \wedge G_n]$

After promotion, we may now continue by applying σ_1 to the variables of the other subgoals in the conjunction.

4st Step: *Propagation*

$\sim [\sigma_1 \wedge G_1 \wedge \ldots \wedge G_{i-1} \wedge B_{11} \wedge \ldots \wedge B_{1k} \wedge G_{i+1} \wedge \ldots \wedge G_n]$

We can now return to the clausal form:

$:- \, (G_1, \ldots, G_{i-1}, B_{11}, \ldots, B_{1k}, G_{i+1}, \ldots, G_n)\sigma_1$

This result is thus equivalent to replacing G_i with the body subgoals of the first valid alternative for G_i, being σ_1 the substitution that validates $A_1\sigma_1 = G_i\sigma_1$.

The EAM computational rules can thus implement any resolution based mechanism, and therefore inherits resolution's refutation completeness. In fact, the EAM computation rules may be regarded as a fine grained description of logic program execution that allows one to separate reduction (creation of new goals) from binding propagation. our interest in these rules is to take advantage of the extra flexibility provided by their finer granularity as a vehicle for providing new execution strategies.

4.4 Simplification and Optimization Rules

The previous rules give the main principles for the EAM. An actual implementation must be able to simplify the And-Or Tree in order to propagate success and failure and to recover space by discarding boxes. Our kernel design for the EAM, the BEAM, therefore includes additional simplification rules that generate compact configurations. Extra rules further allow us to optimize the computation process by early pruning of branches, resulting in early detection of failure. These are:

1. **Success Propagation:** The original EAM does not explicitly provide a notion of successful computation that has completed execution and that may now be discarded. The following rules identify success situations in the BEAM and allow the propagation of success towards the upper boxes. To implement these rules we use the notion of a *true-box*: an and-box is called *true-box* when all the local computations have been completed and the and-box does not contain constraints on external variables:

$$[\exists X :] \equiv \texttt{true}$$

   Note that the and-box might have had constraints on external variables, but those constraints would have left the and-box after application of the promotion rule. We thus say that a success occurs when we found a true-box. The possible situations are:

   (a) **true-in-or simplification:**

$$\{\texttt{true}\} \rightarrow \texttt{true}$$

If an or-box contains a unique alternative which succeeds, the or-box has succeeded. True-boxes may be used to achieve implicit pruning, as discussed in section 5.3.

(b) **true-in-and simplification**

$$[\exists X : \sigma \ \& \ G_1 \ \& \ldots \& \ G_{i-1} \ \& \ \texttt{true} \ \& \ G_{i+1} \ldots \& \ G_n \ ] \rightarrow$$

$$[\exists X : \sigma \ \& \ G_1 \ \& \ldots \& \ G_{i-1} \ \& \ G_{i+1} \ \& \ldots \& \ G_n \ ]$$

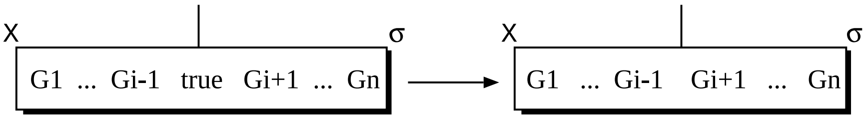

Figure 4.5: BEAM true-in-and simplification.

An occurrence of a true-box inside another and-box can be removed since the true-box is not adding any knowledge (see figure 4.5).

The ultimate goal of a BEAM computation is to reduce all and-boxes to true-boxes so that the initial box can be itself simplified.

2. **Failure Propagation:** The symmetric problem to the propagation of success is the propagation of failure. It is quite important to identify failing computations in order to allow the propagation of failure towards the upper boxes and to recover space. Again, the basic EAM design does not contain explicit rules for propagating failure. First, we define a *fail-box* as an empty or-box:

$$\{\} \equiv \texttt{false}$$

We thus say that a failure occurs when we found a fail-box. The possible situations are:

(a) **false-in-unique-or simplification:**

$$\{\texttt{false}\} \rightarrow \texttt{false}$$

If an or-box contains an unique alternative which fails, the or-box has failed.

(b) **false-in-or-simplification:**

$$\{C_1\vee\ldots\vee C_{i-1}\vee \texttt{false}\vee C_{i+1}\vee\ldots\vee C_n\}\rightarrow\{C_1\vee\ldots\vee C_{i-1}\vee C_{i+1}\vee\ldots\vee C_n\}$$

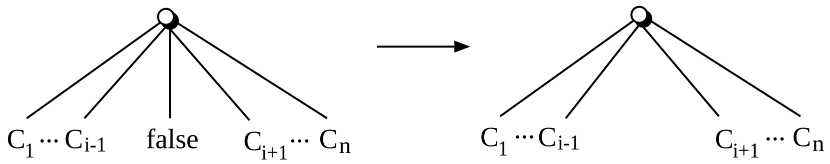

Figure 4.6: BEAM false in or simplification.

If one alternative to an or-box fails, the alternative can be removed from the or-box (see figure 4.6).

(c) **false-in-and-simplification:**

$$[\texttt{false}\ \&\ G_i\ \&\ \ldots\&\ G_n]\rightarrow\texttt{false}$$

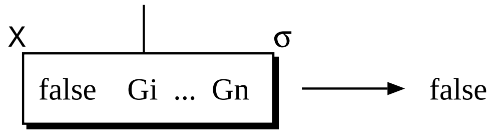

Figure 4.7: BEAM false in and simplification.

If the first goal of an and-box fails, then the and-box has failed (see figure 4.7). A more powerful version of this rule for simplification of and-boxes is presented in the context of our discussion on pruning in section 5.3.

Failure simplification rules have priority over all the other rules to allow propagation of failure as fast as possible.

3. **And-Compression:**

$$[\exists X:\sigma\ \&\ A\ \&\ \{[B]\}\ \&\ C]\rightarrow[\exists X:\sigma\ \&\ A\ \&\ B\ \&\ C]$$

This rule removes a nesting of two and-boxes by promoting the inner and-box to the outer and-box (see figure 4.8). It thus complements the promotion rule discussed previously. The BEAM cannot apply this rule if there is a

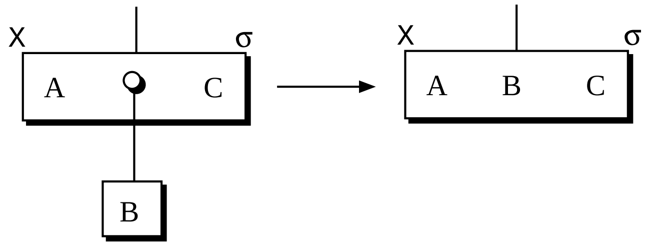

Figure 4.8: BEAM and-compression rule.

pruning operator such as `cut` in the inner box (see section 5.4 for more detail on these cases). Doing and-compression has several advantages. First, we can immediately recover memory. Second, as the compressed tree is smaller, it is simpler to find work. Last, there is less information to duplicate when performing splitting.

4. **Deterministic-reduce-and-promote:**

   $[\exists X : \sigma \ \& \ A \ \& \ B \ \& \ C] \rightarrow [\exists X, Y : \sigma\theta \ \& \ A \ \& \ G_1 \ \& \ldots \& \ G_n \ \& \ C]$

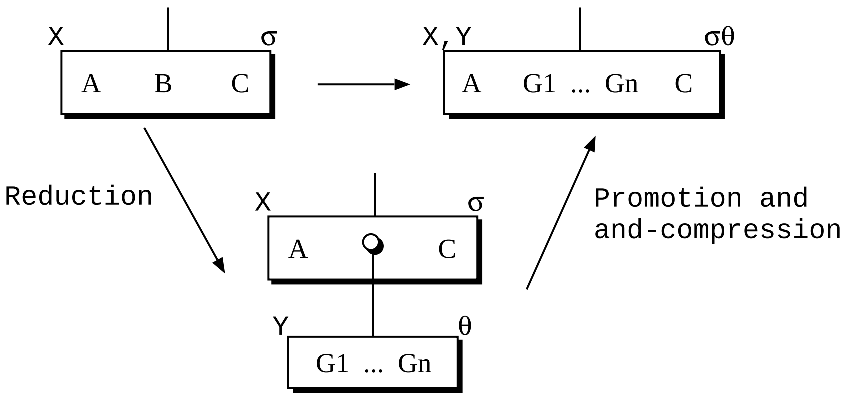

Figure 4.9: BEAM deterministic-reduce-and-promote rule.

Whenever a deterministic goal B is to be reduced to a single alternative with goals $G_1, \ldots, G_n$, the reduction can be applied directly in the home and-box, thus avoiding creating two nested and-boxes, where one would be immediately promoted (see figure 4.9). Similar to the *and-compression* rule, the BEAM cannot apply this rule if there is a pruning operator such as `cut` in the inner box (see section 5.4 for more detail in these cases).

This rule is quite useful in simple recursive programs because it allows reusing the space of the original and-box for the inner and-box. It may be seen as a generalization of tail-recursion.

4.5 Control for the BEAM

The previous rewrite and simplification rules provide the basic machinery for the correct execution of logic programs. To this machinery, we must add control strategies that decide which step to make and when. The BEAM with implicit control uses a default strategy according to David Warren's original design:

- Reduction applies to goals whenever:
  1. a goal belongs to an and-box that is a single alternative to its parent or-box (deterministic computation), or,
  2. the goals on and-box that do not hold constraints on external variables.
- Promotion is always allowed.
- Propagation of constraints from an and-box is allowed to all the descendants and-boxes that have constraints on the variable being propagated.
- Simplification rules are always allowed.
- Splitting is allowed only if no other rules apply.

Describing when each rule is allowed to execute does not completely define the BEAM control strategy. It is also necessary to define the priority for each rule so that in the cases when several rules are allowed the system can decide which rule to apply. The BEAM default control scheme is illustrated in figure 4.10.

The default scheme of the execution control in BEAM favors rules that reduce the search space: the simplification rules, the promotion rule and the propagation rule. All these rules are deterministic. When used they either allow one to reduce the search space or compact the And-Or Tree. Note that the propagation rule is always allowed after a successful execution of a promotion rule. This is very important since the promoted variables need to be propagated to the lower subtree so that boxes

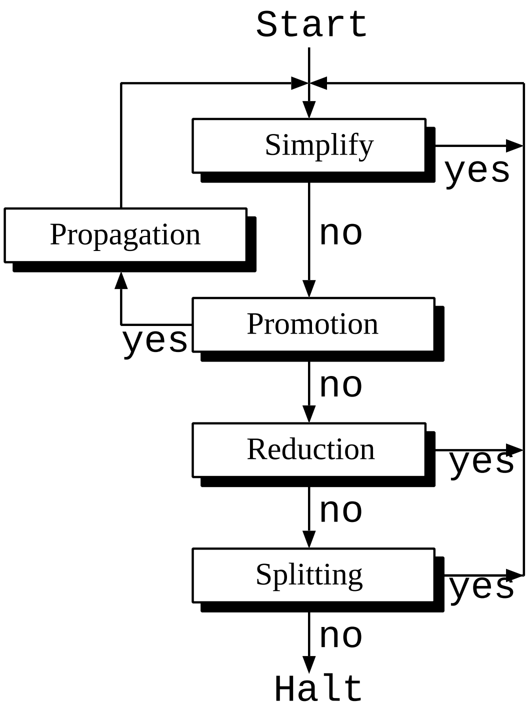

Figure 4.10: Execution control in the BEAM.

that were suspended in those variables may fail if the constraints are incompatible or continue execution otherwise.

The remaining two rules, reduction and splitting, expand work. The reduction rule is deterministic while the splitting rule is non-deterministic and can lead to duplication of work. For this reason it is reasonable to apply the splitting rule only when all the other operations cannot be applied further. Note that eventually the boxes will suspend when they constrain external variables (created above the current box), thus making the splitting rule the only applicable rule. The system will then apply the splitting rule so that one of the other rules are re-enabled.

Although this form of execution control is sufficient for running most programs, there are some important considerations on the propagation rule that we discuss next.

4.5.1 Propagation

Often after a successful application of the promotion rule, one needs to propagate the promoted bindings. For this reason propagation is always allowed after a promotion.

However, executing propagation to all subtrees below the box where the variable is local can be very expensive. On the BEAM we have implemented a different approach. Instead of propagating to all boxes immediately after binding on a variable, we first send a signal to all descendent and-boxes that have constraints on the variable in question. All and-boxes that received the signal become aware that they must perform a consistency check of the constraints generated in the and-box with their constraints composing their environment. We call this operation as `environment synchronization`.

The `environment synchronization` fails if one of the constraints imposed by the and-box is incompatible with the current value of the variable (in the case of Herbrand domain this corresponds to not unifying). The failure will be propagated to the and-box and the box will be marked as failed. Note that it is very important for the whole computation to have failures immediately propagated to allow memory recovery and to allow other branches to restart their computation. If the constraints imposed by the and-box are all compatible with the current values of the variables being constrained, then `environment synchronization` succeeds. However, the and-box will only be able to perform reduction steps or to succeed if it does not constrain other external variables.

Figure 4.11 presents an example that illustrates the importance of executing the propagation rule after the promotion rule and why binding propagation should be done lazily. In this example we show a situation where several boxes are suspended on the variables `X` and `Y` and where there is a single and-box d that is allowed to promote the binding `X=2` to its parent and-box. Performing the propagation immediately after the promotion helps to accomplish two main goals of the EAM:

- if compatible constraints are found, several and-boxes may be allowed to reduce, increasing the available work and thus increasing possible parallelism. In the example the and-box g is allowed to resume computation because it was suspended on the `X` variable and it has a compatible constrain.
- if incompatible constraints are found the tree will be simplified and possibly the search space will also be reduced. In the example the and-box f fails because it has an incompatible binding to `X`. This failure allows the and-box e to resume computation because, although it has a binding to an external variable `Y`, it now

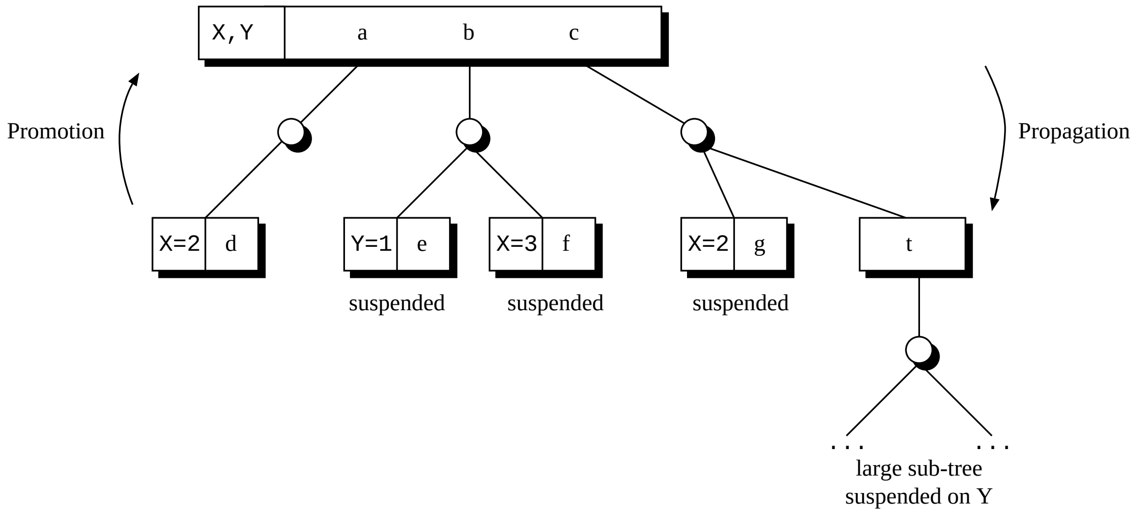

Figure 4.11: Propagation example.

become a unique alternative and promotion can be applied.

Doing propagation immediately after the promotion is very important but, one still has to be careful because doing propagation from the top and-box to all descendent and-boxes may be expensive and unnecessary. On the example, the propagation of `X=2` to the and-boxes e and t would be unnecessary since it will not change the computational state of the boxes. Moreover if we suppose that the and-box t is the root of a large tree then it would be a very expensive operation. For this reason, BEAM propagation only signals boxes that are suspended on the variable that has been constrained (`X` in the example), because only those boxes can benefit from the new information being available.

4.6 An Example of BEAM Execution

To better understand the essential ideas of how the computation and control of a logic program proceeds in the BEAM, we next present a very simple example:

```
ancestor(X,Z):- parent(X,Y), parent(Y,Z).

parent(david,john).    parent(mike,david).
```

Figures 4.12 to 4.16 show several execution steps for this program versus the query `?- ancestor(X,Z).` We remember the reader that or-boxes are represented as small circles. And-boxes are represented as rectangles divided in 3 areas. The left area is reserved for the local variables, the middle area is for the conjunction of goals, and the right one is used to record assignments to external variables.

As the first step (figure 4.12), the BEAM performs a deterministic reduce step and promotion on `ancestor(X,Z)` since `ancestor` only has an alternative.

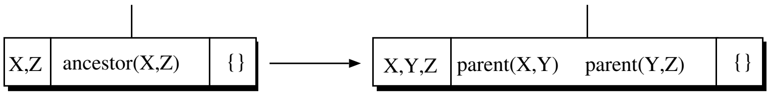

Figure 4.12: Reduction and promotion in a single step.

On the second step (figure 4.13), both `parent(X,Y)` and `parent(Y,Z)` are reduced. Each one has two alternatives. The first alternative of `parent(X,Y)` tries to perform `X=david` and `Y=john`, leading to suspension because `X` and `Y` are external variables. For a similar reason, the second alternative will also suspend when trying to bind `X=mike` and `Y=david`. Again, the two alternatives to `parent(Y,Z)` suspend when trying to assign values to `Y` and `Z`.

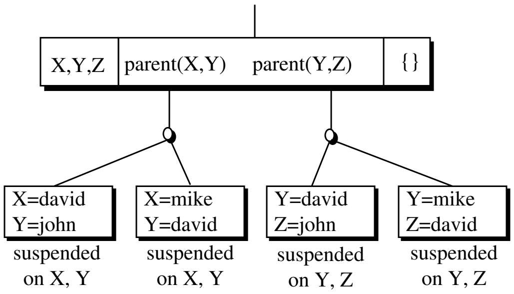

Figure 4.13: Reduction.

At this point, all and-boxes have suspended and splitting is the only rule that can be applied. There are two or-boxes candidates to split. By default, the BEAM chooses the or-box that is the leftmost on the tree. After splitting (figure 4.14) we have a new or-box as root of the tree, having two alternatives that were derived from the previous state. Both alternatives are similar except for the or-box where splitting occurred. The two and-boxes that were in the or-box were the splitting occurred are marked as awaken in order to continue execution. The promotion rule can be applied to both and-boxes since both are single alternatives in their parent or-box.

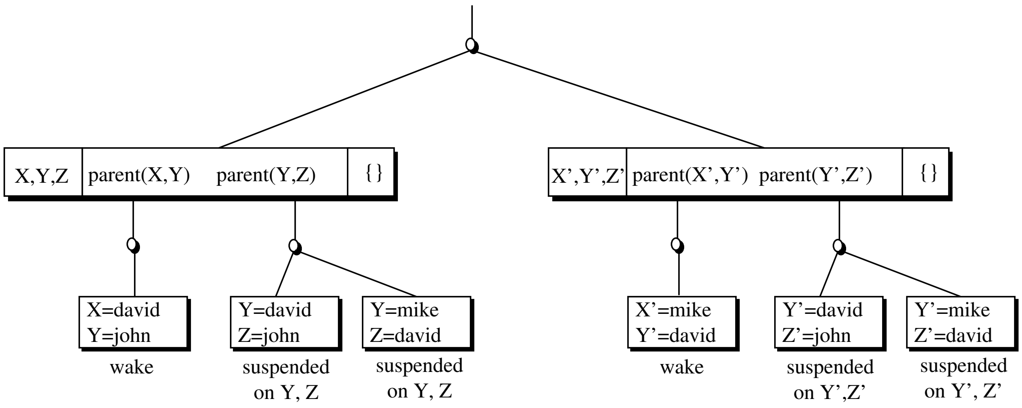

Figure 4.14: Splitting.

As a result of promotion (figure 4.15), we have that variables `X` and `Y` are now constrained. No box is suspended on `X`, hence no propagation is necessary for this variable. The `Y` variable has currently four boxes suspended and thus propagation is necessary to inform them that they must do *environment check*.

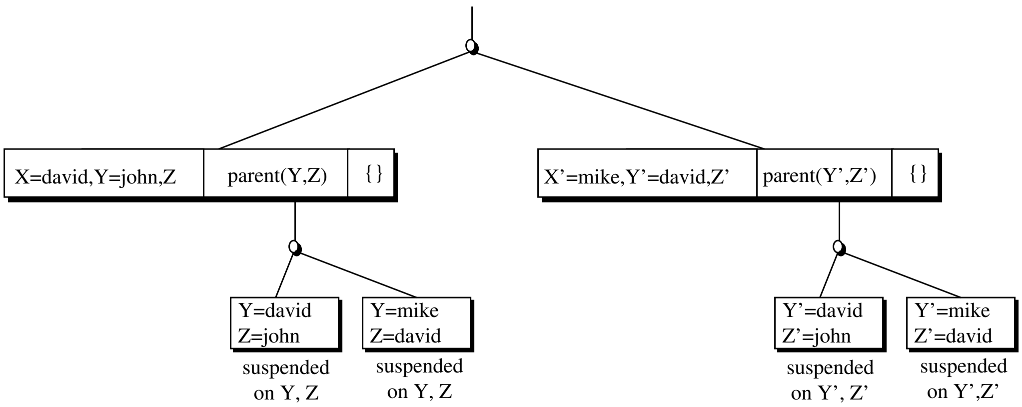

Figure 4.15: Promotion.

During the propagation only one of all suspended and-boxes does not fail (figure 4.16). After failure simplification, the tree is reduced to a single alternative with a deterministic solution, that is `X=mike` and `Z=david`.

In most cases the EAM should be able to significantly reduce the execution space for logic programs. Unfortunately, this is not always the case, as exemplified next.

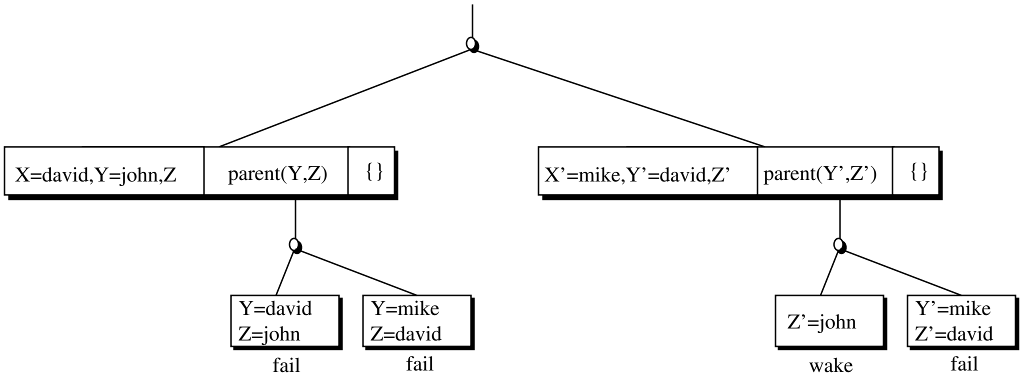

Figure 4.16: Propagation.

4.7 Non-Termination in the EAM

Favoring computations that do not bind external goals does have a drawback. Consider a left-recursive definition of the ancestor relationship presented in figure 4.17.

```
ancestor(X,Y):- parent(X,Y).
ancestor(X,Z):- parent(X,Y), ancestor(Y,Z).

parent(a,fa).     parent(a,ma).

?- ancestor(a,Z).
```

Figure 4.17: The ancestor example that leads to Non-Termination in EAM.

Consider the query `?- ancestor(a,Z).` with `Z` unbound.

Figure 4.17 shows how the EAM leads to non-termination in this example. To simplify the notation we use `a(X,Y)` for `ancestor(X,Y)` and `p(X,Y)` as `parent(X,Y)` in Figure 4.18 and in the following discussion.

When the query `a(a,Z)` is executed an or-box with two alternatives is created (see step 1 in figure 4.18). The first alternative, will call `p(a,Z)` that successfully unifies with two `parent` alternatives. Although, both alternatives suspend since both try to non-deterministically bind the external variable `Z`. The second `ancestor` alternative has two goals. The first goal calls `p(a,X1)`, where `X1` is a new variable local to the current alternative. Once again the `parent` call unifies with two alternatives that

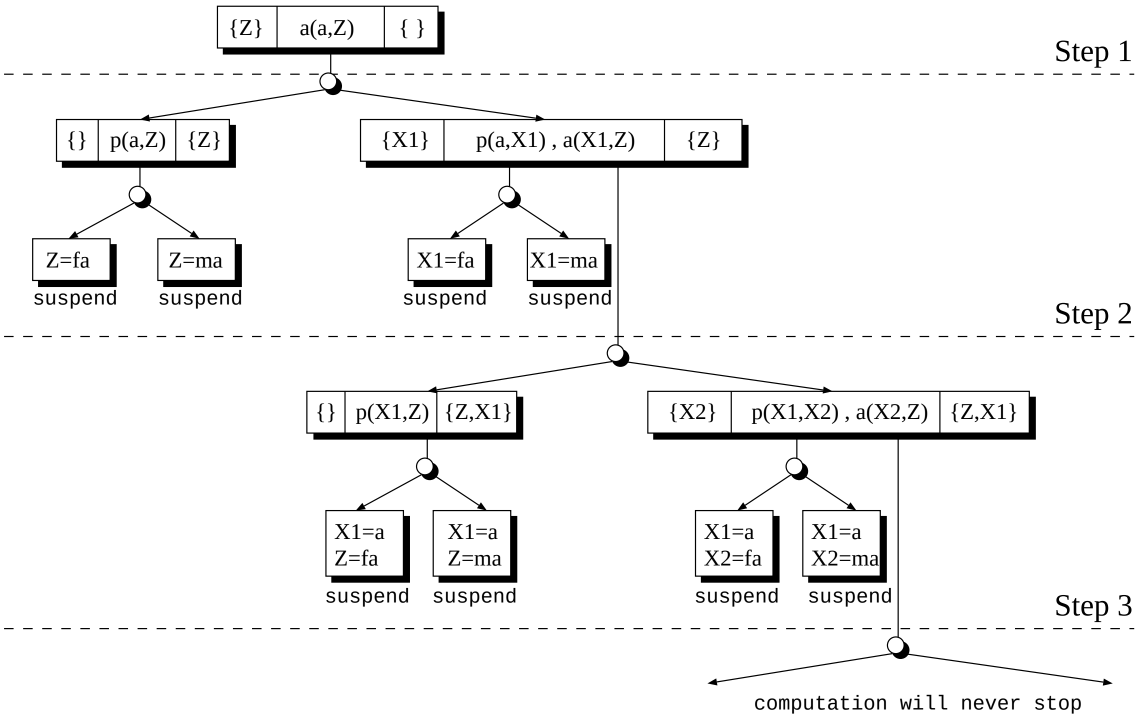

Figure 4.18: The ancestor execution that leads to non-termination in EAM.

end up suspending for trying to non-deterministically bind an external variable (`X1`). The second goal is a recursive call to the `ancestor` predicate with unbound variables as arguments. It successfully expands into two alternatives. This whole process is repeated and the promotion of the bindings is indefinitely delayed, thus leading the computation to non-termination since no value will ever be assigned.

Warren and Gupta proposed a solution to this problem by using Eager Non-Determinate promotion [56]. However, to use this solution, one must classify variables as *guessable*, *non-guessable* or *others* before execution. This annotation can be generated by means of compile-time analysis, or it can be supplied by the programmer explicitly. Unfortunately, their solution is not complete since in some problems it may be very difficult to mark a variable as guessable. It is also not ideal since one of the aims of EAM is to obtain maximum performance with minimum control information from the user (Implicit Control). Of course, there is no magic wand capable of always generating the best strategy for all programs: in some cases the user must still provide control. In the next chapter we discuss how to improve control in the BEAM.

4.8 Final Remarks

Agents Kernel Language and the Extended Andorra Model are two different approaches based on the Basic Andorra Model. We have already discussed the main differences on these languages on section 3.7. The BEAM is an approach based on David Warren's original design of the EAM where control is optional. The system must work even if it has minimal knowledge on the program that is being generated. In contrast, in AKL the control is intrinsic to the language, guards are the basic mechanisms of the language.

Other differences exists if we compare the basic design of both models. The original EAM used three different boxes for the And-Or Tree: and-box, or-box and choice-box. Unlike the original EAM, AKL only implements and-boxes and choice-boxes. It does not implement or-boxes since all clauses should have a guard operator. On the other hand, the BEAM only uses and-boxes and or-boxes. Thus BEAM implements pruning without the overhead of the extra data-structure choice-boxes required in EAM. Both models share a set of similar rewrite rules, based on Warren's original EAM. The major difference is in the simplification rules. Whereas the BEAM releases memory eagerly (see for example the *Success Propagation* rule), AGENTS implementation of AKL releases boxes lazily. The cost for AGENTS is that while choosing rules it will have to walk through useless boxes.

BEAM, similarly to AKL and in contrast to Gupta's EAM interpreter, does not implements the *substitution* rule. Instead a lazy *propagation* of the bindings was proposed. A new operation called `environment synchronization` is performed only on those boxes that can benefit from the information being available. Another important remark about the BEAM is that we do not implement the *global forking* rule as proposed in Gupta's interpreter. The reason is that we wanted to mimic the Prolog execution as close as possible. Only one of the suspended branches is chosen to split, usually the leftmost branch. Eventually, global forking is fully performed. As we present in the next chapter, we try to delay splitting on the or-branches as much as possible until either they fail or they become the leftmost suspended box in the tree.

Chapter 5

Controlling Search

In the previous chapter we have presented the basic ideas of our kernel implementation of the EAM with implicit control, the BEAM. In practice, the user may need to introduce control himself, say, to obtain extra parallelism, to avoid looping, or to prune unnecessary search space. In this chapter we extend the BEAM towards the LIGHT-BEAM (Leftmost Innermost Goal Hopping Tabulation - BEAM). The LIGHT-BEAM builds on the BEAM to support more flexible search, implicit and explicit pruning of the search space. We also discuss a mechanism for guaranteed termination on computations that use depth-limited terms.

5.1 Improving Search: When to Suspend

The original EAM proposal states that a box should immediately suspend when trying to bind an external variable non-deterministically. A first problem that we would like to address is the meaning of *deterministic* for the BEAM, that is, when the BEAM recognizes that a single clause matches.

First, we should observe that it is usual to see Prolog programmers creating alternative clauses that can become deterministic during the head unification or during the execution of builtins on the body of the clauses. As an example, consider the following excerpt from two well-known Prolog programs: `ancestor` and `quick-sort`.

```
parent(john, richard).     partition([X|L],Y,[X|L1],L2) :- X =< Y,
parent(john, mary).                  partition(L,Y,L1,L2).
parent(patrick, paul).     partition([X|L],Y,L1,[X|L2]) :- X > Y,
parent(patrick, susan).              partition(L,Y,L1,L2).
                           partition([],_,[],[]).
```

First, consider the query `?- parent(X,mary)`. Although a single alternative succeeds for this query, the EAM might not able to deterministically determine the solution because all four alternatives for `parent` first try to bind a value to `X` an thus force the box to suspend. The same problem may happen with the query `?- partition([4,3,5],2,A,B)`. This call is deterministic but the EAM would suspend when unifying `[X|L1]` to `A`.

To deal with these situations we propose a rule to define when to suspend in the BEAM:

> The test for suspension should only occur after the unification of all arguments in the head of the clause and always before the end of the clause code or before the first call to a goal that is not a test builtin (flat guard). An and-box should only suspend if unification of the head arguments have generated one or more bindings to external variables.

Using our rule the queries above would execute deterministically without requiring suspension and splitting. A second advantage is that our scheme is simpler to implement since the suspension may only occur at a fixed point of the code thus simplifying the number of tests necessary to determine if the box should or not suspend.

Our approach contrasts with AKL's [68]. AKL uses information explicitly inserted in the program to decide when to split. In the `partition` predicate for example, the AKL language requires the user to insert a guard operator after the test comparing `X` and `Y` for attaining determinism. If the user does not insert the guard operator or if inserts the guard operator before the test condition, then partition would become non-deterministic in AKL resulting in a severe performance penalty when executing this predicate.

Andorra-I is able to detect determinism in the two examples presented above by using determinacy detection algorithm in the preprocessor [122]. This algorithm generates at compile-time a group of indexing instructions based on a decision graph to achieve determinacy. The decision graph may use any of head arguments plus the flat guards of the clause to detect determinacy. However the Andorra-I scheme still has some limitations. Consider the query `?- parent(X,X), lots_of_deterministic_work`. This query clearly fails for the example presented. Unfortunately, the Andorra-I preprocessor cannot generate sufficient information to classify the `parent` call as deterministic. Thus Andorra-I would delay execution of the goal, believing it to be non-deterministic, and would enter the second goal. The BEAM, on the other hand, would not. The BEAM would not classify the query as deterministic beforehand, but during the unification of the head arguments for the `parent` call, all alternatives would fail and thus the query would also immediately fail .

5.2 Improving Search: When to Split

The Andorra Principle states when to execute a goal *early*. The symmetrical question is *when* to execute a goal *late*. In the our rules for the EAM, to execute a goal late is to execute *splitting*. By default, splitting is performed when no other rule is available.

Our default control scheme for the BEAM chooses the leftmost suspended goal to apply the splitting rule. Thus, the default scheme tries to mimic the left-right behavior of Prolog. Notice that if a box was suspended we must at least have one binding to a variable goal local to the box before we execute the splitting rule. After applying the splitting rule, this binding should be promoted in the newly created and-box and eventually propagated to other boxes.

Our default control scheme is often sufficient if the goal is to reduce the search space. On the other hand, the approach is very rigid and does not allow us to take best advantage of all the BEAM features. One possibility is to allow dynamic selection of boxes to do splitting as in Pandora [10, 11]. However, to improve concurrency and parallelism, further mechanisms should be used.

We have considered three major extensions to the default rule:

- *Declare predicates as producers* and allow them to perform *Eager-Forking*, that is, to do splitting as soon as they are called. A similar idea was first proposed by

Gupta and Warren [56], where it applied to variables not goals. Intuitively, we declare goals to be producers if we expect them to produce, but not consume, bindings from other goals.

Eager-forking is dangerous, because producers may in fact benefit from bindings generated by other goals. Nevertheless, it is also attractive, because it is simple to understand, can increase parallelism, and in a few cases it actually improve the search space. We have implemented eager-forking on the BEAM and we discuss it in more detail on section 5.5.1.

- *Define the scope for non-deterministic promotions.* Instead of doing splitting in the whole tree, the user specifies a boundary up to which we should check for splitting. The idea has appeared in many guises: mini-scopes in Gupta and Warren [56], independent computations in Bueno and Hermenegildo [16]. Scoping can also be obtained dynamically through the use of *stability*, that we discuss in some detail next. Scoping can reduce the search space and is required to support independent parallelism.

- *Use the sequential conjunction.* The right-hand side of a *sequential conjunction* can only be evaluated after the left-and-side has succeeded. This has two major advantages. We can delay goals that we are sure will never constrain search-space, thus reducing the overheads in performing splitting. We can use the sequential conjunction in order to guarantee correct ordering for side-effects builtins such as `read/1` or `write/1` [122].

5.2.1 Stability

An aggressive approach to decide when to apply the splitting rule is to use *stability*, as in AKL [67, 45]. Stability provides a dynamic notion of scoping. An and-box G which is a subgoal of another and-box G' is *stable* relative to G' if:

1. no deterministic operation is applicable to or within G, and

2. no series of rewrites to different parts of the configuration (And-Or Tree) may allow a deterministic rule to be applied to G.

If a box G is stable, then we can split immediately on it because no other goal in G' will further constrain G. Forking early on stable boxes clearly improves concurrency and parallelism, although it does not necessarily improve the search space.

The main problem with stability is that detecting whether an and-box is stable is expensive. In general, to prove that a goal G is stable we need to know which other goals share variables with G, and this problem is known to be NP-Complete [39]. There has been work on compile-time analysis to recognize stability, but most solutions rely on user control.

AKL approximates stability by marking and-boxes when suspending on external variables. These marks are never cleared, so some stable boxes may not be detected. On the BEAM we have adopted a different approach for detecting stability. Our approach is less conservative while it introduces only a small overhead to the system. Our approach requires and-boxes to have two fields that, for simplicity, we have designated as *depth* and *stability*. The *depth* field is an integer recording the depth level of the box in the And-Or Tree, and is also useful for other purposes (please refer to section 6.2). The *Stability* field is an integer recording the depth level of the nearest possible stable box for the current box (that may include itself). A box `B` is said to be stable if **(i)** no other rules besides splitting apply to `B` or its subtree; and **(ii)** no other worker is working in the box `B` or its subtree; and **(iii)** if the box *stability* value is equal to its *depth* value. These two fields are easily maintained:

- All boxes are initialized with *depth* equal to the parent's value plus one. The *stability* value is initialized as being equal to the *depth* value. Note that if splitting is the only applicable rule to the box, then the box is correctly marked as stable, since *stability* is equal to *depth*.

- Whenever there is a binding to an external variable, the current box *stability* value is set to whichever is smallest: the current *stability* or the variable's home box *depth*. If the box *stability* and *depth* values are equal, then the *stability* will be set to the *depth* value of the external variable home box which states that the box is not stable.

- When going up in the And-Or Tree searching for work, the box's *stability* value should be set to the minimum value determined from the current *stability* value and the children boxes *stability* values.

- During *environment check* if a variable is removed from the list of external variables for an and-box, then the *stability*'s value for the and-box must be refreshed to the minimum *depth* value of all external variable home boxes.

Detecting stability using the rules presented has a small overhead to the system (between 1% and 2%). Note that our algorithm truly marks if a box is stable or not, in contrast to AKL's algorithm that only approximates stability.

5.3 Implicit Pruning

The LIGHT-BEAM implements two major simplifications that improve the search space by pruning logically redundant branches. The two rules are symmetrical, but whilst one is concerned with failed boxes, the other is concerned with successful boxes:

1. **false-in-and-simplification:**

   $$[\exists X : \theta \& A \& \ldots \& \texttt{false} \& \ldots \& B] \rightarrow \texttt{false}$$

   If a failure occurs at any point of the and-box, the and-box can be removed and failure is propagated to the upper boxes. This rule can be considered a generalisation of the failure propagation rules used in Independent And-Parallel systems [64].

2. **true-in-or-simplification:**

   $$\{\ldots \vee \texttt{ true } \vee \ldots\} \rightarrow \texttt{true}$$

   Provides implicit pruning of redundant branches in the search tree.

These two rules can provide substantial pruning. In a pure logic environment, the presence of a true-box in a branch of an or-box should allow immediate pruning of all the other branches in the or-box. In an environment with side-effects, the user may still want the other branches to execute anyway. For this reason, the LIGHT-BEAM allows the true-in-or simplification and the false-in-and simplification to be enabled for the leftmost goal only. We thus allow the user to explicitly disable these simplifications. A second possibility, in the style of Andorra-I, is to do compile-time analysis to automatically disable the true-in-or and false-in-and optimization for the boxes where some branches include builtins calls with side-effects, such as `write/1`

5.4 Explicit Pruning

In quite a few cases, the implicit pruning mechanisms we provide are not sufficient for controlling and reducing the search space. Standard examples are the problems where we require a single solution. The most popular (and controversial) pruning operator in Prolog is cut (`!`). The Prolog cut prunes both alternative clauses for the current goal and alternatives for all goals created for the current clause. For correct execution, cut assumes that the goals in the clause to its right have not yet exported constraints to the goals to the left of the cut. After the execution of cut, all boxes to the left of the cut operator may be discarded and their constraints on external variables may be promoted to the current and-box.

Figure 5.1 gives an example of pruning. In this example the and-boxes for the sibling clause `I` and for the rightmost alternative `G2` will be discarded. The constraints for `G1` will be promoted to the and-box for `G, !, H`.

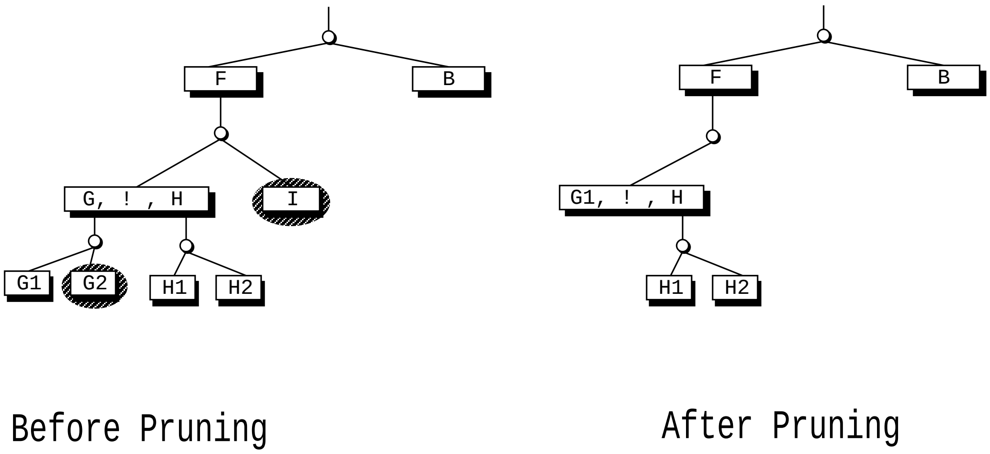

Figure 5.1: Cut scope.

The rule for cut can thus be written:

$$\{[\exists Y : \theta_1 \& \{[W : \theta_2 \& \mathtt{true}] \vee \ldots\} \& \; ! \; \& \; A] \vee B\} \rightarrow \{[\exists Y, X : \theta_1 \& \theta_2 \& A]\}$$

We name the conjunction of goals in the clause to the left of the cut operator the *guard* of the cut. Note that our rule says that cut only applies when the goals in the guard have been unfolded into a configuration of the form:

$$\{[W : \theta_2 \& \texttt{true}] \vee \ldots\}$$

That is, when the leftmost branch of the guard has been completely resolved and all of its deterministic constraints exported. The cut rule can thus be applied to three situations:

1. If θ_2 is empty or entailed by the current environment (that is, if it can be simplified to empty). We say that the cut is *quiet* [116]. Or,
2. if the disjunction consists of a single alternative. In this case we say that the cut is *degenerate*.
3. Otherwise, the cut is said to be *noisy*. The LIGHT-BEAM allows noisy cuts to execute if:
   - only splitting rules are available, and,
   - the and-box with the cut is leftmost in the current scope.

   These conditions are necessary, but not sufficient, to support Prolog semantics for noisy cuts.

We next discuss in more detail the issues in the design of explicit pruning for the LIGHT-BEAM. In the following discussion we refer mainly to cut, but similar restrictions apply to the usage of the commit operator.

5.4.1 Control for cut

Both splitting or Warren's forking should be used carefully when duplicating an and-box that contains a cut. Consider the example presented in Figure 5.2a. The lower-leftmost and-box contains a cut (`X,!,Y`). The and-box `W` is the only box within the cut scope. Suppose that all boxes are suspended and that the only available rule is fork/splitting. Both Warren's forking and the BEAM's splitting will transform the computational state presented in this example into an incorrect one. After applying the forking rule, the and-box C is in danger of being deleted by the execution of cut (see figure 5.2b). On the other hand, when applying splitting, the and-box W leaves the scope of the cut (see figure 5.2c).

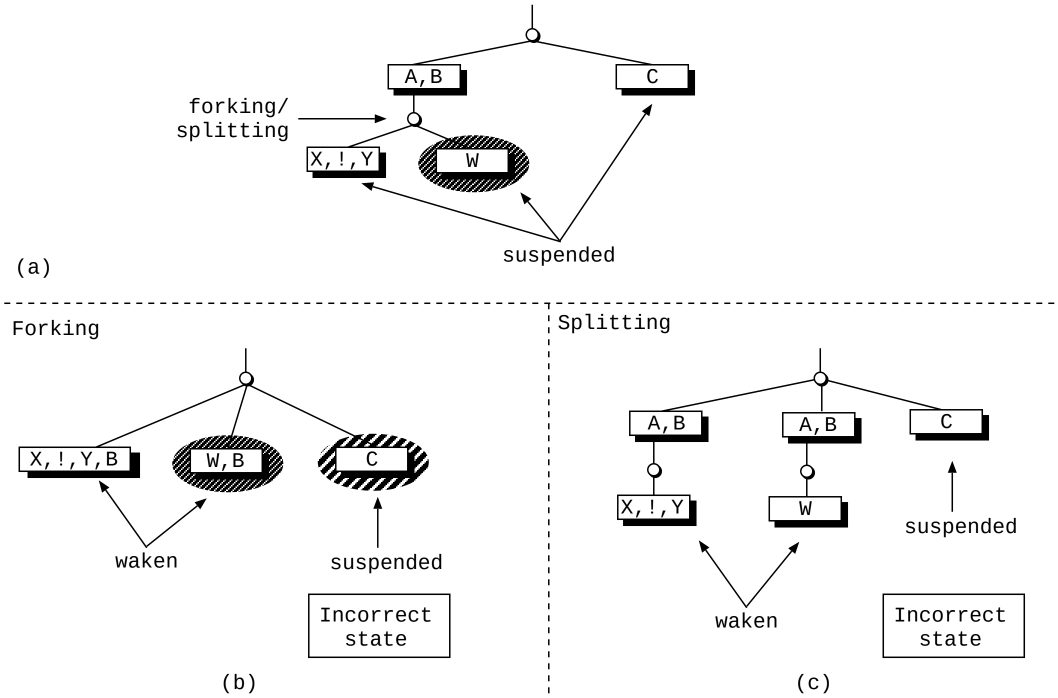

Figure 5.2: Incorrect use of fork/splitting and cut.

This problem is solved in our LIGHT-BEAM by explicitly disallowing splitting to an and-box containing a cut. Therefore, the scheduler does allow clauses with a cut to continue execution even when head unification constrains external variables (note that these bindings may not be made visible to the parent boxes). In this regard the BEAM is close to the AKL. On the other hand, the BEAM differs from the AKL in that goals to the right of the cut may also execute first: the cut does not provide sequencing.

As a results of this rule we have that:

- splitting can be applied freely to stable goals within the guard of the cut, as in this case splitting may not export constraints for variables external to the guard, or change the scope of the cut (see example on Figure 5.3).
- a cut can always execute immediately if it is the leftmost call in the and-box and if the and-box does not have external constraints.

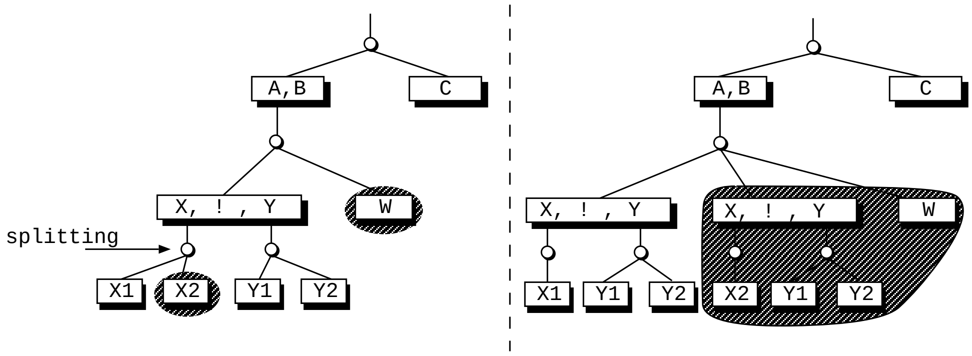

Figure 5.3: Correct use of splitting and cut.

$$\{[\exists Y : \theta_1 \equiv \{\} \ \& \ \mathtt{true} \ \& \ ! \ \& \ Y] \vee W\} \rightarrow \{[\exists Y : \theta_1 \equiv \{\}\& \ Y]\}$$

Consider the example illustrated in figure 5.4a. Both alternatives to the goal `a` suspended trying to bind the external variable `X`. The first alternative to the goal `b` contains a *quiet* cut that will be allowed to execute since it respects the conditions described previously: the alternative does not impose external constraints, and the cut is the leftmost call in the and-box. Note that in normal Prolog execution, the second alternative to `b` would also never execute, since the cut always executes, independently of value generated in the `a` goal to `X`.

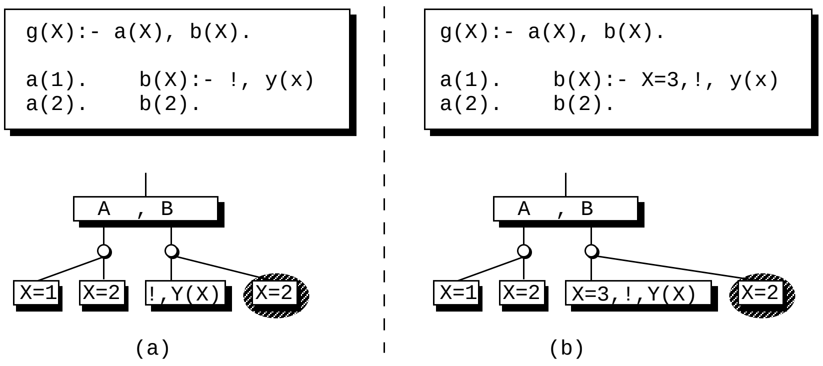

Figure 5.4: Cut example.

Figure 5.4b illustrates a different situation. In this case, the cut would not be allowed to execute since the alternative restricts the external variable `X` to the value `3`. Thus, the computation in this example would only be allowed to continue with a splitting on `a`. After the splitting, the values `1` and `2` would be promoted to `X` and thus make the first alternative to `b` fail.

- if an and-box containing a cut becomes leftmost in the tree, the cut can execute immediately when all the calls before it succeed (even if there are external constraints).

$[\exists X_1 : \theta_1 \ \& \ \texttt{true} \ \& \ \{\ldots\{[\exists X_n : \theta_n \ \& \ \texttt{true} \ \& \ ! \ \& \ Y] \vee W\}\ldots\}Z] \rightarrow$
$[\exists X_1 : \theta_1 \ \& \ \texttt{true} \ \& \ \{\ldots\{[\exists X_n : \theta_n \ \& \ Y]\}\ldots\}Z]$

For example, on Figure 5.5 the cut is allowed to execute immediately when X succeeds even if there are external constraints.

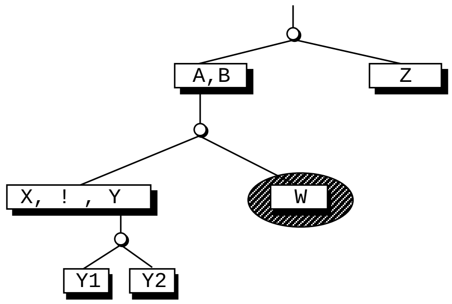

Figure 5.5: Cut in the leftmost box in the tree.

Allowing early execution of cuts will in most cases prune alternatives early and thus reduce the search space.

5.4.2 Degenerate parent or-boxes

One interesting problem occurs when the parent or-box for the and-box containing the cut degenerates to a single alternative. In this case, promotion and and-compression would allow us to merge the two resulting and-boxes. As a result, cut could prune goals in the original parent and-box. Figure 5.6 shows an example where promotion of an and-box containing a cut leads to an incorrect state as the and-box C is in danger of being removed by execution of cut.

We address this problem by disallowing and-compression when the inner and-box has a cut. Note that promotion is still allowed. Thus deterministic constraints are still allowed to be exported.

$[\exists Z_1, \ldots, Z_n : A \ \& \ \{[\exists W_1, \ldots, W_n : Z_i = \theta(W_j) \ \& \ X \ \& \ ! \ \& \ Y]\} \ \& \ B] \rightarrow$
$[\exists Z_1, \ldots, Z_n, W_1, \ldots, W_n : Z_i = \theta(W_j) \ \& \ A \ \& \ \{[X \ \& \ ! \ \& \ Y]\} \ \& \ B]$

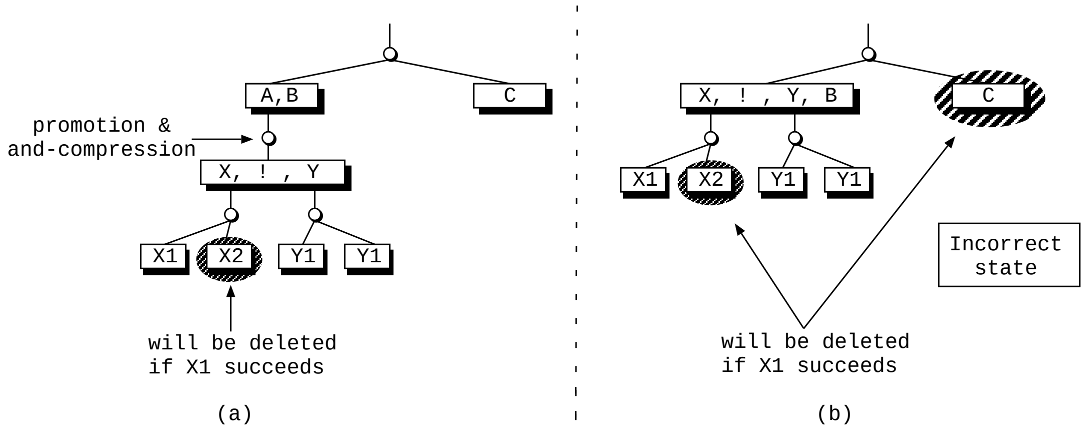

Figure 5.6: Incorrect use of promotion and cut.

The and-boxes are kept separated, as shown on Fig. 5.6a, but any variable that is local to the parent and-box (A in the example) also becomes local to the and-box containing the cut (and vice versa). This is achieved in our implementation by changing the and-box parameter *level* [1] to be equal to the parent and-box. And-boxes with the same *level* value, share all variables.

Our restriction preserves cut scope, hence guaranteeing that the BEAM will never go into the incorrect state presented on Figure 5.6b.

5.4.3 Early execution of goals to the right of cut

We allow execution of goals to the right of the cut even before the cut has been executed. This is disallowed in systems such as Andorra-I [116] and in languages such as AKL. In AKL the cut is a guard operator that provides sequencing. We have chosen a different approach because goals to the right of the cut are not speculative. In fact, we may have interest in running several and-goals, an example in which this is the case is finding the first solution to several independent queries [119].

In general, and in contrast with AKL, we would like to separate pruning and sequencing issues. Overloading the two operations in the same operator makes sense for a concurrent language, but it is not necessarily a good idea for query resolution.

[1]depth counter that can be used to classify variables as local or external (see section 6.2.3).

5.4.4 Quietness and noise

Note that our algorithm only allows non-leftmost execution of quiet guards: all constraints to exported variables must be either entailed by the current environment, or deterministic. We follow here the Andorra-I definition of quietness that allows deterministic non-entailed constraints [116]. Guards that execute quietly generate the same result independently of how external goals are executed.

In contrast to quiet guards, noisy guards can be used to implement meta-predicates. Their meaning depends on program execution order. In general, sequencing as performed in Andorra-I is the only form of guaranteeing correct execution for these programs [116]. We have allowed noisy pruning in the LIGHT-BEAM in order to support Prolog programming style.

5.5 Termination

Ideally, we would like all computations that have a solution to terminate in the EAM. Unfortunately and as discussed in section 4.7, the original definition for the EAM does not even guarantee termination for some programs that do terminate under Prolog. Control for the LIGHT-BEAM as defined so far does not guarantee this property. Extra forms of control are thus required.

5.5.1 Eager non-deterministic promotion

Warren and Gupta proposed a solution to the termination problem by using Eager Non-Deterministic Promotion [56]. Their solution requires that variables whose non-determinate bindings can be eagerly promoted to be annotated. These annotations can be generated by means of compile time analysis, or they can be supplied by the programmer explicitly. The variables for producer goals should be classified as *guessable*, *non-guessable* or *others* before execution. Forking is immediately performed whenever a binding for a *guessable* variable is generated by a producer.

The LIGHT-BEAM supports a similar solution. We allow the user to explicitly declare predicates as producers. These predicates perform splitting as soon as they are called.

Although this solution is not as complete as Warren and Gupta's, it is easier to understand, easier to implement, can still increase parallelism, and in a few cases it actually improve the search space. We discuss performance results in chapter 8.

Eager-splitting can also be used to avoid non-termination. As example consider the ancestor program, that makes the EAM loop, presented in figure 4.17. Annotating the `parent` predicate as producer would force splitting immediately when calling this goal and the loop situation presented in figure 4.18 would no longer occur. Figure 5.7 presents the new computation for the query `ancestor(a,Z)` using eager splitting on the `parent` goal. To simplify the notation we use `a(X,Y)` for `ancestor(X,Y)` and `p(X,Y)` as `parent(X,Y)` in Figure 5.7 and in the following discussion.

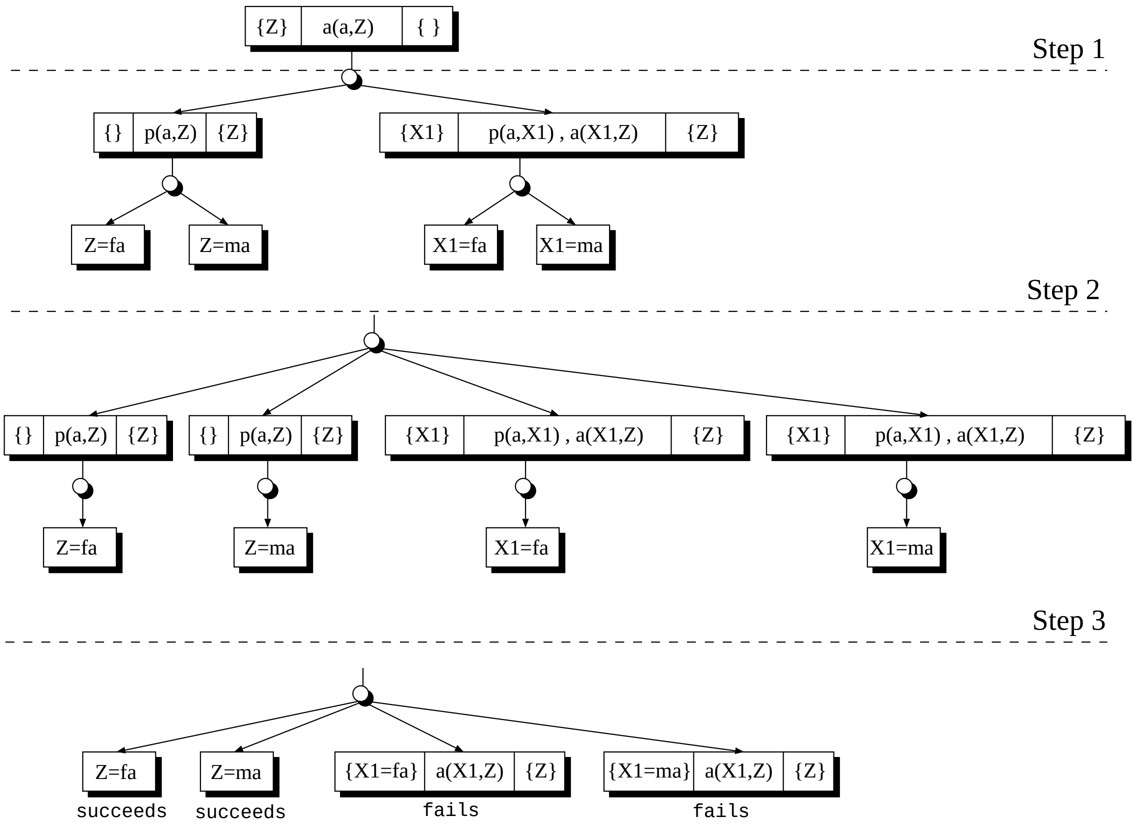

Figure 5.7: The ancestor example using eager splitting.

When the query `a(a,Z)` is executed an or-box with two alternatives is created (step 1). On the second step, instead of suspending all calls to the `parent` goal, splitting would be taken (as it is a producer goal). After the splitting, (step 2), the bindings generated would be promoted (step 3). At this point, the computation would allow `a(X1,Z)` to be executed with `X1` already bound. But, both calls to `ancestor` (`a(fa,Z)` and `a(ma,Z)`), would immediately fail when running the `parent goal`. Thus the computation would terminate with two solutions `Z=fa` and `Z=ma`.

5.5.2 Tabling

A more elegant solution that guarantees termination to a significant class of programs, including Datalog programs, is to use tabling, that is to store intermediate solutions to computations. This solution seems to be ideal in two contexts. First it guarantees that the computation ends in programs that have finite solutions. In second it fits naturally with one of two main premises of the EAM, reducing search space through the reuse of goals.

Once again we refer to the `ancestor` program presented in figure 4.17. Hereby, we use it to exemplify how tabling would be used to solve the non-termination in the EAM. We select the `ancestor` goal to be tabled. The first steps of execution are illustrated in figure 5.8. The computation proceeds normally until the third level of and-boxes, with all calls to the `parent` goal suspending. In this level, the call to the `ancestor` goal, `a(X2,Z)` will suspend on table, because a similar call to ancestor was already made on the second level and-boxes (`a(X1,Z)`). With all computation suspended, splitting is the only applicable rule. Splitting can thus be performed in any suspended boxes to the `parent` goal. Note that splitting on the rightmost second level or-box (p(a,X1) would make its parent and-box fail since neither `X1=fa` neither `X1=ma` is compatible with the restriction imposed by the lower level and-boxes (`X1=a`). The query solutions are thus obtained by the split on the leftmost or-box, being: `Z=fa` and `Z=ma`. The tabling mechanism with the eager splitting make bindings to be propagated earlier, and thus avoid the initial looping situation.

We next discuss a design the tabling computational rules for the LIGHT-BEAM. We assume variant-based tabling [25].

We assume that there is a store Π with mappings of the form $G \triangleright \{G\sigma_1, \ldots, G\sigma_n\}$. A mapping may be partial or complete. The store is also named the table space and can be implemented through hash-tables or tries. We also assume an attribute $Tabled$ that holds true if we can (want to) table this procedure.

The LIGHT-BEAM only allows the following two rules for tables goals:

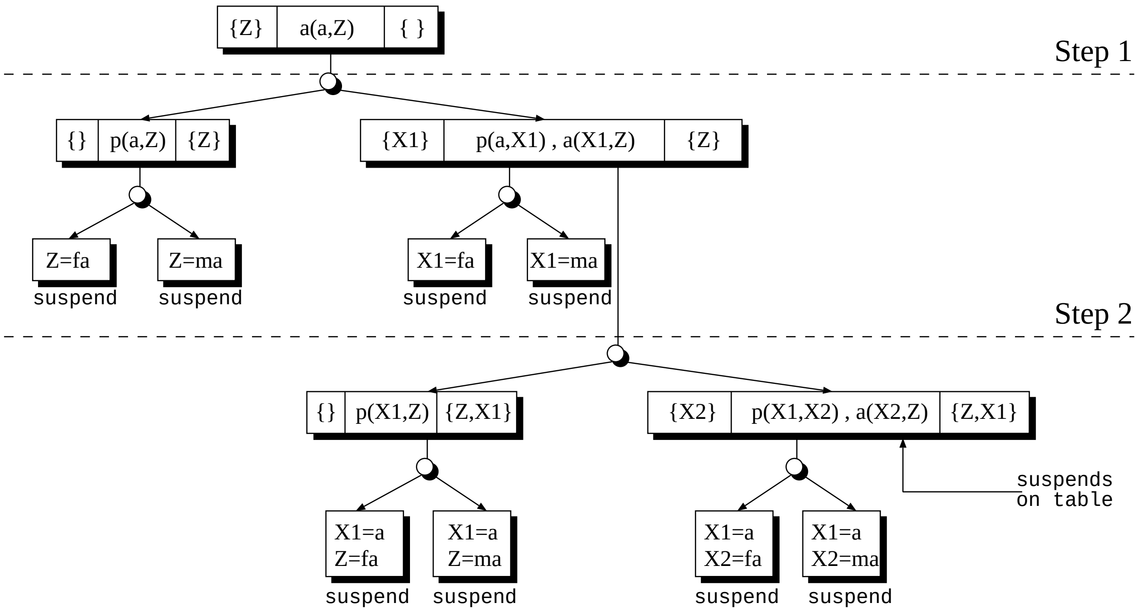

Figure 5.8: The ancestor example with tabulated goal.

1. **Tabled Reduction:**

   $G \wedge (G \triangleright \{G\sigma_1, \ldots, G\sigma_n\}) \in \Pi \rightarrow \{[\exists Y_1 : \sigma_1] \vee \ldots \vee [\exists Y_n : \sigma_n]\}$

   In tabling terminology, this rule implements a consumer for variant-based tabling. Intuitively, if solutions for a goal are stored in a table we will consume from the table instead of reducing the goal. In EAM terminology, we will not unfold a goal if the goal is described in Π. Instead, we replace the goal by an or-box with all the constraints imposed by the goal.

2. **Tabled Generation:**

   $G \wedge \neg(G \triangleright \{\ldots\}) \wedge \mathit{Tabled}(G) \rightarrow \{[: \sigma \& \texttt{open\_table}(G)] \vee [\exists Y_1 : \sigma_1 \& C_1 : \texttt{new\_answer}(\texttt{C}_1)] \vee \ldots \vee [\exists Y_n : \sigma_n \& C_n : \texttt{new\_answer}(\texttt{C}_\texttt{n})] \vee [: \sigma \& \texttt{complete}(G)]\}$

   where $\sigma = \{v = v | v \in vars(G)\}$.

The key to the implementation of generators are the builtins `open_table/1`, `complete/1` and `new_answer/1`. They can be described as an intermediate-level description of the tabling operations used in the SLG-WAM, where the first builtin controls start of the tabling operation, the second builtin adds solutions to the table, and the last builtin controls completion. Their operation is as follows:

- `open_table/1` waits until we can table the goal. Three cases may happen:

1. The goal G may become a variant of some tabled goal G'. `open_table/1` applies true-in-or simplification and then calls tabled reduction on G.
2. The computation is pruned away.
3. The goal G enters the table: we create an entry for the goal using current bindings for G.

- `new_answer/1` waits until `open_table` finished execution and next adds a solution corresponding to the current value for G.
- `complete/1` waits until all other branches of the or-box reduce to true boxes.

The control issues in the implementation of tabling are:

- Tabled Reduction always applies for tabled goals if a variant of the goal is in the table space.
- Tabled Generation always applies if a variant of the goal is not present in the table.
- We can only apply reduction and splitting to an `open_table/1` box. Reduction applies if:

  1. all alternatives of G concluded and we found a single solution;
  2. a binding to a variable in G made G a variant of some other goal G';
  3. `open_table/1` is the only member of its disjunction.

  The last rule guarantees that `open_table/1` will execute immediately after splitting.

- `new_answer` applies if **(i)** the corresponding `open_table/1` executed, and **(ii)** all goals to its right executed.
- we can only apply reduction for a box with `complete/1`. Reduction applies if no-rules apply to all sibling or-branches.

We next discuss some issues in more detail.

Early Export of Bindings The alternatives for a generator G may never deterministically export bindings to the outside environment. Otherwise, a tabled goal G might wake up some other goal G' which would then deterministically constrain a variable of G.

The `complete/1` and-box guarantees non-determinism quite naturally, by being the last alternative to the or-box for G.

Batched and Local Scheduling The strategy we present is quite similar to batched scheduling [46], in the sense that after execution of `open_table/1`, splitting is allowed on the branches of the search-tree. Solutions can be hence exported immediately. Unfortunately, this creates a problem in the BEAM, as some other goal may be woken up and deterministically constrain a variable in G. We have two alternative solutions to this problem:

1. In the same vein as local scheduling [46], we can introduce a consumer before the producer, and always export solutions via the consumer. To implement this solution we simply have to add the right-hand-side of the consumer rule as the alternative to the right of `open_table/1` and guarantee that `new_answer/1` always fails.

2. In the same vein as the *filtered bindings scheme* [101, 98] we can rename variables before we execute the clauses and control access to the external environment through filters. This solution is more complex but would provide greater flexibility.

Completion We can perform completion for a generator G if **(i)** no other operations apply on the scope of G; **(ii)** no consumer on the scope of G depends on a generator above G.

To understand operation **(ii)** we can abstract completion by drawing a graph such that each node corresponds to the or-box for a tabled goal, and we have an edge from node a to node b if node b is the closest tabled ancestor of node b. The BEAM rules are quite similar to parallel implementations, such as OPTYap, in that they allow several generators to proceed in parallel [109]. Thus to detect completion in the general case we need a generic parallel algorithm, such as the one proposed by Rocha [107].

On the other hand, we would like to use sequential algorithms, as they have a simpler implementation. We can use the simpler sequential algorithm if to guarantee that two generator nodes always have a common ancestor which is also a generator node (please see Swift [136] and more recently Rocha [108]). This can be guaranteed by only allowing tabled generation if **(i)** no other tabled generation is in progress; or if **(ii)** the current goal is a descendent of a generator.

Pruning The BEAM has been designed to allow both implicit and explicit pruning. Implicit pruning is made possible by the rules that propagate success and failure. Explicit pruning is achieved through cut and commit.

Aggressive pruning may interfere with tabling. Careless pruning may result in never computing all solutions to a goal and generating incomplete tables. Fortunately, the BEAM provides much better flexibility than Prolog-based schemes. Implicit pruning also makes it clear that we should conclude computation for a tabled goal, even if pruned.

We next discuss how pruning should be implemented in the BEAM. Without loss of generality, consider we want to prune a tree rooted at node N:

- If the node is non-tabled, we can physically recover space for the node, and prune its children.
- If the node is a consumer, we can prune the node and its children.
- If the node is a generator, the node cannot be complete, although it cannot also generate further solutions. The parent of such a node should be the special node $\top$.

Last note that pruning allows one to simplify the dependencies between generator and consumer nodes.

Discussion We see our algorithm as a generalization of the tabling algorithm used in the SLG-WAM [112]. We could take advantage of observation that the EAM is really a low-level model for the execution of logic in order to integrate the main principles of the SLG-WAM into our execution rules: checking for variants, new answers, and completion.

The major problem we had to address was when to start execution of tabled goals. Whereas this is trivial in Prolog, the BEAM allows considerable latitude. At this point the notion of stability was quite helpful in providing us with a heuristic to tackle this problem. The problem of when to export new answer can be addressed by considering the same problem in the scheduling of tabled programs. Last, the problem of completion could take advantage of recent progress in the implementation of tabling for shared memory parallel systems.

We believe that our work show that tabling fits naturally with the EAM scheme. The combination provides a very powerful computational model, and we expect to pursue this in the future. Note that, tabling per se does not guarantee termination (or even Prolog-like termination) for programs with unbounded structures. We need at least eager non-deterministic promotion to achieve this goal.

5.6 Final Remarks

In this chapter we have address four major issues over our kernel design for the Extended Andorra Model: when to suspend, when to allow splitting, how to perform pruning, and how to obtain better termination properties.

We discuss on how to take the best advantage of the EAM with the least programmer intervention. We have presented different approaches to exploiting control and contrast them to the guard-style approach used in AKL. We show that the model offers scope for both implicit and explicit pruning and that tabulation can be naturally integrated into the EAM.

We call this model the LIGHT-BEAM. The LIGHT-BEAM extends the BEAM to support more flexible search, implicit and explicit pruning of the search space, and a mechanism for guaranteed termination. These extensions are a step towards Warren's ideal of a *minimum number of inferences*.

Chapter 6

Implementing the BEAM

The purpose of this chapter is to present the architecture used to implement BEAM, namely the *And-Or Tree Manager* and the *Abstract Machine*.

6.1 BEAM Architecture

Figure 6.1 illustrates the architecture organization for the BEAM execution model. The BEAM was built on top of the YAP [38] Prolog system. It reuses most of the YAP *compiler* and its *builtin library*. The shadowed boxes show where the EAM stores data. The *Code Space* stores the data-base with the predicate/clause information, plus the bytecode to be interpreted by BEAM's *Abstract Machine Emulator* (which we refer shortly as *Emulator*). The *Global Memory* stores the And-Or Tree, and it is further subdivided into the *Heap* and the *Box Memory*. The *Box Memory* stores dynamic data structures including boxes and variables. The *Heap* holds Prolog terms, such as lists and structures. The *Heap* uses term copying to store compound terms and is thus very similar to the WAM's *Heap*. The major difference is that on the BEAM, *Heap* memory can not be recovered after backtracking. A *Garbage Collector* is thus necessary to recover space in this area. We will leave the details on memory management and on the garbage collecting algorithm until chapter 7.

The BEAM relies on two main components, shown as ovals in Fig. 6.1:

Emulator: runs WAM-like code to perform unification and to setup boxes. Unifi-

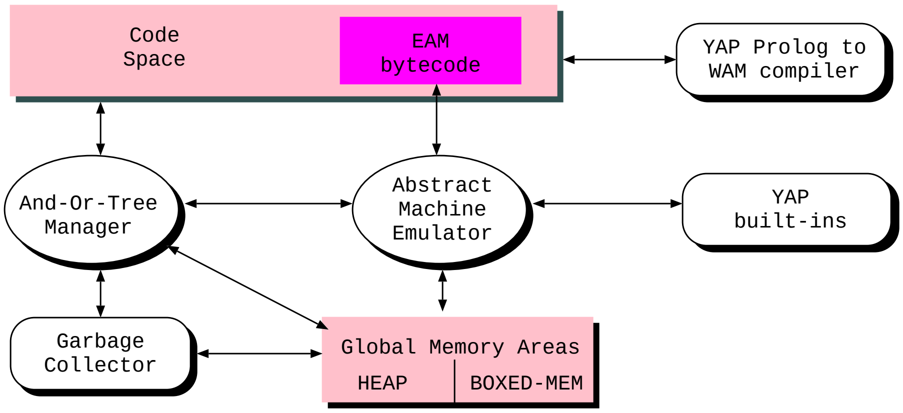

Figure 6.1: Execution Model.

cation code is similar to the WAM. Control instructions follow a compilation scheme similar to the WAM but result in a rather different execution pattern.

And-Or Tree Manager: applies the BEAM rewriting rules to the existing and-boxes and or-boxes until WAM-like execution for the selected goal can start.

The *And-Or Tree Manager* handles most of the complexity in the EAM. It uses the *Code Space* area to determine how many alternatives a goal has and how many goals a clause calls. With this information the *And-Or Tree Manager* constructs the tree and uses the EAM rewriting rules to manipulate it. The Manager requests memory for the boxes from the *Global Memory Areas*. The *Emulator* is called by the *And-Or Tree Manager* in order to execute and unify the arguments of the goals and clauses. As an example consider the clause: `p(X,Y):- g(X), f(Y)`. When running this clause the *And-Or Tree Manager* transforms the `p(X,Y)` into one and-box, and calls the *Emulator* to create the subgoals and or-boxes for `g(X)` and `f(Y)`. Control returns to the *And-Or Tree Manager* if the boxes need to suspend.

The details on how BEAM stores the And-Or Tree, the design of the *Emulator* and of the *And-or Tree manager* are described in more detail in the following sections.

6.2 Object Representation

In this section we describe how the BEAM represents or-boxes, and-boxes, local variables, and external references to local variables.

6.2.1 Or-Boxes

Or-boxes represent open alternatives for a goal. Figure 6.2 presents the structure of an or-box. Each or-box refers to its parent and-box through the `parent` pointer and the call in the parent and-box that created this or-box, `id_call`. `nr_all_alternatives` gives the number of alternatives available. Last, the box points to a list of `alternatives`, where each element in the list includes:

- a pointer to a corresponding and-box, `alternative`, initially empty; it is initialized only when the alternative is explored;
- a pointer to the goal arguments, `args`; The first alternative creates the arguments vector. The last alternative to execute, after performing head unification, recovers the `args` vector as free memory. Each alternative needs an entry to the `args` vector because the *and-compression* and the *splitting* rule can join in the same or-box branches that use different arguments;
- a pointer to the code for the alternative, `code`; and,
- the `state` of the alternative. Initially, alternatives are in the `ready` state. They next move to the `running` state, from where they may reach the `success` or `fail` states, or they may enter the `suspend` state. Suspended alternatives will eventually move to the `wake` state. As an optimization, if no more goals need to be executed for the goal but the alternative is suspended, the alternative enters a special `suspended_on_end` state.

Note that if *nr_all_alternatives* is one, then there is no need to keep the or-box. Furthermore the determinate promotion rule can be applied to the single alternative. If *nr_all_alternatives* equals zero, the box has failed and we can perform *fail propagation*.

6.2.2 And-Boxes

And-boxes represent active clauses. And-boxes require a more complex structure than or-boxes, as they store information on what goals are active as well as on external and internal variables associated with the clause. Figure 6.3 shows an and-box.

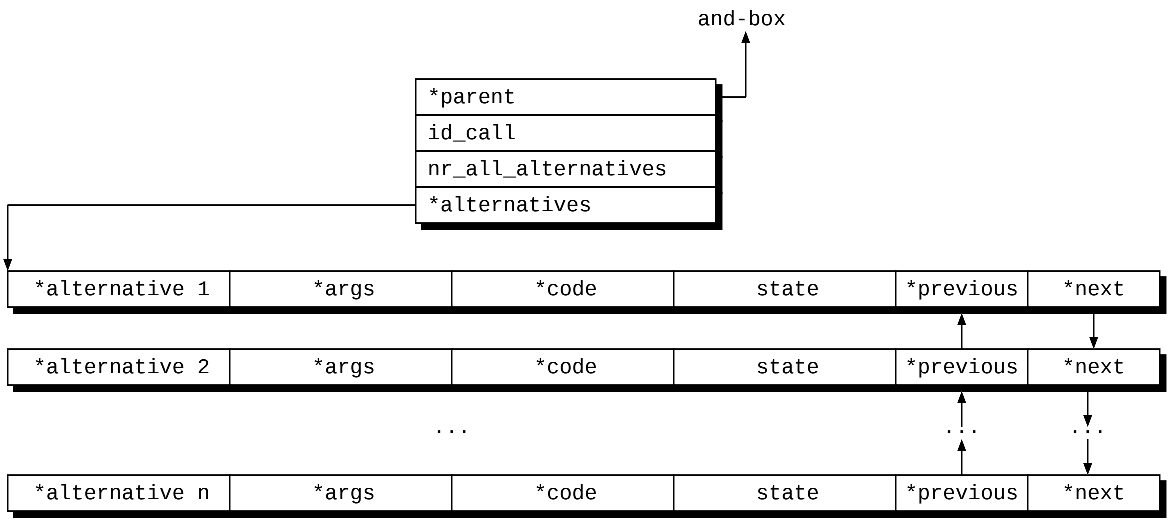

Figure 6.2: Or-Box representation.

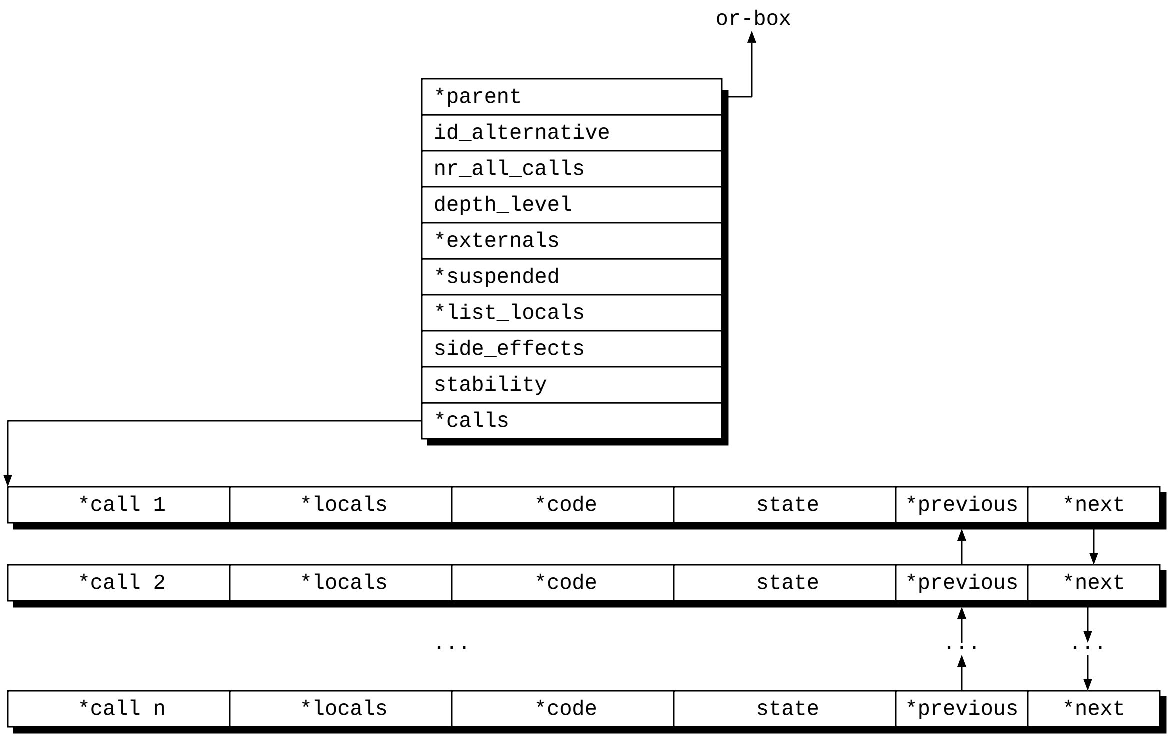

Figure 6.3: And-Boxes representation.

Access to the parent node is performed through the `parent` pointer and through the `id_alternative` field. The first points back to the parent and-box. The second indicates to which alternative the and-box belongs. Subgoal management, requires knowing the number of subgoals in the clause, `nr_all_calls`. Each and-box maintains a `depth_level` counter that is used to classify variables. The `list_locals` field maintain a list of vectors containing the variables local to this and-box. Variable control is discussed in more detail in section 6.2.3. A list of bindings to external variables is accessed through the `externals` field. The and-box may have suspended trying to bind some variables, if so this is registered in the `suspended` field. The `stability` field is used to identify if the box is stable. If side-effects predicates are present in the goals of this and-box, they are registed on the `side_effects` field. Last, each subgoal or call requires separate information:

- a pointer to a corresponding or-box, `call`, initially empty; it is initialized when the call is open;
- a pointer to the `locals` variables vector. Each goal needs a entry to the `locals` variables because the *promotion* rule may add other goals and other variables to the and-box. Still each goal needs to be able to identify its own local variables;
- a pointer to the `code` for the subgoal;
- `State` information says whether the goal is `ready` to enter execution, is `running`, or has entered the `success` or `fail` states. Goals may also be `suspended` or `waiting` on some variable, upon which they will enter the `wake` state.

Initially each and-box has one vector of local variables. However, the number of vectors of local variables in an and-box may increase since the promotion rule allows one to promote local variables to a different and-box. We discuss local variables next.

6.2.3 Local variables

The representation for a variable is the *local variable* structure as illustrated in Figure 6.4. A local variable either belongs to a single subgoal in an and-box, or it is shared between subgoals. The `value` field stores the current working value for a variable. Unbound variables are represented as self-referencing pointers. Variables also maintain a list of and-boxes suspended on them, and point back to their `home` and-box.

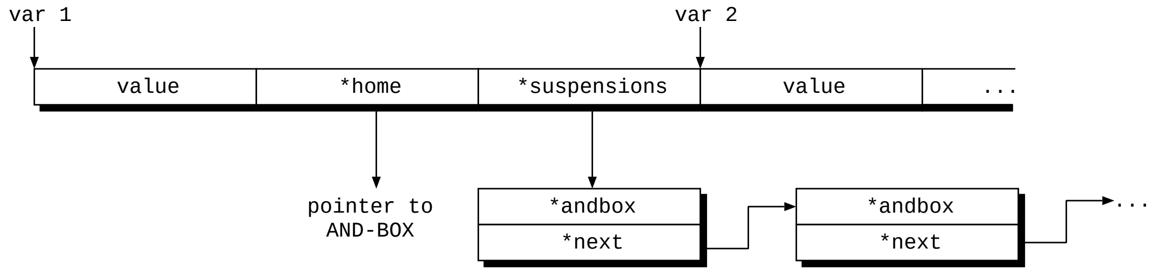

Figure 6.4: Local variables representation.

The `home` field of a variable structure points directly to its home and-box. However this field is not sufficient to completely determine if a variable is local or not to an and-box, since a variable can be local to several boxes. The problem arises in the case where an and-box is a single alternative and includes pruning operators. The promotion rule can be applied but the and-boxes must be kept separate even if sharing the same variables (please see figure 6.5). Hence, the and-compression rule cannot be applied since moving the pruning operators to an upper level in the tree may lead to an incorrect state.

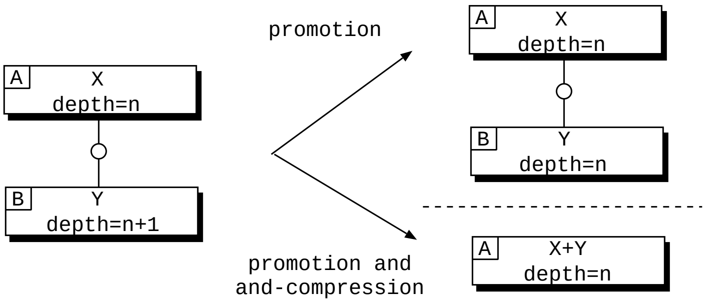

Figure 6.5: Promotion and variables.

Having the `home` field of the variable structure pointing to a list of and-boxes would allow us to know all and-boxes where the variable is local. We have not used this solution because having to search through a list every time one needs to classify a variable would be very inefficient.

The BEAM thus uses a different solution. The BEAM uses a depth-counter associated to an and-box to classify variables. We can now precisely define local variable as follows:

A variable is said to be local to a and-box G if the `depth` counter of the and-box G equals the `depth` counter of the variable's `home`. Otherwise the variable is said to be external to the and-box G.

6.2.4 External variables

Each and-box maintains a list of *External variables*, that is, of bindings for variables older than the current and-box (see Figure 6.6). Each such binding is represented as a data-structure that includes a pointer to the variable definition, `local_var`, and to its new value, `value`. Whenever a goal binds an external variable, the assignment is recorded both in the current and-box as an external reference and at the local variable itself. This way, whenever a descendent and-box wants to use the value of the external reference, it can simply access the local variable. The `external_reference` data-structure generalizes the Prolog's trail, allowing both unwinding and the rewinding of bindings performed in the current and-box. Our scheme for the external variables representation is very similar to the *forward trail* [154] used in the SLG-WAM [136, 113]. The SLG-WAM's *forward trail* is also a list rather than a WAM stack trail. It contains, information about the value to which the variable was bound, the address of the bounded variable and a pointer to the previous trail frame.

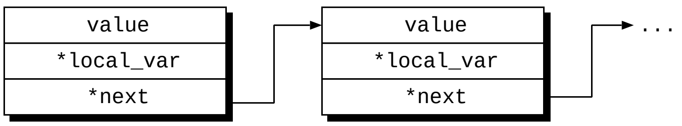

Figure 6.6: External variables representation.

6.2.5 Suspension list

The suspension list is a doubly linked list that keeps information on all suspended boxes (see figure 6.7). Each entry in the list maintain a pointer to the suspend and-box, `and_box`, and information on why the and-box suspended, `reason`. Goals may be suspended for trying to bind external variables, or they can be waiting for some event to occur such as, waiting to be leftmost or waiting for a variable to be bounded so that it can be used in a builtin operation.

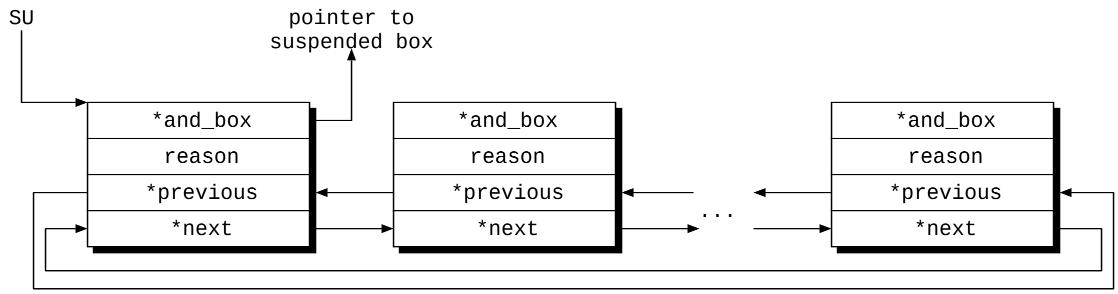

Figure 6.7: Suspension list representation.

In contrast to the AKL implementation that uses one stack for suspended boxes and another to waken boxes, the BEAM uses the same list to maintain information on suspended and awoken and-boxes. The `SU` pointer marks the beginning of the suspension list (that can be empty). Whenever an and-box suspends, an entry is added to the end of the suspension list. If a suspended and-box receives a waken signal, the and-box entry is moved to the beginning of the list. Thus, if there are awoken boxes, they are immediately accessed by the `SU` pointer. Also note that we always want to work with awoken boxes before working with the suspended ones. By default, the order in which suspended boxes are chosen to be split is determined not by the order in which they are in the suspension list, but for being the leftmost in the And-Or Tree. The BEAM finds the leftmost suspended box by performing explicit depth-search on the And-Or Tree.

6.2.6 Unification of two variables

In this section we discuss in more detail the algorithm that performs the unification of two variables. One important consideration is that an and-box only suspends when trying to bind one or more external variables. Some of these bindings can be generated when unifying two variables.

The original proposal for the EAM did not present details on variable to variable unification. In fact, the BEAM, as the WAM, has an optimal form to unify two different variables in an EAM environment. Making an arbitrary choice may affect performance by forcing unnecessary suspensions as we explain next.

First, in the BEAM a variable can belong to an and-box, we refer to these variables as *permanent* variables, or may be a *temporary* variable if it is stored in the Heap. Thus, there are three possible cases of variable to variable binding:

1. *temporary variable* to *permanent variable*: in this case the unification should makes the temporary variable refer to the permanent variable. An immediate advantage is that the computation may not suspend. Moreover, unifying in the opposite direction may lead to an incorrect state, since future dereferencing of the permanent variable would reference a temporary variable that can unify without suspending the computation.

2. *temporary variable* to *temporary variable*: this case may never occur. Temporary variables are only created when constructing compound terms in the Heap. Moreover, whenever these variables are created they immediately unify with a permanent variable. Thus, using the previous rule, a temporary variable is always guaranteed to reference a permanent variable.

3. *permanent variable* to *permanent variable*: the permanent variable that has its home box at a lower level of the tree should always reference the permanent variable that has its home box closest to the root of the tree.

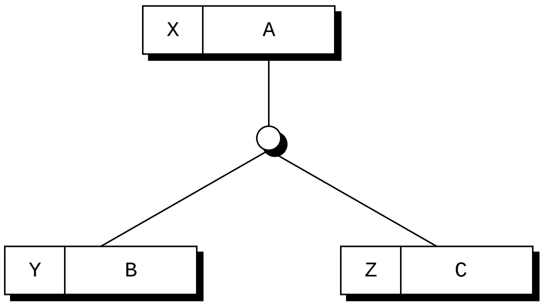

Figure 6.8: Binding two permanent variables.

Consider as an example the tree illustrated in figure 6.8 with three and-boxes: `A`, `B`, and `C`. Each box contains a single local (permanent) variable: `X`,`Y`, and `Z` respectively. Consider that the computation is processing the and-box `B` and that becomes necessary to unify the variables `X` and `Y`. If the variable `Y` is made to reference the variable `X`, no suspension is necessary since the variable `Y` is local to the and-box `B`. Moreover, if the and-box `B` fails or if the computation continues to the and-box `C` no reset would be necessary in the `X` variable. On the other hand, if `X` is made to reference the variable `Y`, the computation would need to suspend since `X` is not local to the and-box `B`. Moreover, if the and-box `B` fails or if the computation continues to the and-box `C`, the `X` variable would necessarily have to be reset.

By following these unification rules one can often delay the suspension of an and-box and thus delay application of the splitting rule.

6.3 The Emulator

The *Emulator* is responsible for running WAM-like code in order to perform unification and setup goals. The *Emulator* executes abstract machine instructions such that the state of the execution is characterized by several registers. The Emulator re-uses most of the WAM instructions and WAM registers. However new instructions and new registers are needed to cope with this rather different execution model.

6.3.1 Registers

Similar to the Warren's Abstract Machine the BEAM internal state is saved in several registers:

- `PC`: Program Counter;
- `H`: Top of Heap;
- `S`: Structure pointer (points to the Heap);
- `Mode`: controls whether unification is in read or write mode;
- `X1,X2,...`: registers for temporary variables, also used as arguments registers;
- `OBX`: pointer to the working or-box.
- `ABX`: pointer to the working and-box.
- `SU`: pointer to the start of the list of suspended and-boxes.

Note that, except for the last three, the registers are inherited from the WAM. On the other hand, many of the WAM registers such as `B`, `ENV`, and `HB` are not needed in the BEAM emulator as the BEAM does not implement backtracking. Instead information is stored directly in the And-Or Tree.

6.3.2 Abstract machine instructions

Code for the BEAM abstract machine very closely follows the WAM. The BEAM abstract machine instructions include the WAM `get`, `put` and `unify` instructions, plus some novel instructions that deal with control:

`explore_alternative i`: explore the `i`th alternative for the current or-box. If there are more than one alternative, creates a new and-box. Else readjusts the execution state so the parent and-box is reused for the alternative being executed (*deterministic reduce and promote* optimization). On both cases, prepare the arguments registers, and starts executing the code for the unification of the arguments with the head of the goal.

`prepare_calls n`: prepares the and-box to manage `n` subgoals. Each subgoal record points to the start code for the call, and is initialized as `READY`, meaning that they are ready to be explored. If the and-box does not have external variables, execution is then passed to the *And-Or Tree Manager* through the `next_call` module. Otherwise, the and-box is marked as suspended, and execution enters the *And-Or Tree Manager* through the `suspend` module.

`call_pred n`: creates one or-box with `n` branches, where `n` is the number of alternatives to `pred`. Each branch records points to the starting code of the corresponding alternative, and all branches are also initialized as `READY`, meaning that they are ready to be explored. Execution is then passed to the *And-Or Tree Manager* through the `next_alternative` module.

`proceed`: this instruction returns control from a clause to the *manager*. If the and-box does not have external variables, it has succeed, and enters the *And-Or Tree Manager* through the `success` module. Otherwise, the and-box is marked as suspended, and execution enters the *And-Or Tree Manager* through the `suspend` module.

The major difference between the BEAM's `get`, `put`, and `unify` and the corresponding WAM instructions is that in the BEAM permanent variable slots, or (`Y`) slots, refer in fact to the and-box local variables. Whenever one of these instructions tries to bind an external variable, an entry is added to the `externals` field on the current and-box. Note that on WAM, during the unification of variables a check is done to determine if trailing is necessary. Thus, the BEAM `externals` field can be viewed as similar to the WAM trailing mechanism.

Compilation Compiling Prolog clauses to the BEAM abstract machine instructions is very similar to WAM compilation. Figure 6.9 illustrates an example of code generation.

Note that, unlike in the WAM, code for rules in the BEAM does not end with a `execute` instruction. The BEAM abstract machine is goal based. The `explore_alternative i` initializes the ith or-branch by creating a new and-box in it. It is followed by a sequence of `get` instructions that perform the head unification. For *facts* follows the `proceed` instruction that decides if the computation succeeds or if suspends (if there are restrictions to external variables), entering the *Manager* through the `success` or the `suspend` modules respectively. For *rules*, follows the `prepare_calls` instruction instead. This instruction creates in the and-box as many subgoals as calls. Each subgoal is initialized to point to the start code of each call (marked as `Ln` in figure 6.9). Then, the `prepare_calls` jumps to the `suspend` module if there are constraints on external variables, or to the `next_call` port otherwise. Thus, it is up to the *Manager* to decide how and when to execute the calls. The calls code is composed with a series of `get` (and possibly a few `unify`) instructions followed by the `call_pred` instruction. The `call_pred` instruction creates and initializes an or-box with as many branches as the number of valid alternatives (determine by the indexing on the first argument). Execution is then passed to the `Manager` through the `next_alternative_port` that will decide which alternative to execute (the leftmost by default).

6.4 The And-Or Tree Manager

The *And-Or Tree Manager* is the heart of our system. Its task is to decide which rewrite rule should be applied to the current tree and then execute it. The compu-

```
        ancestor(X,Y):- parent(X,Y).
        ancestor(X,Z):- parent(X,Y), ancestor(Y,Z).

        parent(a,fa).
        parent(a,ma).
------------------------------------------------------------
                          ancestor/2

   explore_alternative  1       |    explore_alternative  2
   get_var           A1,Y1      |    get_var           A1,Y3
   get_var           A2,Y2      |    get_var           A2,Y1
   prepare_calls      1 L1      |    prepare_calls  2 L1 L2
 L1:                            | L1:
   put_val           A1,Y1      |    put_val           A1,Y3
   put_val           A2,Y2      |    put_val           A2,Y2
   call_pred     parent/2       |    call_pred     parent/2
                                | L2:
                                |    put_val           A1,Y2
                                |    put_val           A2,Y1
                                |    call_pred   ancestor/2
------------------------------------------------------------
                           parent/2

   explore_alternative  1       |    explore_alternative  2
   get_atom          1, a       |    get_atom          1, a
   get_atom          2,fa       |    get_atom          2,ma
   procceed                     |    procceed
                                |
------------------------------------------------------------
```

Figure 6.9: BEAM Abstract Machine Code for ancestor.

tational tree contains and-boxes and or-boxes that can be in different states. The possible states for a box are:

- `ready`: when a box is ready to start execution;
- `running`: the box is already active and running;
- `fail`: the box has failed;
- `success`: the box has succeed;
- `suspended`: the box is suspended in some point;
- `suspended-on-end`: the box is suspended at the end and there is no more code left to execute in it. This is a special case of `suspended`. The general case needs to continue the box execution. In this case we know that the execution is over, so when the suspension is activated, the box can jump immediately to `success` state.
- `awoken`: the box was suspended but has received a signal to be activated and can be restarted anytime.

The *And-Or Tree Manager* manage the states of and-boxes and decides when to move boxes from one state to another.

The *And-Or Tree Manager* can be divided into eight different operations:

`suspend`: this routine adds the current and-box to the suspension list. Next, the routine clears all assignments saved in the list of the external variables. Each external variable is also added to the suspension list included in the respective local variable. After that, the *And-Or Tree Manager* continues on `next_alternative`.

`success`: this operation marks the current or-box as successful in its parent. The memory of the or-box is released. Next the parent and-box is checked. If all calls have reached `success`, space for the and-box is reclaimed and the operation is reentered for the upper or-box (*Success Propagation*). Otherwise, execution enters the `next_call` operation.

`fail`: this routine marks the current and-box as failed in its parent. All the assignments made by the and-box are removed, then space for the and-box is reclaimed. If all the alternatives for the parent or-box have failed, the operation is recursively called for the parent and-box (*Failure Propagation*). Otherwise, if there is one more alternative, execution moves to `unique_alternative`. If there are several alternatives, execution continues at `next_alternative`.

`next_call`: this operation searches for the next non-suspended call in the current and-box. If there is a ready call in the current and-box, *Reduction* is applied by setting the `PC` to the start of the call's code, and then execution jumps to the *Emulator*. Otherwise, if the and-box is not the root of the And-Or Tree, execution moves to `next_alternative`. If there is no ready call in the current and-box and if the current box is the root of the And-Or Tree, execution moves to the `select_work` operation.

`next_alternative`: this routine searches for the next non-suspended alternative in the current or-box to continue with the *Reduction* of alternatives. If there is no such alternative, execution jumps to `next_call`. Otherwise, if the alternative is in the `wake` state, execution moves to the `wake` operation, else execution sets the `PC` to the code for the alternative code, and enters the *Emulator*.

`unique_alternative`: this operation applies a *promotion* to the current and-box, since its parent or-box has a single alternative. All external variables are checked, as after promotion some external variables may have become local. As result, a wake signal is sent to all boxes suspended on this variables (*propagation*) that have their bindings promoted. If during the promotion of external variables unification fails, execution moves to the `fail` operation. If external variables still exist after the promotion, the and-box continues suspended, and execution moves to `next_call`. Otherwise, if the and-box is `suspended_on_end` execution can move to `success`. If goals are still left to execute, the *And-Or Tree Manager* marks the and-box as *running* and continues its execution by entering the `next_call` operation.

`wake`: this operation chooses a suspended and-box that has received a wake up signal (*propagation*). All externals variables are checked for changes: *environment synchronization*. The *environment synchronization* tests the compatibility (unify) of all the constrains imposed to external variables that are already bound. If an unification fails, execution for the box jumps to `fail`. If unification succeeds for a variable, the and-box suspension on that variable can be deleted. If bindings to externals variables still exist, execution continues to `select_work`. If no more restrictions to external variables are left, then either the and-box is `suspended_on_end` and we can move immediately to `success`. Otherwise, execution marks the and-box as *running* and continues its execution by entering the `next_call` operation.

`select_work`: this operation looks for work in the suspension list. If no extra work is available in the suspension list, execution will terminate. Otherwise, the `ABX` register is set to point to a box candidate for *splitting*. By default, the BEAM splits the leftmost suspended box. After splitting, one of the resulting and-boxes is waken, and its execution is restarted in `wake`.

6.5 And-Or Tree Manager Interaction with the Emulator

The *And-Or Tree Manager* interacts with the *Emulator* as illustrated in figure 6.10. In order to execute a query, the `next_alternative` first creates an or-box to store all possible alternative clauses. An alternative is then chosen and the execution passes to the *Emulator*, through the `explore_alternative` instruction. Following this, execution will run through a sequence of `get` and possibly some `unify` instructions that implement the unification of the head arguments. If this alternative is a fact clause, then the next, and also the last instruction executed by the emulator is the `proceed` instruction. If there are assignments to external variables, this instruction will move execution to the And-Or Tree Manager `suspend` operation. Otherwise execution moves to the `success` operation.

If the body of the alternative being explored is not empty, then instead of the `proceed` instruction we have a `prepare_calls` followed by a sequence of `put's` and `call_pred` instructions. The `prepare_calls` instruction creates an and-box to store the calls and jumps to `suspend` if there are assignments to external variables in the and-box, or `next_call` otherwise.

The `next_call` operation, chooses a call to execute (by default the leftmost), and

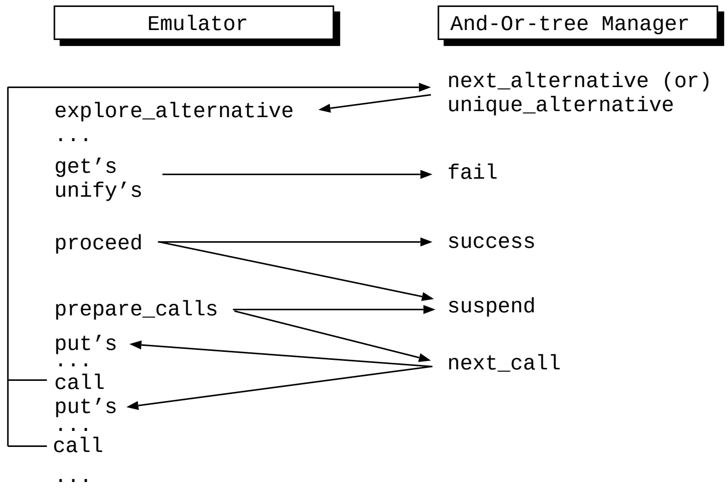

Figure 6.10: Connecting the And-Or Tree Manager with the Emulator.

jumps to the *Emulator* where it executes the `put` instructions followed by a `call_pred`. The `call_pred` instruction will then jump to the `next_alternative` operation in order to repeat the entire process.

This description, considers the straight forward execution case. Indeed, when running a normal program, the and-boxes will suspend when constraining external variables. Thus, a computational state with all and-boxes suspended is usual, and one must used the splitting rule (`select_work`) to create more deterministic work. The `select_work` operation selects a candidate to split (by default the leftmost suspended and-box). After splitting the computation can be restart by waking one of the resulting and-boxes. The `wake` operation will then perform an `environment check` to determine if the constrains being promoted are compatible with (possible) restrictions imposed by other and-boxes.

6.6 Implementation of the BEAM Splitting Rule

We have so far presented how we implement boxes and how most operations are used. In this section we present the implementation on the most complex rule for the BEAM: splitting. The split operation is executed whenever no deterministic operation is applicable to the And-Or Tree. Since it is the most complex rewrite on BEAM, we will explain its implementation through a small example illustrated in figure 6.11.

```
start:- a(X), b(...).     a(first).     a(second).
```

Figure 6.11 (a) presents execution of the goal `start`. More precisely we present BEAM execution state before splitting. We assume that the computation has suspended for both calls `a(X)` and `b(...)`. Note that the and-boxes (5) and (6) have tried to assign `first` and `second`, respectively, to `X` (local variable structure (3)), forcing both and-boxes to suspend. Both and-boxes are therefore in the suspension list for the local variable `X`.

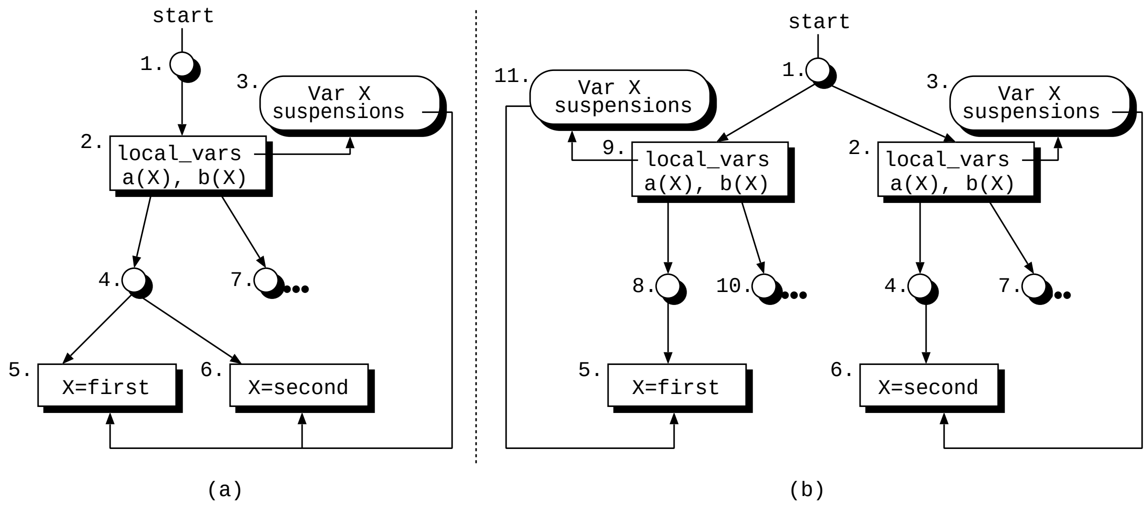

Figure 6.11: Example of Splitting.

Suppose that all computation is suspended with no deterministic rule applicable. Thus splitting is the only rule that can be used to restart the suspended computation. To start splitting, an or-box must be chosen (4), usually the parent of the leftmost suspended and-box (5). Figure 6.11 (b) presents the BEAM execution state after executing splitting. The algorithm starts by creating a new or-box (8). This or-box is initialized as having a single alternative. The leftmost suspended and-box (5) is then set as the unique alternative in this recently created or-box. The branch of this and-box on the older or-box (4) is then removed.

The key step in the algorithm is to replicate the and-box `a(X),b(X)`: we need to replicate the whole And-Or Tree rooted at `a`, and to do so we must traverse the whole tree rooted at and-box (2). This is performed through the `replicate_andbox` routine (see figure 6.12). The copy (9;10) is identical to the original except for the split or-box, which was already created in the previous step. Copying a subtree is a question of recursively copying or-boxes and and-boxes. The complexity arises when copying variables and terms. Our algorithm does as follows:

1. For all new and-boxes (9), a vector with new local variables is created (11). During the execution of this routine, old local variables are temporarily set to point to the newly created local variables so that references to the local variables in terms always refer to the new local variables.

2. All external variable structures available on the and-box (9) and children must be set to point to the new local variables, created in the previous step. For example, the external variable structure in (5) is now set to point to the newly created local variable structure (11). The suspensions list on the local variables must also be reset.

3. We must refresh the suspension lists on the local variables. For example, the suspensions list for the local variables structure (3) of figure 6.11(a), had two elements, the and-box (6) and (5). After the fork, the variable structure (3) was set only to point the and-box (6). The and-box (5) is now included in the suspension list of the newly created local variable (11).

4. Or-boxes are replicated through the `replicate_orbox` routine (see figure 6.12). This routine essentially creates a new or-box and then for each alternative replicates the children and-boxes.

The recently created subtree (9) is then added to the top or-box (1) as a new branch representing a new alternative, to the left of the older and-box (2).

After applying the splitting rule, some boxes are marked as awoken (5;6) as they may be ready for promotion. In the example, both and-boxes (5) and (6) will be promoted and will have their parent local variable `X` assigned to `first` and `second` respectively. The computation of `b(..)` on (10) and (7) will then be ready to proceed.

Our implementation of splitting is in fact quite similar to the copying based implementation used in AGENTS. There are two major differences:

- we do not copy all terms as the AKL does. The BEAM only copies the terms that have unbounded variables. Fully instantiated structures are shared between the *old* subtree and the *new* one.

```
do_splitting_box(or_box o)
{
  new_orbox=create_new_orbox();
  move_leftmost_alternative_from_to(o,new_orbox);
  new_andbox=replicate_andbox(o->parent);
  add_andbox_as_alternative(new_andbox,o->parent->parent);
  add_to(new_orbox,new_andbox);
}

replicate_andbox(andbox old_andbox)
{
  new_andbox=request_memory_for_andbox();
  copy_andbox_attributes_from_to(old_andbox,new_andbox);
  create_new_local_variables(new_andbox);
  set_old_local_variables_to_point_new_vars();
  copy_old_local_vars_value_to_new_local_vars();
  replicate_external_vars_bindings_using_newvars();

  for each call on old_andbox {
    new_andbox->call=replicate_orbox(old_andbox->call);
  }
  if (is_suspended(old_andbox)) add_to_suspensions_list(new_andbox);
  recover_local_variables_values(old_andbox);
  return(new_andbox);
}

replicate_orbox(orbox old_orbox)
{
  new_orbox=request_memory_for_orbox();
  copy_orbox_attributes_from_to(old_orbox,new_orbox);
  for each alternative on old_orbox {
    new_orbox->alternative=replicate_andbox(old_orbox->alternative);
  }
  return(new_orbox);
}
```

Figure 6.12: Splitting Algorithm.

- we do not copy the child and-box where the splitting is being performed. Instead, the and-box is moved directly to a new branch in the *new* subtree. Although all variables are checked so that references to the *old* subtree are changed to the *new* subtree. In the example the and-box (5) is not replicated during the splitting but it must reset its external reference to the new variable X (11).

6.7 Final Remarks

The splitting rule is obviously the most complex rule used in BEAM. All the main rules in BEAM perform direct changes in the And-Or Tree. We can easily conclude that during an computation the And-Or Tree will suffer a lot of changes. The memory usage is thus very important to achieve an efficient implementation. In the next chapter we will explain the memory usage of BEAM in more detail.

Chapter 7

BEAM: Memory Management

This chapter focus on memory management for the BEAM. Memory usage is a crucial issue on the BEAM. Whereas standard WAM implementations follow a stack discipline, the EAM control strategy is much more flexible. Therefore, WAM-based Prologs manage a stack where space can be recovered during backtracking, the BEAM must carefully detect the points where to recover space. How to do so is the subject of this chapter.

As described in chapter 6 the state of the BEAM, as represented in memory, is divided into two main areas: the *Code Space* and the *Global Memory*. The *Code Space* holds the database, including the compiled logic program, information on predicates, and the symbol-table. The BEAM uses this data-area in much the same way as traditional Prolog Systems. This area is largely static during execution.

The *Global Memory* stores the And-Or Tree that is manipulated during the execution of logic programs. This area further subdivides into the *Heap* and the *Box Memory*. The *Box Memory* stores boxes and variables. The *Heap* holds Prolog terms, such as lists and structures. Compound terms are represented through term copying. It is thus very similar to the WAM's *Heap*, with the difference that on the BEAM *Heap* memory can not be recovered at failure. A Garbage Collector is thus necessary to recover space in this area.

In this chapter we describe in detail how the BEAM recovers space. As we show next, we have two techniques: we can reuse space for pruned boxes, or we can garbage collect useless data.

7.1 Reusing Space in the And-Or Tree

The *Box Memory* must satisfy intensive requests for the creation of and-boxes, or-boxes, local variables, external references, and suspension lists. Objects are small and most, but not all, will have short living spans. Objects are created very frequently and minimizing allocation and deallocation overheads is crucial.

In order to address this problem, Prolog systems traditionally use a stack based discipline and rely on backtracking and garbage collection to recover space. Unfortunately, we cannot recover space through backtracking. Instead we explicit maintain liveness of data structures, and rely on a hybrid memory allocation algorithm to allocate space:

1. We keep a top of stack and an array of n buckets, one per sizes i, $1 \leq i \leq n$, all initially set to empty.

2. To allocate a block we check whether bucket i has a free block. Otherwise we allocate the block from the top of the stack.

3. To release a block of size i we add it to the front of the list for bucket i.

The BEAM is therefore able to recover all memory from boxes whenever they fail or succeed. Memory from failed boxes can obviously be recovered since they do not add any knowledge to the computation. Memory from succeeding boxes can also be recovered because the BEAM unification rules (see section 6.2.6), guarantee that and-box variables cannot reference variables within the box subtree, that is, younger box variables can reference variables in upper boxes, but not the other way around. Note that although local variables can reference compound terms in the *Heap*, *Heap* memory is in fact only released by the garbage collector when not in use.

We have chosen this scheme because it has a low overhead and because most requests tend to vary between a relative small number of sizes. To prove so, we have studied our algorithm with a small group of known benchmarks detailed on table 7.1. The first three benchmarks are deterministic, and the rest are non-deterministic.

Table 7.2 shows how our algorithm performs for the benchmark set. The *Memory requests* column shows how much memory was requested, the *Reuse* column shows the percentage of memory that was served from the free lists, and the *Memory used* shows

Name	Description
nreverse	naive reverse of a 30-element list.
qsort	quick-sort of a 50-element list using difference lists.
kkqueens	smart finder of the solutions for the n-queens problem.
ancestor	query a static database.
houses	logical puzzle based on constraints.
query	finds countries with approximately equal population density.
zebra	logical puzzle based on constraints.
send	the SEND+MORE=MONEY puzzle.
queens-9	finds all safe placements of 9-queens on $9 * 9$ chessboard.

Table 7.1: The benchmarks.

Benchmark	Memory requests	Reuses	Memory used
nreverse	40Kb	0.00%	40Kb
qsort	83Kb	56.51%	36Kb
kkqueens	40Mb	38.63%	25Kb
ancestor	32Kb	72.52%	9Kb
houses	552Kb	82.47%	97Kb
query	401Kb	66.50%	134Kb
zebra	2Mb	89.83%	208Kb
send	10Mb	89.95%	1Mb
queens-9	244Mb	88.53%	28Mb

Table 7.2: Box memory reuse.

the real size of memory used by each benchmark. The algorithm has no memory reuse for the deterministic `nreverse` benchmark, where the abstract machine itself reuses and-boxes through our implementation of the nested and-boxes simplification rule. On the other hand, we achieve reuse rates close to 90% in the `zebra`, `send`, and `queens-9` benchmarks. Hence, reuse is very effective in applications where the BEAM often prunes branches, such as in search applications, thus providing the advantages of backtracking to same extent.

To emphasize our point, note that in the `queens-9` benchmark, boxed memory had a total of 244Mb of memory requests, and about 89% of those requests were served from previously used memory. Thus we only required 28Mb of *Box Memory* throughout our computation.

In order to better understand these results, we further studied how memory is divided between boxes and variables. Figures 7.1 and 7.2 detail the BEAM's memory usage for the queens benchmark. Both graphics detail on the longevity (`X` axis) for the memory on boxes and variables. We measure time in tick, where each tick is a call to the memory allocator. We will assume the rate of calls to the memory allocator is constant throughout execution.

The first graphic (figure 7.1) illustrates how boxes and variables are requested and removed during the execution of the benchmark. From this graph we can conclude that the liveness of boxes is usually very short, and much shorter than the duration of variables. Indeed, a large number of boxes (1315 or 75%) has very short lives of less than 5% of the execution time. After this first set of boxes, there is a less marked tendency for the number of boxes to decrease as we increase longevity. At the end, only a mere 21 boxes survive more than 95% of the total execution time.

On the other hand, variables do seem to live longer. In contrast to the memory usage for boxes, there are very few variables to live less than 5% of the execution time. Moreover, most variables seem to live for about half of the computation. Of all 11 thousands variables created, only a bit over 400 survive 95% of the computation.

Figure 7.2 presents the same values, with a different representation. The values are accumulated. This graphic is interesting because it shows clearly that more than 70% of the memory for boxes lives less than 5% of the runtime. In contrast, variables are essentially equally distributed. In fact half of the variables live less than 50% of the runtime, and the other half live longer.

The two graphics show very clearly how an algorithm to recover box space is indeed important to an EAM implementation. In this search application, most boxes live for a very small time. Although variables do live longer, a substantial percentage should also be recoverable during the computation.

In practice, we have observed that *Box Memory* will grow through the computation even for search applications. This is largely because the BEAM releases memory eagerly, hence releasing smaller blocks at a time, and causing fragmentation. In contrast, the AGENTS implementation of AKL [70] releases boxes lazily, but at the cost of having to walk through dead boxes while choosing rules. A second problem is

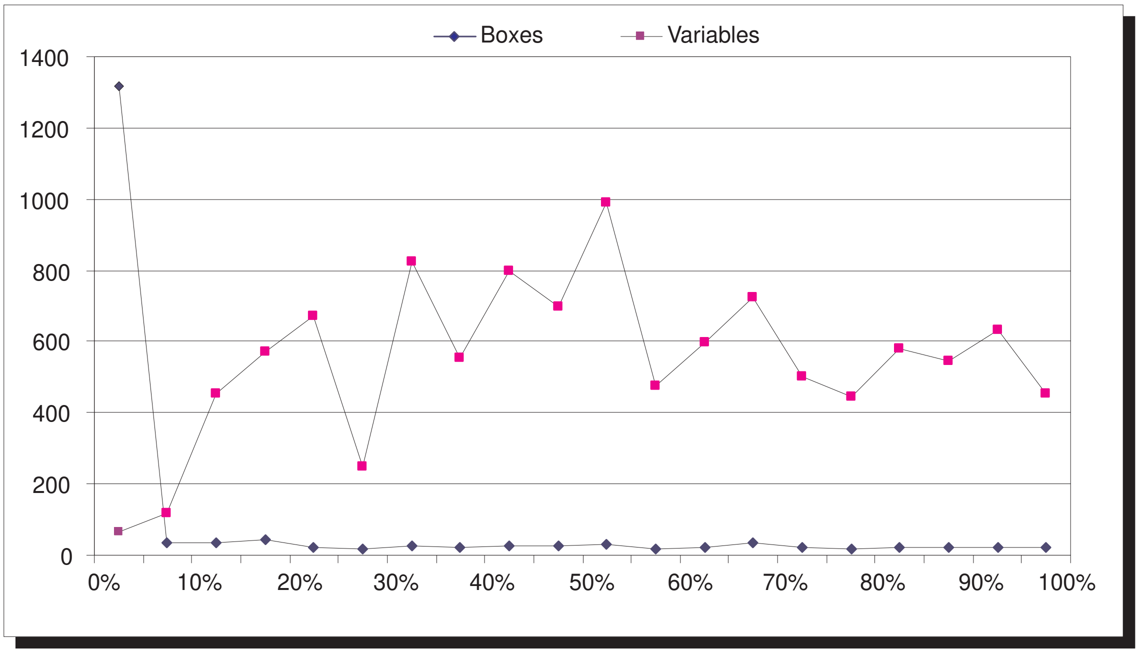

Figure 7.1: Boxes and variables longevity.

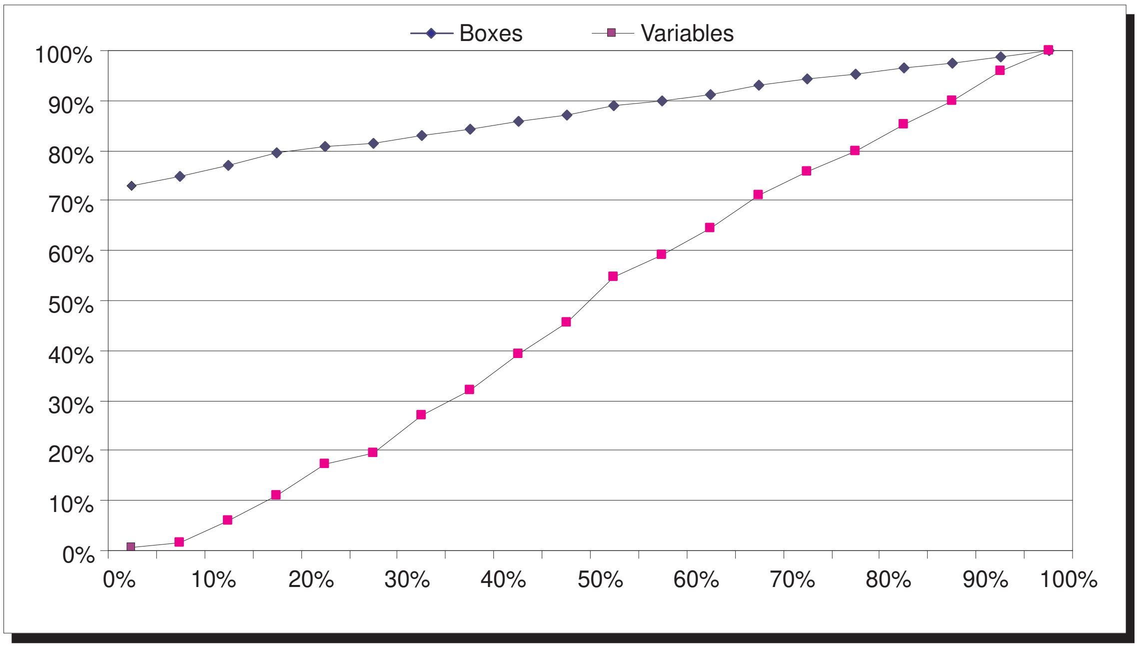

Figure 7.2: Boxes and variables longevity (accumulated).

that the BEAM asks for ever larger sized blocks when promoting and-boxes. We can coalesce blocks to increase available block size [41], but the price would be to increase overheads whilst still not addressing the problems for the *Heap* where objects tend to be very small. Instead, we have chosen to rely on a garbage collector to compact for both the *Heap* and the *Box Memory*. Our experimental data validates this idea: as we shall see, garbage collections were mainly caused by overflows in the *Heap*.

7.2 The Garbage Collector

We implemented a *copying* garbage collector [71] for the BEAM: living data structures are copied to a new memory area and the old memory area is released. The *Heap* memory is divided into two equal halves. The *Box Memory* is also divided into two parts with equal size, growing in the opposite direction. 7.3).

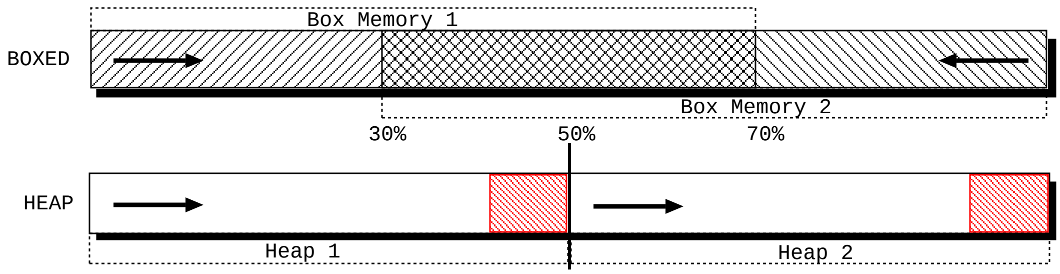

Figure 7.3: Memory organization.

The two parts of the *Box Memory* can share a common middle area. This allows for a more flexible use of memory. If the limit is exceeded then the garbage collection flag is set. By default the limit is set at 50%, but it can be changed for a more aggressive use of the *Box Memory*.

The *Heap* memory cannot use this scheme because the BEAM uses YAP builtins, and they expect the Heap to always grow upwards. Therefore the limit for the *Heap* cannot be exceeded. Instead we have a pre-defined limit-zone that, when reached, will activate the garbage collection mechanism by setting the garbage collector flag.

The garbage collection flag is periodically checked by the And-Or Tree manager to activate garbage collection. Garbage collection is activated for either areas if one of the areas becomes half-full.

The algorithm for the Garbage Collector essentially replicates the And-Or Tree. It therefore relies on algorithm for splitting described in section 6.6. The major difference is that the garbage collector does not need to recover the local variables values from the "old" memory after replicating an and-box. This is because when performing garbage collection the "old" And-Or Tree will never be used after the garbage collection, while when splitting, the "old" tree that is replicated is still used after the split operation. Thus the `recover_local_variables_values` algorithm is not executed by the garbage collector.

The initialization of the Garbage Collection algorithm is presented in Figure 7.4. The first procedure `prepare_pointers` sets all needed information so that the free half memory can be used. To do so, it sets the memory area pointers so that memory requests that occur during garbage collection will use the "new" half of memory. The `current_and_or_tree` always points to the root of the And-Or Tree. The `replicate_and_box` is a recursive function that will create a new tree similar to the one received as argument. The `replicate_and_box` calls the `replicate_or_box` for all and-box goals. The `replicate_or_box` creates a new or-box similar to the one received as argument, and recursively calls `replicate_and_box` for all the or-box alternatives. This process stops in unexplored boxes or in and-boxes that have no goals (facts). Thus, the garbage collector starts by replicating the root of the And-Or Tree and then follows a top-down-leftmost approach.

```
Init_Garbage_Collector() {

  prepare_pointers_for_the_other_half_mem_to_be_used();

  /* Note that the and-or tree always starts by a and-box on BEAM */
  starting_and_box=current_and_or_tree;
  new_andbox=replicate_and_box(starting_and_box);

  current_and_or_tree=new_andbox;
} /* the system can now continue execution using the new_tree */

/* the replicate_and_box & replicate_or_box routines are used
   from the Splitting code  */
```

Figure 7.4: Garbage Collector Algorithm.

We use a simple example, that is illustrated in figure 7.5 to explain the garbage collector algorithm. Consider three and-boxes each one having one local variable `X,Y,Z`. Moreover we consider `X` unbounded, `Y` bounded to `g(X)`, and `Z` bounded to

`[X,f(X)]`. Figure 7.5 illustrates the memory organization before executing the garbage collector.

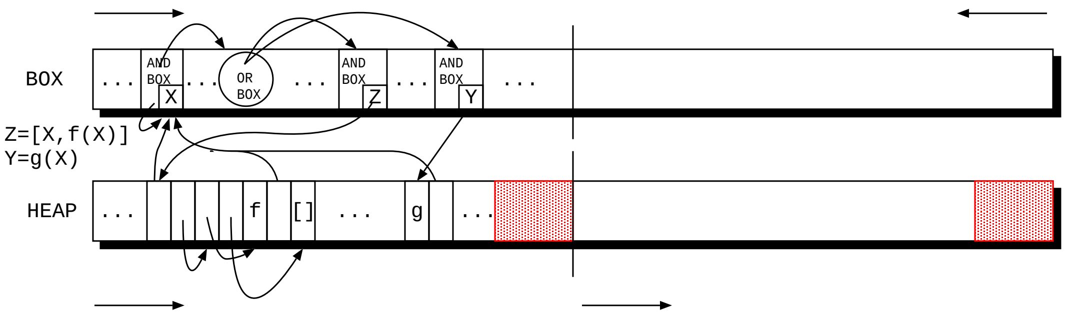

Figure 7.5: Memory before running the garbage collection.

In order to simplify our explanation the garbage collector algorithm is presented separately for the *Box Memory* and for the *Heap Memory*. We discuss the *Box Memory* first.

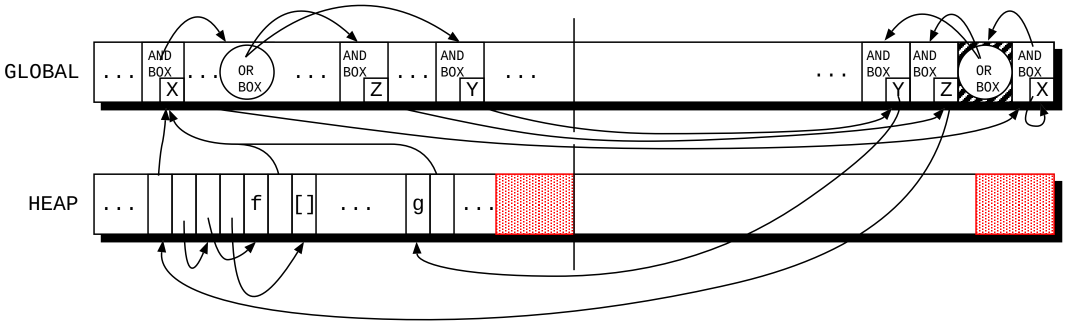

Figure 7.6: Garbage collection on the Box Memory.

When doing garbage collection on the the *Box Memory*, the live And-Or Tree is copied to the other half, as presented in the example in figure 7.6. Local variables are also replicated. Old local variables are set to point to the newly created local variables, so that references to a local variable will refer to the new local variables. Note that this is only possible because the BEAM guarantees that variables may only refer to variables in the same and-box or to variables in an upper and-box. Thus, if a variable points to a variable in another box, then that box is a parent and-box and was certainly replicated since the garbage collector follows a top-down approach.

When doing garbage collection on the *Heap*, the living data structures of the *Heap* are duplicated to the other free half of the *Heap* memory (see figure 7.7). The new local variables created on the *Box Memory* are set to point to the newly created structures on the *Heap*.

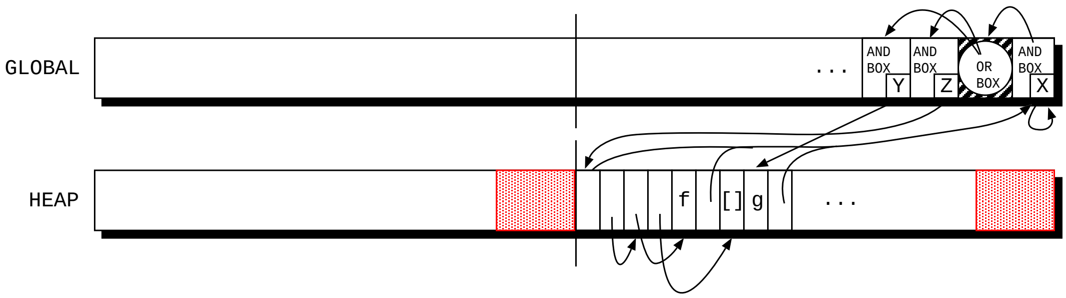

Figure 7.7: Memory organization after running the garbage collector.

After replicating the whole And-Or Tree, the "old" half of memory is finally released and the system continues its execution with the newly created tree.

The advantage of a *stop-and-copy* garbage collector implementation are that we can reuse the code for splitting and that the execution time for the garbage collection is proportional to the amount of data in use by the system. In contrast, the mark-and-sweep garbage collection algorithm has execution time proportional to the original stacks. The main disadvantage of a copying algorithm is that the system can only use part of the total memory available. We do not use generational garbage collection because the percentage of garbage cells is very high, as we shall discuss next.

Performance Analysis Table 7.3 shows the BEAM garbage collector running the `queens` benchmark with different memory configurations. The time is presented in milliseconds, and the number of invocations to the garbage collector is presented in brackets. The timings were measured running the benchmarks on a AMD Thunderbird 900Mhz (200Mhz FSB[1]) with 256Kb *on chip* cache, equipped with 512MB at 100Mhz SDRAM and running Red Hat Linux 7.1. We have chosen the `queens` benchmark for a deeper analysis because **(i)** it is a well-known benchmark, **(ii)** we can easily vary problem size by increasing the number of `queens` (see Appendix A); and **(iii)** it is

[1]FSB: Front Side Bus

MEM config	**queens=9 (all)**		**queens-15 (1st)**		**queens-16 (1st)**	
Heap + Boxed	Time	GC	Time	GC	Time	GC
1MB + 1MB	3,160	(272)	3,540	(323)	35,790	(3246)
2MB + 1MB	2,940	(132)	2,780	(137)	26,250	(1300)
4MB + 2MB	2,810	(65)	2,470	(64)	22,860	(592)
8MB + 4MB	2,780	(32)	2,330	(31)	21,300	(283)
16MB + 8MB	2,760	(16)	2,230	(15)	20,500	(138)
32MB + 16MB	2,730	(8)	2,170	(7)	20,240	(68)
64MB + 32MB	2,520	(4)	2,050	(3)	19,990	(34)
128MB + 64MB	2,380	(1)	1,920	(1)	19,810	(17)
256MB + 64MB	2,350	(0)	1,880	(0)	19,290	(8)

Table 7.3: BEAM with different memory configurations (time in milliseconds).

an example where the BEAM can outperform Prolog by limiting the search space: the YAP Prolog system [118] takes more than 24 hours to find the first solution to `queens-16`, whereas the BEAM finds the first solution in approximately 20 seconds. Note also that `queens-16` depends on garbage collection to run with the memory configurations used. Finally, we should remark that in this benchmark, garbage collections were always activated by *Heap* overflows.

Our main conclusion is that the impact of garbage collection on execution time is not very significant for large, and more reasonable, stack sizes. On the other hand, garbage collection does have a major effect on execution times for small stack sizes. This seems to be a "thrashing" effect: total execution times are very high for small stack sizes and then stabilize between 6MB and 24MB. Notice that, in all cases the actual number of garbage collections grows almost linearly as we decrease stack size: the number of garbage collection doubles if we halve the memory.

We can conclude that garbage collection is mostly useful at recovering *Heap* memory as we do not reuse *Heap* otherwise. Table 7.4 shows how effective garbage collection is on average for `queens-9`. Essentially, the working size for this benchmark is around 140KB for *Heap* and 220KB for box space. Increasing stack space allows delaying garbage collection and results in very good effectiveness for the *Heap* and for the *Box Memory*, even if the *Box Memory* has an 88.53% rate of memory reutilization for this benchmark (see Table 7.2). In general, we believe that these numbers indicate that, when well-tuned, garbage collection introduces an acceptable overhead for the

MEM config	**Heap**			**Box Space**		
	before	after	recovered	before	after	recovered
1MB + 1MB	364,646	141,280	61.26%	325,289	236,540	27.28%
2MB + 1MB	723,344	140,251	80.61%	399,376	234,461	41.29%
4MB + 2MB	1,445,654	136,437	90.56%	532,223	228,374	57.09%
8MB + 4MB	2,887,437	138,083	95.22%	790,503	230,830	70.80%
16MB + 8MB	5,772,233	135,985	97.64%	1,288,902	227,568	82.34%
32MB + 16MB	11,538,384	136,459	98.82%	2,277,527	226,985	90.03%
64MB + 32MB	23,074,209	134,186	99.42%	4,231,662	224,053	94.71%
128MB + 64MB	46,143,724	157,732	99.66%	8,099,458	258,029	96.81%

Table 7.4: Average Effect of Garbage Collection for `queens-9`

BEAM, and confirm our approach of using a simple allocator leaving compression for the garbage collector.

7.3 Simulation-Based Analysis

The previous numbers have shown that the BEAM is a memory intensive-system and that its performance may largely depend on the target machine's memory subsystem. To fully understand the characteristics of the BEAM we have used the execution-driven simulator *sim-cache*, a part of the *SimpleScalar* Tool Set v2.0 [17]. This simulator implements a functional cache simulator. Cache statistics are generated for a user-selected cache configuration, which may include up to two levels of instruction and data cache (with any levels unified). We study BEAM memory characteristics for a split Instruction-Data Level 1 caches (L1 I-cache and L1 D-cache) and unified Level 2 caches (L2 cache). In order to obtain complete data we studied memory performance for realistic different cache configurations. Cache configurations are represented trough triples <nsets>:<bsize>:<assoc>, where *nsets* is the number of sets in the cache, *bsize* the bus size of the cache, and *assoc* the associativity of the cache. For the sake of completeness we include similar data for the YAP Prolog system (version 4.3.19).

The number of L1-cache accesses are detailed in Table 7.5. The simulation reveled that, for both BEAM and YAP, the L1 I-cache has approximately two times more accesses than the L1 D-cache. The BEAM has less accesses to the level-1 cache because the BEAM is able to reduce the search space for the queens benchmark, wielding a smaller number of instructions executed and a reduced number of accesses to memory.

	L1 - Inst.	**L1 - Data**
YAP	2,800	1,300
BEAM	2,050	900

Table 7.5: SimpleScalar L1-cache accesses for queens-9 (in Millions).

The main miss-rates for the level-1 cache are presented in Table 7.6. We have used two different configurations that reflect those used in the recent family of Intel and AMD processors. The Intel processors have two level-1 caches: one has 16Kb cache for instructions plus another 16Kb cache for data. Both are configured with a 4 way associativity on a 32 bits bus. The AMD processors have 64Kb cache for instructions plus another 64Kb cache for data configured with a 2 way associativity on a 64 bits bus.

The BEAM has a very good miss rate for the L1 I-cache for the configuration correspondent to the AMD processors (larger cache). The low miss-rate indicates that even if the BEAM machinery is more complex than Prolog's we can still obtain very good performance from a 64KB L1 I-cache, which seems to become the standard for modern machines (for example the AMD Duron and Thunderbird processors and recent RISC processors like Sun UltraSparc and Alpha). Interestingly, on the Intel processors, the BEAM has a lower miss rate for small memory configurations, probably due to the higher locality of the garbage collector which is called less often for larger memory configurations.

Figure 7.8 presents the BEAM miss rates for different configurations of the L1 I-cache. The graph includes results from the Intel and AMD L1 I-cache configurations, plus four different configurations for a 32Kb cache. As shown in Figure 7.8 the L1 I-cache benefits directly from the increased size. When comparing the different 32Kb configurations it is also possible to conclude that the BEAM benefits from moving from a 32 bits to a 64 bits block size, and that also benefits from moving from direct to two or four-way associative.

The numbers detailed in Table 7.6 show a somewhat different story for the L1 D-cache. Whereas backtracking allows Prolog to fit its working set for this benchmark well into a 16KB cache the BEAM miss-rates for the L1 D-cache vary between around 3% to 4% for a small 16KB L1 D-cache to a much better 1% for a 64KB L1 D-cache.

		Intel PIII Processors		AMD Athlon Processors	
Cache config.		**inst-16K 128:32:4**	**data-16K 128:32:4**	**inst-64K 512:64:2**	**data-64K 512:64:2**
YAP		0.225%	0.001%	0.002%	<0.001%
BEAM	1MB	**0.900%**	3.427%	0.017%	0.835%
	2MB	0.933%	3.381%	0.016%	0.769%
	4MB	0.948%	**3.379%**	0.015%	0.760%
	8MB	0.956%	3.406%	0.015%	**0.755%**
	16MB	0.960%	3.449%	0.014%	0.777%
	32MB	0.962%	3.482%	0.014%	0.809%
	64MB	0.963%	3.524%	0.014%	0.825%
	128MB	0.963%	3.585%	0.014%	0.847%
	256MB	0.958%	3.664%	**0.014%**	0.940%

Table 7.6: SimpleScalar L1 cache misses simulations for queens 9 benchmark.

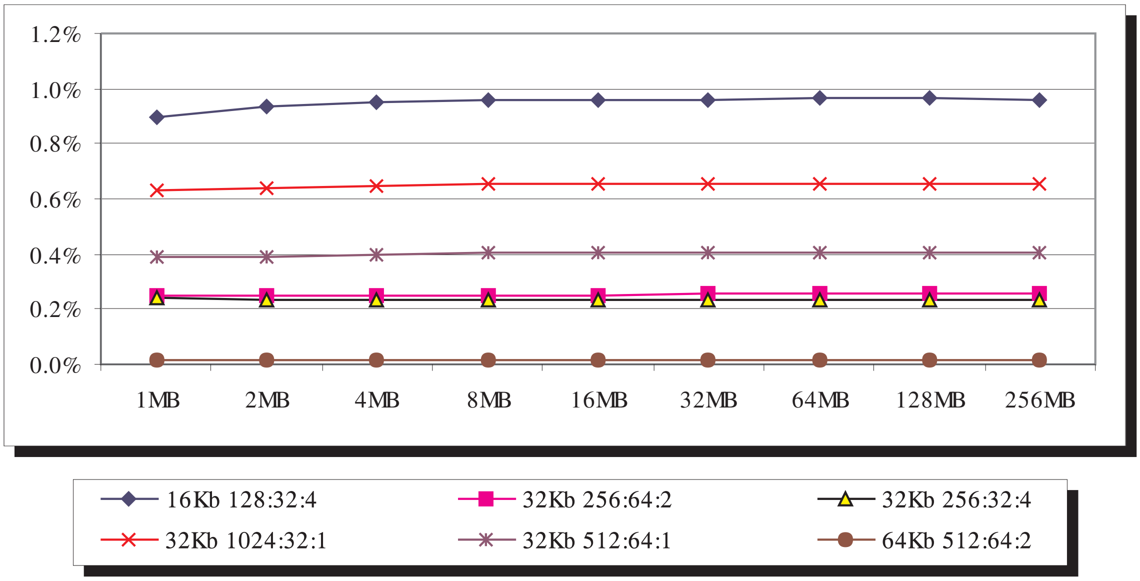

Figure 7.8: Miss rates for the L1 I-Cache.

The interplay between memory area size and miss rates shows that all configurations have slightly better miss-rates at around 4MB and 8MB. This is arguably the optimal point between worsening miss-rates through excessive garbage collection or through a larger memory configuration. Interestingly, the L1 D-cache miss-rate for the smallest memory configuration is better than the L1 D-cache miss-rate for the largest memory configuration.

Figure 7.9 presents the BEAM miss rates for different configurations of the L1 D-cache. Once again the graph shows that, similar to the L1 I-cache, the L1 D-cache benefits directly from the increased size. The results for the 32KB size show that the L1 D-cache benefits substantially from a 64 bits block size. Moreover, a 2-way 64 bit block sized cache, has better performance than the 32 bit 4-way version.

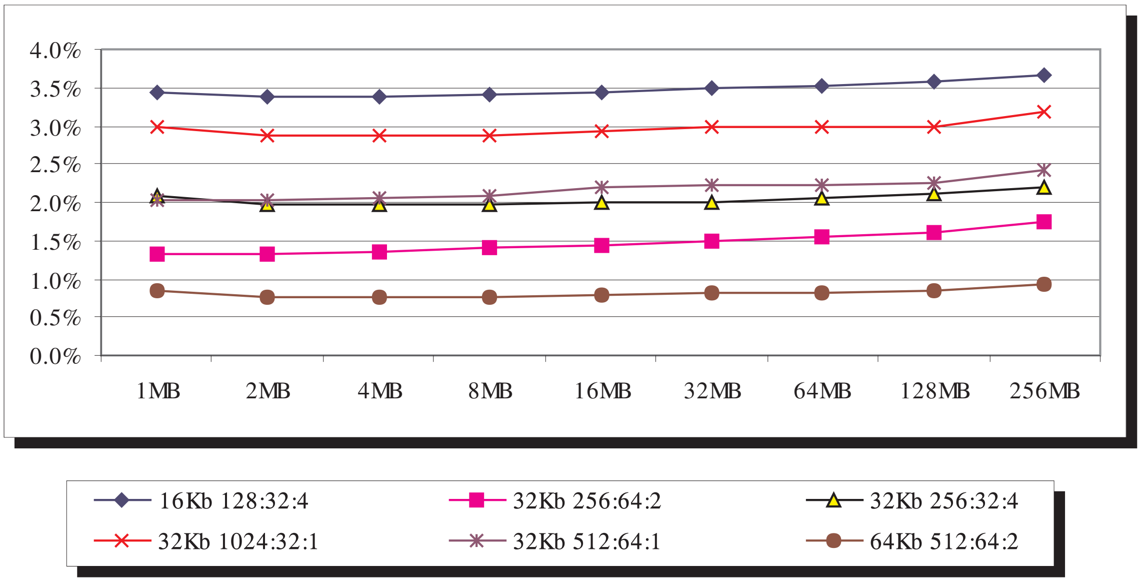

Figure 7.9: Miss-rates for the L1 D-Cache.

The number of L2-cache accesses is shown in Table 7.7. The values for the BEAM represent the medium number of cache accesses between the different BEAM's versions. The number of L2-cache accesses on the Intel processors is much larger than on AMD processors since these processors have different L1-cache configurations. The level-1 cache configuration clearly affects the number of accesses to the level-2 cache, as the level-2 cache is only accessed when data is not available in the level-1 cache. Thus, the larger L1-cache used on the AMD will mean a lower number of accesses to the second level cache.

		L2 - Unified
Intel Processors	YAP	6,500
	BEAM	75,000
AMD Processors	YAP	53
	BEAM	15,000

Table 7.7: SimpleScalar L2-Cache accesses for queens-9 (in Thousands).

The level-2 cache shows a completely different story for the two systems. YAP has a small number of level-2 cache accesses when compared to the BEAM, resulting from the very high and effective success rate for the level-1 cache. The AMD configuration has five times less accesses to the L2-cache when compared to the Intel configuration. Amazingly, on the AMD processors, YAP only has 53 thousand accesses to the L2-cache, which is almost negligible.

The main miss-rates for the level-2 cache are presented in Table 7.8. This cache is configured as a unified cache (stores both instructions and data) similar to the configurations used in the recent family of Intel and AMD processors. Once again several configurations are presented, including the AMD Thunderbird processor, and Intel PIII family of processors, from the Intel Celeron to the Intel Xeon with 2Mb of cache.

		AMD	**Intel PIII Processors**			
Cache config.		**256Kb 256:64:16**	**128Kb 64:256:8**	**256Kb 128:256:8**	**512Kb 256:256:8**	**2048Kb 1024:256:8**
YAP		7.5%	2.8%	2.4%	2.3%	2.3%
BEAM	1MB	32.0%	12.0%	8.4%	7.8%	**0.1%**
	2MB	27.0%	**11.9%**	7.2%	6.5%	3.8%
	4MB	23.4%	12.1%	6.3%	5.7%	5.3%
	8MB	21.5%	12.5%	5.9%	5.2%	4.9%
	16MB	20.0%	12.6%	5.6%	4.8%	4.5%
	32MB	18.8%	12.8%	5.4%	4.6%	4.3%
	64MB	18.3%	13.2%	5.4%	4.4%	4.1%
	128MB	17.7%	13.6%	5.3%	4.3%	4.0%
	256MB	**16.0%**	13.2%	**5.0%**	**3.9%**	3.6%

Table 7.8: SimpleScalar L2 cache misses simulations for queens 9 benchmark.

Miss-rates for the L2-cache are much worse than for the L1 caches. Still, note that the number of L2-cache accesses is much less than the number of L1-cache accesses (L1-cache accesses are presented in millions while L2-cache accesses are presented in thousands). Thus 10% of L2-cache misses is indeed less than 1% of L1-cache misses. Also note that the AMD L2-cache misses is not directly comparable to the Intel processors because the number of L2-cache accesses is very different for the two systems.

Figure 7.10 shows graphically the results found on table 7.8. The high number of cache-misses suggests that whereas the L1 D-cache could capture a locality plateau, probably stemming from the inner data-structures in the BEAM, the L2 cache, except for the 2MB configuration, cannot catch a secondary plateau, probably because of the large memory footprint of the BEAM. Up to the larger memory configurations we can observe that there is a inverse correlation between miss-rate and memory area size, suggesting that the garbage collector is the main culprit for the increased miss-rates. The results are also confirmed for the 2MB cache configuration: in this case we can fit the whole memory of BEAM using 1Mb for Heap and 1Mb for boxes in the L2 cache, and the miss rates achieve the best performance.

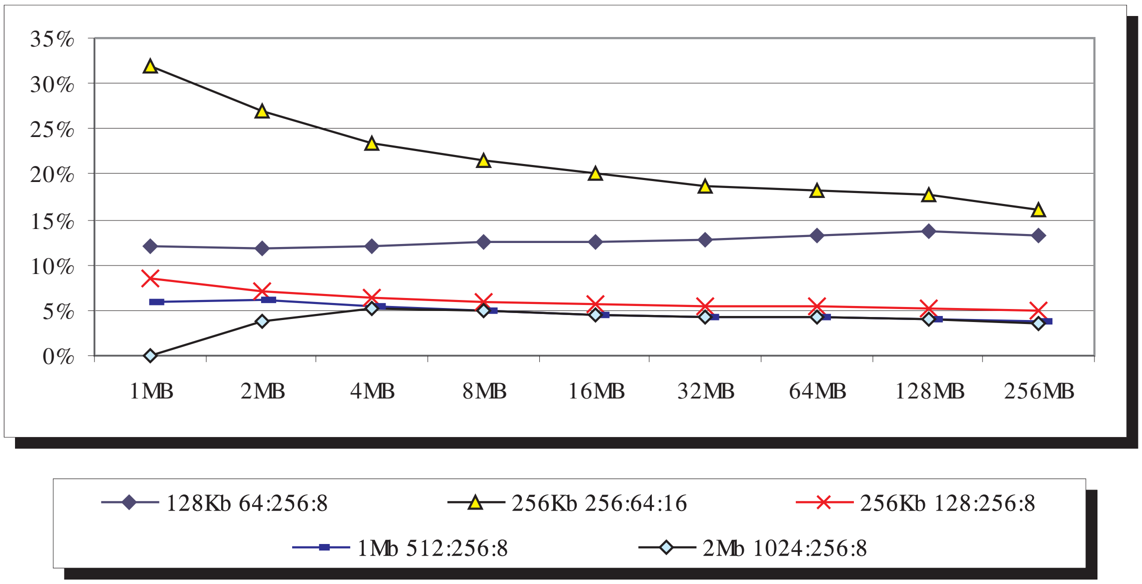

Figure 7.10: Miss-rates for the L2 Cache.

7.4 Final Remarks

In this chapter we have presented a detailed study of the memory management techniques used for our sequential implementation of the EAM, the BEAM.

We have shown that a simple, low overhead, allocation mechanism can be quite effective in reducing the need for frequent garbage collection in the And-Or Tree. Even so, garbage collection is required. Our benchmark results, confirmed through simulation, show that garbage collection can have a large impact on execution time, and that is partly caused by bad miss-rates on the data caches. The solution is either to guarantee that our workspace will fit into the L2 cache, or to use a larger workspace that would avoid the need for frequent garbage collections.

Chapter 8

BEAM: Performance Analysis

This chapter presents the performance results of the prototype BEAM system and compares our system with the latest versions of SICStus Prolog, YAP, Andorra-I and AKL (Agents). An in-depth study of the BEAM memory usage was presented in the previous chapter, and thus, this chapter focus exclusively on the performance of the BEAM system versus other implementations.

8.1 Methodology for the Analysis

For the analysis of the BEAM performance we compare it with the following systems:

- **SICStus Prolog 3.8.6** [21]: is a state-of-the-art, ISO standard compliant, Prolog system developed at the SICS (Swedish Institute of Computer Science). It is a commercial version and is currently the most well known Prolog system. All benchmarks were executed using compiled emulated code.

- **YAP 4.3.19** [38]: is another state-of-the-art emulated Prolog system that was developed at University of Porto. This system is widely regarded as the fastest Prolog system available for the PC Platform.

- **YAP 98**: is an older version of the YAP Prolog. We use it in our comparison because the BEAM was implemented on top of it.

- **Andorra-I v1.14** [116]: is an implementation of the Basic Andorra Model that exploits or-parallelism and determinate dependent and-parallelism while fully supporting Prolog. We have used the sequential version for the comparison. All benchmarks were pre-compiled by the Andorra-I preprocessor before execution.

- **AKL Agents v1.0** [70]: is a sequential Andorra Kernel Language implementation. The language was designed by Sverker Janson and Seif Haridi. AGENTS was developed by Johan Bevemyr and others, at SICS, Sweden. This system follows an execution scheme that is similar to BEAM's but has the control intrinsic in the language. All benchmarks were rewritten to the AKL language before compiling and executing them on the Agents system.

We have used a representative group of well-known benchmarks. For each benchmark we present the best execution time from a series of ten runs. The runtime is presented for all systems in milliseconds. The timings were measured running the benchmarks on an AMD Thunderbird 900Mhz (200Mhz FSB) with 256Kb *on chip* cache, equipped with 512MB at 100Mhz SDRAM and running Red Hat Linux 7.1. The BEAM was configured with 64Mb of *Heap* plus 32Mb of *Box Memory*. Benchmark code is presented in appendix A.

One important consideration to keep in mind when reading this chapter is that the BEAM was created to prove the feasibility of the EAM and it is still in a early prototypal stage, thus including important limitations. One major issue with the BEAM is builtin support. The BEAM uses the unmodified builtins from YAP98. This has several limitations, as these builtins do not inform the BEAM about bindings to local variables. The builtins are also not prepared to deal with several collateral effects that can emerge from the different execution order of BEAM. For example, delaying instead of aborting if a variable should be used in arithmetic operations. These issues limit the number of benchmarks we can run with the BEAM.

8.2 The Benchmark Programs

Table 8.1 gives a small description of the benchmarks used in this chapter. The selected group of benchmarks is composed by well known test programs used within the Prolog scientific community.

	Name	Description
	cal	last 10000 FoolsDays arithmetic benchmark.
	deriv	symbolically differentiates four functions of a single variable.
D	**qsort**	quick-sort of a 50-element list using difference lists.
E	**serialise**	calculate serial numbers of a list.
T	**reverse**	smart reverse of a 1000-element list.
	nreverse	naive reverse of a 1000-element list.
	kkqueens	smart finder of the solutions for the n-queens problem.
	tak	heavily recursive with lots of simple integer arithmetic.
	ancestor	query a static database.
N	**houses**	logical puzzle based on constraints.
O	**query**	finds countries with approximately equal population density.
N	**zebra**	logical puzzle based on constraints.
	puzzle4x4	finds a solution for a quadratic puzzle.
D	**send**	the SEND+MORE=MONEY puzzle.
E	**queens**	finds all safe placements of n-queens on $n * n$ chessboard.
T	**check_list**	list checker that verifies if duplicate elements exist.
	ppuzzle	naive generation and test valid paths in a quadratic puzzle.

Table 8.1: The benchmarks.

The benchmarks are divided into two classes: deterministic and non-deterministic. The non-deterministic benchmarks are further subdivided into two groups: benchmarks that do not benefit from the Andorra rule and benchmarks where the Andorra rule allows the search space to be reduced.

8.3 Performance on Deterministic Applications

Table 8.2 and Table 8.3 show how the BEAM performs versus Andorra-I, AGENTS, and several Prolog systems for deterministic applications. Neither the BEAM nor AGENTS perform splitting, and Andorra-I always executes deterministically. Prolog systems may create choice points.

The YAP and SICStus Prolog systems are recognized as the fastest Prolog systems on the x86 architectures. The difference between Yap98 (on which the BEAM is based)

Benchs.	BEAM	AGENTS	Andorra-I	YAP		SICStus
				98	4.3	3.8.6
cal	0.010	0.020	0.020	0.009	0.005	0.012
deriv	0.060	0.380	0.050	0.019	0.013	0.024
qsort	0.49	1.07	0.46	0.18	0.11	0.38
serialise	0.23	0.69	0.38	0.08	0.06	0.18
reverse_1000	0.30	1.60	0.37	0.13	0.12	0.17
nreverse_1000	130	780	140	50	40	60
kkqueens	240	460	240	127	40	70
tak	110	154	140	60	30	50

Table 8.2: Deterministic benchmarks (time in milliseconds).

Benchs.	BEAM	AGENTS	Andorra-I	YAP		SICStus
				98	4.3	3.8.6
cal	120%	60%	60%	133%	240%	100%
deriv	40%	6%	48%	126%	185%	100%
qsort	78%	36%	83%	211%	345%	100%
serialise	78%	26%	47%	175%	300%	100%
reverse_1000	57%	11%	46%	131%	142%	100%
nreverse_1000	46%	8%	43%	120%	150%	100%
kkqueens	29%	15%	29%	55%	175%	100%
tak	45%	32%	36%	83%	167%	100%
average	62%	24%	49%	136%	213%	100%

Table 8.3: Deterministic benchmarks relative speed comparison to SICStus.

and Yap4.3 shows that there is scope for improvement even for Prolog systems. These improvements should also benefit the BEAM. Comparing with the BEAM, Yap4.3 is between 2 and 5 times faster than the BEAM. SICStus Prolog and Yap98 are a bit less fast. This is quite a good result for the BEAM, considering the extra complexity of the Extended Andorra Model.

The BEAM tends to perform better than the AGENTS especially on tail-recursive computations. We believe this is because the BEAM has special rules for performing tail-recursive computation that avoid creating intermediate or-boxes and and-boxes. The results are especially good for the BEAM considering that the BEAM does not need any explicit control on these benchmarks. On the other hand, AGENTS benefits from extra control to run the benchmarks deterministically.

Performance of the BEAM is very close to the performance of Andorra-I. Andorra-I beats BEAM on two benchmarks: `deriv` and `qsort`. This seems to depend on determinacy detection performed by the Andorra-I preprocessor. Consider the following code from the `qsort` benchmark:

```
partition([X|L],Y,[X|L1],L2) :- X =< Y, partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :- X  > Y, partition(L,Y,L1,L2).
```

Andorra-I classifies this code as deterministic, and never creates a choice point. Unlike Andorra-I, BEAM does not have a pre-compilation with determinacy analysis to classify this predicate as deterministic. Thus, the sophisticated determinacy code in Andorra-I can limit the overheads that the BEAM has to go through by creating unnecessary and-boxes. For better understanding these overheads that BEAM suffers, consider the two possible cases when running the `partition` predicate:

- `X` $<=$ `Y`: the BEAM creates an or-box with two alternatives. It performs the head unification for the first alternative and it succeeds executing the test comparing `X` with `Y`. Execution then immediately suspends, because head unification generates bindings to variables external to the box. Execution then continues with the second alternative that will fail when comparing `X` to `Y`. This failure makes the first alternative the unique alternative in the or-box, and a promotion will occur allowing the suspended computation to resume.

- `X` > `Y`: the BEAM creates an or-box with two alternatives. The first alternative will fail when comparing `X` and `Y`. This failure makes the second alternative unique in its or-box. Promotion thus will occur, allowing the second alternative to run deterministically without suspending.

Concluding, the BEAM deterministic performance seems to be somewhat better than AGENTS and equivalent to Andorra-I, although in some code the BEAM still has greater overheads than Andorra-I.

8.4 Performance on Non-Deterministic Applications

Comparing the different systems for non-deterministic benchmarks is hard, since the search spaces may be quite different for Prolog, BEAM, AGENTS, and Andorra-I. We will consider two classes of non-deterministic applications. First, we consider applications where the Andorra Model does not provide significant improvements to the search space. Note that in general one would not be terribly interested in the BEAM for these applications: first, splitting is very expensive and second, or-parallelism can also be quite effectively exploited in Prolog. Then, we will consider examples where the Andorra rule reduces, very significantly, the search space.

Table 8.4 and table 8.5 show a set of five non-deterministic benchmarks. The number of splits for the BEAM and AGENTS and the number of non-deterministic steps for Andorra-I for this set of benchmarks is presented in table 8.6. We consider two versions of the BEAM. The default version, delays splitting until no other rules are applied. The *ES* version uses eager splitting. In this version splitting on producer goals is performed immediately. Producer goals are identified through the use of a annotation inserted in the Prolog program. Eager splitting makes our computation rule closer to that of Prolog. Note that to attain good results with eager splitting, BEAM depends on the user to identify the producer goals. Ideally, we would prefer to use compile time analysis to detect the producer goals.

Both BEAM and AGENTS perform badly in these benchmarks as we were expecting. Andorra-I is not as fast as the standard Prolog implementation but it still performs reasonably. Interestingly, both BEAM and AGENTS perform less splits then non-deterministic steps performed by Andorra-I. These results are the proof that the

Benchs.	**BEAM**		**AGENTS**	**AND.-I**	**YAP**		**SICStus**
	Default	**ES**			**98**	**4.3**	**3.8.6**
`ancestor`	NA	0.29	0.7	0.19	0.02	0.018	0.059
`houses`	5.0	4.1	13	1.4	0.6	0.5	1.1
`query`	34	5.5	45	2.1	0.73	0.35	0.97
`zebra`	100	24	54	46	13.5	10.5	19.4
`puzzle4x4`	2,730	-	2,850	960	320	270	360

Table 8.4: Non-deterministic benchmarks (time in milliseconds).

Benchs.	**BEAM**		**AGENTS**	**AND.-I**	**YAP**		**SICStus**
	Default	**ES**			**98**	**4.3**	**3.8.6**
`ancestor`	NA	20%	8%	31%	295%	328%	100%
`houses`	22%	27%	8%	79%	183%	220%	100%
`query`	3%	18%	2%	46%	133%	277%	100%
`zebra`	19%	81%	36%	42%	144%	185%	100%
`puzzle4x4`	13%	-	13%	38%	113%	133%	100%
`average`	14%	37%	13%	47%	173%	229%	100%

Table 8.5: Non-deterministic benchmarks relative speed comparison to SICStus.

Benchs.	**BEAM**		**AGENTS**	**ANDORRA-I**
	Default	**ES**		
`ancestor`	NA	30	73	75
`houses`	49	68	236	237
`query`	624	624	624	626
`zebra`	626	294	493	3,631
`puzzle4x4`	53,350	-	53,350	53,351

Table 8.6: Number of splits for BEAM/AGENTS and non-det. steps for And-I.

splitting rule is very expensive.

BEAM performance shows a good improvement with the use of eager splitting. This set of benchmarks cover all major cases that can occur when using eager splitting on BEAM. The `ancestor` benchmark [56], is one example where producers avoid a situation where the EAM may loop. The `houses` benchmark is an interesting case where, although eager splitting increases the number of splits, performance still has a slight improvement. The BEAM has a lower number of splits in this benchmark because the BEAM is able to delay splitting till *success* or *failure propagation* whereas the AGENTS has control dependent on the guard operators. This example shows that splitting earlier with less data to copy may have advantages in some cases. Similarly, using eager splitting on the `query` benchmark, although does not change the number of splits performed, has a huge improvement on system performance. The `zebra` benchmark is another demonstration of the impact of eager-splitting in the EAM. In this example just defining the producer dramatically cuts the search space and achieves a performance much better than Andorra-I. Moreover, the good execution time when compared with Prolog indicates that the system is now actually reducing the search space. Finally, in the `puzzle` benchmark the main goal suspends during the head unification and is forced to perform splitting immediately. Thus eager splitting does not have any effect on this benchmark.

To better understand the effects of the splitting rule, and what implications eager splitting can have on the computation tree, please consider the example illustrated on figure 8.1.

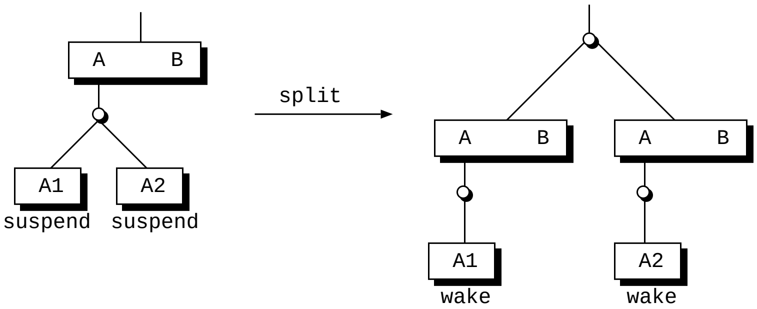

Figure 8.1: Splitting effects.

There are two goals, the producer A and the consumer B. The BEAM allows two methods to determine when to split on the goal A:

- **default rule**: the split on A will only be performed when all the computation on B suspends. If A is a producer, then there is a risk of having the EAM creating speculative work on B (or even lead to non-termination). Moreover, when splitting on A, all the And-Or Tree created on B will be copied. This copy can obviously bring a large penalty to the execution time.

- **with eager splitting on A**: the split on A will be performed before starting execution of B. The split will be simpler since there will be no data associated with the goal B to replicate. The disadvantage is that this goal after the splitting is totally unexplored in two and-boxes of the tree, and thus there is duplication of work.

We next use an example to remind the reader of the issues involved in executing a split. Consider the following Prolog program:

```
p(X):- f(X).      f(a).
p(X):- g(X).      g(b).
```

and the query `?- p(Z), Z=b.` Standard Prolog systems execute this query by creating a choice point for `p` and then executing the first alternative that binds `Z=a`. Thus, the second call in query fails. Prolog then backtracks and executes the second alternative for `p` that binds `Z=b` and thus the query succeeds. The BEAM and AGENTS execute this query deterministically. The BEAM suspends on `f` and `g` for trying to bind `Z` non-deterministically. The second goal in the query is then executed and binds deterministically `Z=b`. As result the goals `f` and `g` are awaken. The first one fails, and the second succeeds. Thus, BEAM for this query does not perform any split. In contrast, Andorra-I has to perform a non-deterministic step in `p` as it cannot decide deterministically between the two `p` alternatives. Although the execution strategies followed by the different systems are very different, the performance is very similar for all when executing the query `?- p(Z), Z=b`.

Consider now the query `?- p(Z) : Z=b.` (`:` stands for wait, the sequential conjunction). For the standard Prolog systems, this query is equivalent to the previous one, since the Prolog execution is by default sequential. The Andorra-I will execute this query as a normal Prolog system, again performing one non-deterministic step on `p`. In contrast, the BEAM and AGENTS have a completely different behavior from the previous query. The sequential conjunction will make the two systems split on `p` before executing `Z=b`. This query will thus suffer an overhead inherent to the split. Although the two queries seem very alike BEAM and AGENTS take about 3 to 4 times more time to execute the last query just due to the splitting.

8.4.1 Improving the Search Space

The main benefit of the BEAM is in applications where we can significantly improve the search space. Such applications may be pure logic programs, or may be applications that take advantage of the concurrency inherent to the Andorra Model. We consider four examples. The `send_more_money` benchmark is a well-known example of a declarative program that performs badly in Prolog. A set of Prolog benchmarks would not be complete without experimenting with a naive solution for the `queens` problem. And finally two benchmarks that process lists, the `check_list` and the `ppuzzle`. Each list element is a pair with the form $p(X, Y)$ representing a position in a $n * n$ matrix. The `check_list` benchmark succeeds if an input list does not hold duplicate elements. The `ppuzzle` generates all lists with all possible combinations of the different elements in an $n * n$ matrix, and validates those that obey to certain predefined conditions.

Results are shown in table 8.7 and table 8.8. The `send-more-money` and the `queens` benchmarks are quite interesting because the BEAM without extra controls performs the same number of splits as AGENTS and it still has slightly better performance. Performance is three order of magnitude faster than Prolog's. These benchmarks are also interesting in that they show a situation where the more Prolog-like Andorra-I, actually obtains the best results. Although performing the same (or a few more) non-deterministic steps than BEAM and AGENTS, Andorra-I is faster, thus proving, once again, that splitting is a very expensive operation.

The `check_list` and the `ppuzzle` benchmarks are an example where the EAM benefits from allowing non-deterministic goals to execute as long as they do not bind external variables. On the other hand, Andorra-I is limited on this benchmark by the non-determinism of the main predicates. On `check_list` Andorra-I has a search space similar to Prolog's, while in the `ppuzzle` Andorra-I, although better than Prolog, still has a larger search space than BEAM and AGENTS. Other important fact is that the

Benchs.	BEAM	AGENTS	ANDORRA-I	YAP 4.3
`send_money` 1st	14	20	1.7	17,820
`send_money` all	82	120	9.0	129,840
`queens-9`	12	20	1.6	30
`queens-10`	40	80	3.9	230
`queens-11`	30	50	2.3	1,650
`queens-12`	170	340	11.6	16,780
`queens-13`	70	190	6.2	171,980
`queens-14`	2,170	3,660	110	2,174,740
`queens-15`	1,900	3,440	90	28,281,520
`queens-16`	19,990	26,820	720	>24 hours
`queens-17`	12,960	18,880	430	...
`queens-18`	117,760	149,710	3,480	
`queens-19`	8,420	12,180	230	
`queens-20`	787,030	962,550	19,460	
`check_list-8`	0.23	0.71	1,880	310
`check_list-9`	0.24	0.90	9,590	1,570
`check_list-10`	0.26	1.13	343,680	56,180
`check_list-11`	0.29	1.47	11,221,800	1,826,400
`check_list-15`	0.49	3.44	>24 hours	>24 hours
`ppuzzle-A`	22	28	246,460	>24 hours
`ppuzzle-B`	60	62	>24 hours	>24 hours
`ppuzzle-C` 1st	7	10	3,747,980	>24 hours

Table 8.7: Reduced search benchmarks (time in milliseconds).

Benchs.	BEAM	AGENTS	ANDORRA-I
`send_money` 1st	277	277	325
`send_money` all	1,564	1,564	2,007
`queens-9`	129	129	130
`queens-10`	364	364	364
`queens-11`	212	212	212
`queens-12`	1,109	1,109	1,110
`queens-13`	522	522	523
`queens-14`	9,046	9,046	9,046
`queens-15`	7,054	7,054	7,055
`queens-16`	52,617	52,617	52,617
`queens-17`	31,210	31,210	31,210
`queens-18`	236,172	236,172	236,173
`queens-19`	16,178	16,178	16,178
`queens-20`	1,229,355	1,229,355	1,229,355
`check_list-8`	1	19	525,447
`check_list-9`	1	22	2,658,697
`check_list-10`	1	26	95,365,524
`check_list-11`	1	32	too big
`check_list-15`	1	64	-
`ppuzzle-A`	86	86	64,994,853
`ppuzzle-B`	215	215	-
`ppuzzle-C` 1st	29	29	751,567,244

Table 8.8: Number of splits for BEAM/AGENTS and non-det. steps for And-I.

BEAM on the `check_list` benchmark performs a single split because it delays splitting until success propagation. Hence it allows alternatives to be removed without even having to be exploited. In this benchmark, AGENTS suffers a small penalty for not being able to delay splitting due to the wait guards that are needed to control the clauses.

8.5 Final Remarks

Although the BEAM is still a prototype, results are promising. BEAM appears as an alternative to run programs where standard Prolog systems behave badly. Unlike AGENTS, the BEAM supports Prolog and unlike Andorra-I BEAM does not need pre-compilation analysis. Thus, the BEAM is an excellent alternative to applications where pre-compilation to Prolog may be expensive and difficult and where queries with large search space are generated in rapid succession. Finally, we have also shown that BEAM's performance can be improved by using simple annotations in the program.

Chapter 9

Parallelism- Towards the RAINBOW

One of the goals of the EAM is to exploit all the forms of implicit parallelism available in logic programs, including or-parallelism, independent-and and dependent-and parallelism. Therefore, only a parallel implementation of the BEAM can take full advantage of the EAM model. Towards this goal, we have designed a shared memory parallel computational model for BEAM, that we named RAINBOW (Rapid And-parallelism with INdependence Built upon Or-parallel Work), which we describe in detail here.

Next, we propose the RAINBOW computational model and all changes that the BEAM requires in order to support parallelism. Then we describe the RAINBOW memory organization and how the data-structures in the BEAM must be changed to accommodate parallelism. Next, we discuss scheduling strategies. Last, we discuss how parallelism affects the BEAM computational rules.

9.1 The Computational Model

In the RAINBOW a set of workers, or processes, executes in parallel trying to exploit a shared and-or execution tree. One of the workers, we shall name it the *chair*, initiates the computation, as in the BEAM, by creating the initial and-box. Parallelism is made available by marking this and-box as *shared*. The RAINBOW thus divides work as either shared or non-shared (this is similar to the notion of public work in or-parallel systems [84]).

All workers but the chair start execution in scheduling mode, that is, looking for their first work assignment. In our design, workers looking for work search for other workers' topmost shareable work. Note that in order to fetch work from a shared box, workers have to ensure mutual exclusion. Otherwise, races would be possible as two workers might pick the same task. This rule holds true even for the box owner. Therefore working on shared work is always slower than working in non-shared work.

We shall assume *demand-driven* scheduling in the RAINBOW: from time to time, each worker checks its marker to determine whether it has been fully exploited and needs to be updated to a different box in its subtree with shared work. The termination condition is given thus:

1. all the workers find themselves idle looking for work;
2. all the worker's markers indicate that there is no shared work. Computation then terminates.

9.2 Representing Local Variables

With multiple running workers there is always the possibility that several different workers may want to bind a value to a variable belonging to a shared and-box. This situation is usually known as the binding problem [148, 54]. In order to deal with this problem, local variables need special representation in a parallel system. We believe two alternative proposals to be the most interesting for RAINBOW: full sharing and copying plus sharing.

9.2.1 Full Sharing

In this solution the memory data areas are organized in a way similar to the sequential BEAM: the whole memory is shared by all the workers. The *Heap* and the *Box Memory* data areas are equally divided among the workers. Each worker manages its own memory areas. During normal execution workers behave as in the sequential BEAM, hence workers allocate memory in their own memory areas without having to synchronize with other workers. Locally allocated blocks are only seen by other workers whenever those blocks become part of the shared part of the execution tree.

We suggest two alternative techniques to address the binding problem for the shared RAINBOW: *version vectors* that store alternatives bindings [27], or by the Sparse Binding Array (SBA) [35].

If we use version vectors we have associated with each local variable a vector indexed by worker-id (see figure 9.1). The `value` field of the local variable structure is initialized to point to the base of the vector. When a worker assigns a binding to a local variable it must add its worker-id as an offset to the base address of the vector stored in the `value` field for that variable. When the binding of a local variable becomes deterministic, we optimize by storing the bindings directly in the `value` field associated with the variable, hence releasing the initial vector. This optimization may take place during a promotion phase. And-boxes that were suspended on the local variable, now found to be deterministic will receive a wake signal to force an environment check (as in the sequential BEAM).

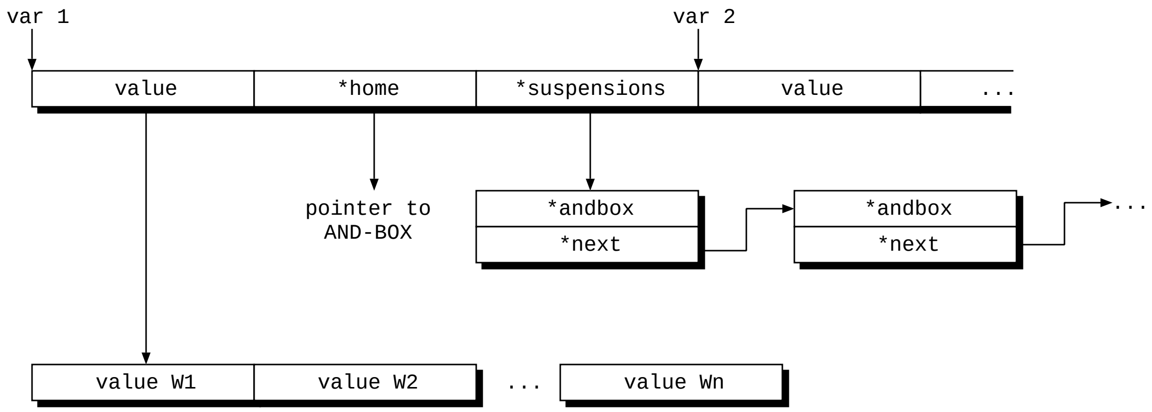

Figure 9.1: Local Variable representation using *version vectors*.

If we use the SBA each cell in the box memory will have a private copy (see figure 9.2). Bindings to shared local variables are stored in this area. The memory address for the variable is common for all workers, and is stored in the value field. Although the variable has the same memory address for all workers, when accessing a variable each worker uses it own private copy. Deterministic bindings are directly stored in the local variable structure.

One advantage of version vectors is that we may create the vectors lazily, that is, only

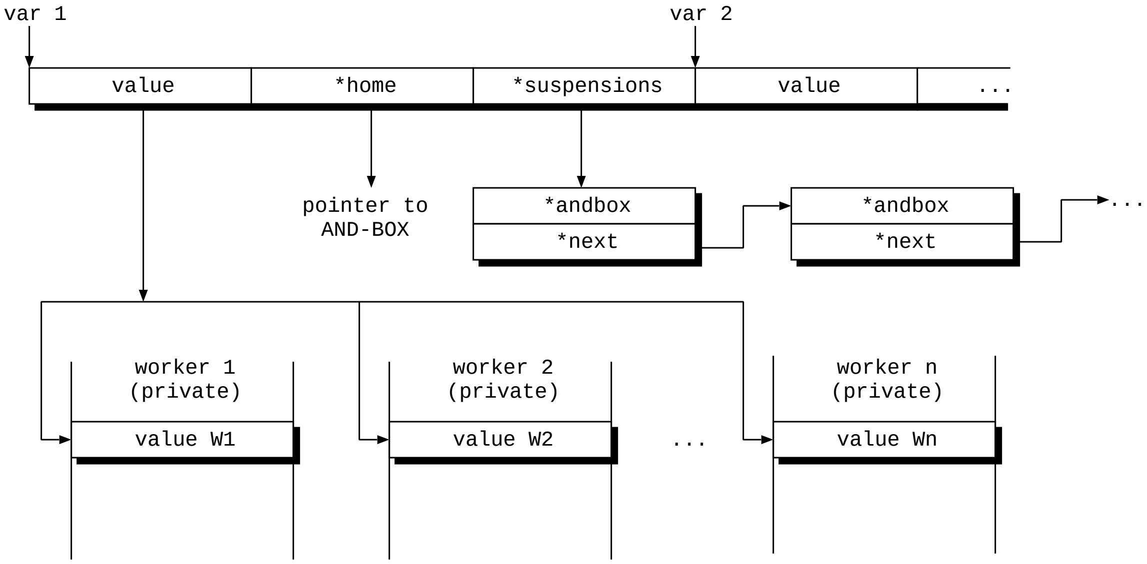

Figure 9.2: Local Variable representation using *Sparse Binding Array*.

when we first access a variable local to a shared and-box. To do so, we must first lock the variable, as otherwise we could have a race between several workers trying to create their own vectors. In RAINBOW this is not a problem as locking is necessary anyway, whereas in traditional or-parallel systems this would be a significant overhead. The SBA logically reserves sufficient space for the whole stacks, but it does have the advantage of avoiding the extra overhead of vector allocation.

An alternative to these two proposals is to use the PEPSys [12, 156] hashing scheme, as suggested for ParAKL [91]. The PEPSys model combines the *time-stamping* [141], used to distinguish the correct bindings for an environment and, *hashing windows* [15] for maintaining multiple environments. The binding for a variable is done directly if the variable is in the current alternative. Otherwise, the binding for the variable is saved in a hash window local to the current alternative. This scheme is optimized for task switching, but looking up for a non-local variable may require a long search through a long chain of hash windows.

In our case, it was the complexity of the BEAM design that make us feel more inclined towards the advantages of simpler data-structures such as version vectors and the SBA.

9.2.2 Copying Plus Sharing

As an alternate solution is to use both copying and sharing. The key ideas are:

1. workers generate work in fully private stacks;
2. shared work is copied to the *shared* work area;
3. the shared work area is available to every worker; the binding problem is address through the SBA.

We call this solution the COSMOS model (*C*opy-*O*nce, *S*hare-*M*any *O*bject *S*tore). Note that our design is such that shared work must never directly refer to private work. This solution has the following advantages:

1. Private work is clearly separated from shared work: this should facilitate memory management.
2. The SBA only applies to shared space, resulting in much smaller SBAs per worker.
3. Accesses to the shared area will most often be read-only. The system should thus nicely support distributed shared memory architectures, such as OAR [4]

It also has a few drawbacks, namely:

1. Extra complexity, namely in copying Heap objects.
2. The advantages of COSMOS will quickly disappear if we need to store a large number of boxes in the shared areas;

Note that our approach contrasts with the copy-all approach used in Penny [89]. The Penny approach was shown to work well for coarse-grained parallelism, but we believe one should experiment with other approaches in order to **(i)** simplify adapting to a distributed execution scheme, such as Oar [4]; and **(ii)** reduce overheads in computations with finer granularity.

9.2.3 Garbage Collection

Both memory organizations are designed to allow the implementation of a fully distributed garbage collector that does not need to stop all the workers to recover space. The key idea is that during execution, each worker must be able to trigger its copy based garbage collector (GC) to work on its private memory areas in a similar way as in the BEAM sequential implementation (see [81]). By avoiding pointers from the private to the shared area, COSMOS guarantees that garbage collection can be performed independently for the private trees. The full-sharing approach and the shareable part within COSMOS require a bit more care. Shared boxes will require reference counters, and we must establish some communication and synchronization among the workers when shifting shared boxes.

9.3 Distributing Work

We will now describe in more detail data structures that we propose towards supporting work distribution in RAINBOW. We assume a shared memory design.

The main data-structure is *Worker-Info*, a shared data structure, that guides workers towards available work in the system. We propose an array, indexed by worker-id, in which each entry gives the marker for a worker. Markers identify, directly in the execution tree, a box with the topmost work in a worker's subtree (this contrasts with queue-based approach used in Penny [89]). Each entry of the *worker-info* data structure includes:

- `box_pointer`: a pointer to the worker's box marked as having shared tasks.
- `box_type`: indicates the type of the marked box: and-box or or-box.
- `depth_level`: specifies the depth-level of the marked box within the execution tree. This field is a copy of the field we find in every and-box. However, we include it to simplify the work search algorithm.
- `work_available`: number of tasks still left on this box.

Workers, when looking for work, search the *worker_info* structure in order to find

W1	W2	...	Wn
box_pointer	box_pointer		box_pointer
box type	box type		box type
depth level	depth level		depth level
work-flag	work-flag		work-flag

Figure 9.3: Worker Information structure.

workers with their topmost marker still sharing work. Ideally, this worker is the one such as: $work\_available = Max(0, work\_available_0, \ldots work\_available_{n-1})$, if one such worker exists.

Markers also separate private work from shared areas. A worker's private area is defined as a subtree below the box pointed to by its marker. Within the private area the worker executes sequentially without having to synchronize.

There are two situations in which a worker must update its marker:

1. *No shared work*: whenever a marker points to a box with no work left (i.e. the *work_available* entry is zero), the owner worker can update it to the root of its private subtree.

2. *No local work left*: whenever a worker exhausts its private subtree, it returns to its marked box (in the shared area). If there is no work left at this box, it goes into scheduling mode looking for work from another worker. If the worker succeeds in finding a new task, then to start executing the task, the worker creates a new box and updates its marker to point to it.

9.3.1 An Example

To illustrate our execution model, consider the situation described in figure 9.4a. It shows three workers, W_1 and W_2 actively working below, respectively, and-box `a21` and or-box `o31`, while worker W_3 is looking for work. Currently, the worker-info array shows that the topmost box with shared work is `o11` of worker W_1. Suppose W_3 succeeds in grabbing a task from that box and starts executing it. Then, it naturally creates an and-box `a22`, and updates its entry in the worker-info array. Meanwhile, the box `o11` has no more work left and therefore worker W_1 must update its marker to the root of its private subtree. This is illustrated in figure 9.4b.

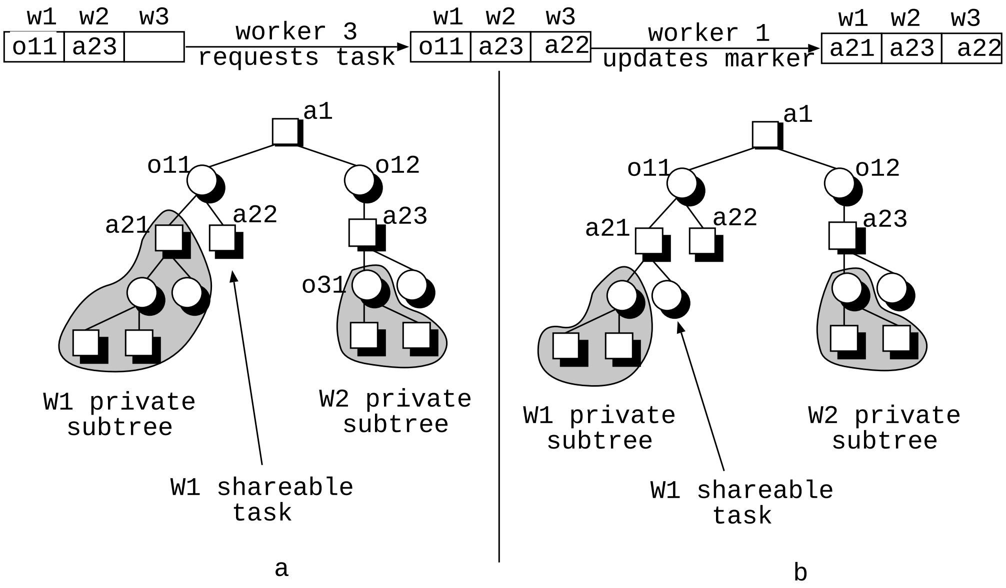

Figure 9.4: Execution Tree in a parallel environment.

9.4 Extending And-Or-Boxes

In RAINBOW several workers exploit the and-or execution tree concurrently. It is therefore necessary to be able to locate the workers in the tree, in order to share work and in order to avoid having multiple workers visiting the same branches. This is accomplished by extending the and-or boxes with two extra fields, the `lock` and the `workers` fields.

The `lock` field ensure mutual exclusion between workers when changing the box contents.

The `workers` bitmap identifies the workers that are working under the current box. A worker is known to be working on a box if its bit number is set. This field is used in situations such as splitting or box simplification where it is necessary to determine all the workers working within a box or determine if a worker is unique within it. When a worker leave a box, it unsets its bit in the `workers` field and checks whether the value of the field is zero. If this is the case, the worker checks the box `state` field to decide whether the box may be deleted.

9.5 Searching For Work

The scheduling strategy we propose follows from our initial description of the execution model. Basically, we distinguish two cases:

Local work search: during execution workers always try first to get work from their private area, starting from their `current_box`. If they are not successful in the current box they move upwards in the tree, in which case they may eventually reach their `marker_box`. Getting work from a box in the private part of the tree is realized as in sequential BEAM, while getting work from the `marker_box` has to be protected by the box's lock. Figure 9.5 shows the pseudo-code for the function `get_local_work()` that implements the strategy just outlined.

The `task` data structure includes all the necessary information for a worker to start computing a new task.

Remote work search: whenever a work fails to find local work in its private tree area or in its marker box, it directs its work search towards the worker in the system that has the topmost marker box. That box is found from the `worker-info` array where the markers for each worker with shared work and the depth level associated with the markers are kept. The worker that has the marker with the least depth level value is chosen as a target for work sharing. This algorithm is given in pseudo-code in function `get_remote_work()` in figure 9.5.

9.6 Implications On The Rewrite Rules

The BEAM rewrite rules are executed normally if the boxes involved in the rewrite operation are all in the worker's private area. Otherwise, some of the BEAM rewrite rules need to be modified and their application may have to be thought carefully. We will address each in turn:

- `reduction`: the reduction rule may be applied with no modifications, since it only changes the status on the worker's current box (i.e. it only affects the private tree area).

```
task get_local_work( )
{
  if (Is_Private(current_box)) {
    newtask=get_work_private(current_box);
    if (newtask!=NULL)
      return newtask;
    current_box=current_box->parent; /* move current_box ptr upwards */
    return get_local_work();         /* and call get_work() again    */
  }
  /* get work from public area since current_box=marker_box */
  if (worker_info[workerID].work_flag>0) {
    LOCK(current_box->lock) {
    newtask=get_work_private(current_box);
    UNLOCK(current_box->lock);
    if (newtask!=NULL)
      return newtask;
  }
  return get_remote_work();
}

task get_remote_work()
{
  targetWID= find_worker_with_topmost_marker();
  current_box= worker_info[targetWID].box_pointer;
  if (worker_info[targetWID].work_flag>0) {
    LOCK(current_box->lock) {
    newtask=get_work_private(current_box);
    if (newtask!=NULL) {
      set_bitmap(current_box->workers, workerID);
      UNLOCK(current_box->lock);
      return newtask;
    }
    UNLOCK(current_box->lock);
  }
  return get_remote_work();
}
```

Figure 9.5: Scheduler Algorithm.

- `splitting`: since the BEAM only applies this rule when no other rule can, splitting does not pose any difficulties in a parallel setting. A worker is only allowed to perform a split operation in a box if there are no other workers installed in a subtree of that box. If there were workers below the current box, then most likely there would be work to be explored from the boxes pointed by the markers of those workers. Notice that the or-box that is the root of the subtree where splitting is to be executed must be locked throughout the operation to guarantee that no other worker will try to work on the same subtree.

- `eager-splitting`: eager splitting is more problematic since there may exist other workers in the parent and-box where the splitting is to be performed. Figure 9.6 illustrates two workers, W_i and W_j, working on goals within the same and-box. In this case, W_i cannot perform eager splitting on `B` because W_j is working on the goal `A`. To deal with this problem, there are two different situations that one must consider:

  - if there is still work available outside the and-box where the eager splitting is to be executed, then the worker that was to perform the eager splitting (W_i in the example) should leave the and-box and try to get available work from another part of the And-Or Tree. Considering the example again, note that W_i should not pick the available work on `C` as it would prevent W_j from doing the eager splitting on `B` after completing its current work.
  - if there is no more work available outside the and-box where the eager-splitting is to be executed, then the worker that will perform the eager splitting (W_i in the example) must issue a synchronization request to the workers working bellow the parent and-box in order to be able to accomplish the eager splitting operation. Considering again our example, let us assume that goal `C` is the only available work in the shared part of the tree and that worker W_i wants to do a eager splitting on `B`. In this case, W_i sends a eager-splitting signal to W_j and waits for a confirmation from it. After receiving the confirmation, W_i performs the splitting operation and once it is completed sends a continuation signal to W_j.

- `promotion`: the application of this rule is a bit more complex in the parallel context. One has to ensure that the parent and-box does not have any other active workers in it for the rule to be applied. This requires mutual exclusion

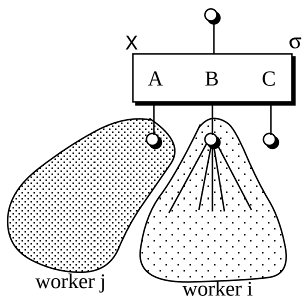

Figure 9.6: Eager splitting restrictions in a parallel environment.

on the parent and-box. However, if the parent and-box resides in the worker's private tree the rule can be applied as in the sequential BEAM.

- `propagation`: whenever a variable receives a deterministic binding, all boxes that were suspended with non-deterministic bindings to the variable must be signaled. When running in parallel there may exist several workers executing bellow the and-box where the variable was bounded. All these workers must receive a signal forcing environment checking, so that their constraints are updated.

- `and-compression` and `deterministic-reduce-and-promote`: these two rules are only allowed if all boxes involved in the operation are in the worker's private area, or if there is only one worker bellow the parent and-box where the operation is to be performed. In this last case the and-box is locked before applying the rewrite rule.

- `success propagation` and `failure propagation`: the execution of these rules often result in the removal of a box or a subtree. Thus, the parent box or subtree that is going to be deleted must be locked. Moreover, there may exist workers active in the subtree to be deleted. In this case, these workers must be signaled to abandon their current work and to start searching for work in another part of the tree.

- `cut`: the cut is used as in the sequential version if the boxes involved are in a worker's private region. Otherwise, the root of the subtree where the pruning operation is to be performed, must be locked. If there are workers working in the alternatives being pruned then, these workers must be signaled to stop working and to start searching for other work. Note that one can also optimize the computation by delaying the assignment of work that is under the cut scope, hence avoiding or reducing speculative work [60, 8].

9.7 Dealing With Suspended Work

Similar to what happens in the sequential BEAM, the and-box should suspend whenever it tries to bind an external variable. In the sequential version, the suspended work will be resumed when no other work is available. This situation is more complex in a parallel version.

We have two possible schemes for dealing with suspended work in a parallel environment. The first one favors the first goal presented by David H. D. Warren, that is, to minimize the number of inferences. A second scheme maximizes parallelism, David H. D. Warren's second goal.

- *Give preference to minimizing the number of inferences*: Workers transfer suspended work to a global shared queue. Workers try shareable deterministic work first. Suspended work resumes only when there is no deterministic work available,

- *Give preference to maximizing parallelism*: Workers transfer suspended work to a private queue. When a worker does not have deterministic work in its private area, the worker first tries to resume a suspended box in its private area. If no suspended work is available in its private are, then the worker tries to find shareable work.

The first scheme has the advantage of avoiding unnecessary splittings since splitting will only take place when no deterministic work is available in the whole tree. The second scheme has the disadvantage of possibly doing unnecessary splittings, but because a worker will always exhaust its private work before moving to shareable work it has the advantage of reducing the number of task switches, hence avoiding locking.

9.8 Final Remarks

We have presented our first design for RAINBOW, a parallel implementation of the BEAM. RAINBOW builds on the good results we have obtained with our implementation of the Extended Andorra Model with Implicit Control and on previous experience with parallel systems such as ParAKL and Penny.

We propose several alternative strategies for environment representation in RAINBOW. Version vectors are a possible solution that although not very popular for Prolog may be quite more appealing in our context. An alternative is COSMOS, a solution where only shared work is stored in shared memory. We also propose a work distribution and scheduling strategy for RAINBOW based on minimizing the amount of sharing in the system.

Chapter 10

Conclusions

We have presented the design and the implementation of the BEAM, a system for the efficient execution of logic programs based on David H. D. Warren's work on the Extended Andorra Model with implicit control. Our work was motivated by our interest in studying how the EAM with implicit control can be effectively implemented and how it can perform versus other execution strategies. We see our work as extending logic programming for applications where Prolog would not cope well, not as a replacement for Prolog. We believe that our results are quite encouraging in this direction.

Our approach contrasts with previous work in concurrent languages such as AKL. These are powerful concurrent languages that open up new programming paradigms, but that also require users to invest in sophisticated new programming frameworks. In contrast, our goal is to first support a very flexible mechanism for the execution of logic programs. The mechanism can then be expanded towards different purposes.

10.1 Contributions

The main contribution of this thesis is thus the design of the LIGHT-BEAM [77], a novel language framework based upon the original EAM. Our work in clarifying the EAM and in designing the LIGHT-BEAM has shown a crisp separation between the rewrite rules and the control rules. We have tried to make this separation clear in this work. This result suggests that there is a bit more to the EAM than what meets the eye:

- The rewrite-rules provide a lower-level description of the computation for a logic program;
- The control-rules provide a scheduling mechanism that takes advantage of the Andorra principle.

By maintaining control separate from the rewrite rules these observations suggest that one could consider different control strategies over the basic rewrite-rules. Indeed control may be made configurable, say, by using a specialized control language that can generate specialized And-Or Tree managers. We believe that a major contribution of the EAM is the exciting prospect of achieving specialized control strategies for different types of logic programs.

The contributions of this dissertation can be summarized as follows:

Complete description of an EAM Model We contribute with a full description of an EAM model: the BEAM. Our model implements the main functionalities of the EAM with implicit control. The BEAM was designed towards an efficient implementation. Unlike the original EAM, BEAM does not use choice-boxes. Instead, the rewrite rules were adapted to support a more flexible implementation of pruning without the overhead of an extra structure. We have also presented a new set of rules that allow the And-Or Tree to be simplified and the computation to be optimized.

Improving and controlling the search space Towards the goal of improving the search space we have addressed three major issues over the original design: when to allow splitting, how to perform pruning, and how to obtain better termination properties.

We have presented several alternatives that can be used to decide when to split, including a new algorithm for detecting *stability*.

The use of the cut together with promotion or the splitting rules of the EAM can generate incorrect computational states. A solution to address this problem was also presented.

We have presented an initial proposal for tabling as an extra control mechanism that can deal with the non-termination problem of the EAM. We have shown that tabling fits naturally with the EAM scheme, providing a very powerful computational model.

A First sequential EAM implementation The BEAM is the first actual (sequential) implementation of the EAM with implicit control. We present the architecture organization used in BEAM, namely the two main components: the *Abstract Machine Emulator* and the *And-Or Tree Manager*. For the first component we have detailed on the required registers and on the new abstract machine instructions. A scheme for compiling Prolog to the new abstract machine instructions was also presented. The And-Or Tree Manager has also been described and all its different modules were presented. An efficient form to implement the splitting was also detailed.

We have contributed with a novel design for a simple memory allocation mechanism used for the BEAM boxes. This mechanism has been proven to perform well, avoiding the need for frequent garbage collection on the And-Or Tree. Nevertheless, as we have shown, the garbage collector is still necessary in an Extended Andorra Model, specially because of the inability of recovering space on the Heap memory.

In-depth study of the BEAM memory usage and performance We have contributed with a detailed study of the BEAM memory usage. The queens benchmark has been used to show that our implementation achieves acceptable memory footprints even with larger benchmarks. We have also confirmed through simulation that garbage collection can have a large impact on execution time, and that this is partly caused by worse miss-rates on the data caches.

We have evaluated the BEAM performance and we have compared it with several different systems. Our results are very promising. The system achieves acceptable base performance. The results also revealed that BEAM with implicit control performs slightly better than AKL (using explicit control). Moreover, we show that implicit control can be in fact quite effective for a sizable number of applications, and that simple annotations can contribute to further improvements with little programmer effort. In this work we show a group of applications where the system benefits from the advanced search inherent to the EAM. A special benchmark was used to show a situation where Andorra-I is unable to reduce the search space and both BEAM and AKL can.

A parallel execution model for BEAM One of the main goals of the EAM was to explore full parallelism. Attending to this fact we proposed the RAINBOW [82] a parallel execution model for the BEAM. We have presented a general overview of how to distribute work, propose alternative approaches towards addressing the binding problem for the EAM, and present a scheduling strategy. We have addressed the control issues and the data-structures required for parallel execution in a shared-memory context, but with a view towards distributed shared memory execution.

10.2 Future Work

The BEAM prototype is currently being improved by providing better support for builtins. We also plan to integrate BEAM with a more recent version of YAP which should further improve BEAM performance. Ultimately, we aim at making the BEAM an extension of the YAP Prolog system that the user can exploit towards maximum performance in declarative applications.

We believe that the BEAM provides an excellent framework for novel logic programming applications. We are particularly interested in performance evaluation for automatically generated queries, say, the ones are found in Inductive Logic Programming [119]. In these applications, queries with large search spaces are generated in rapid succession. Reducing the search space is fundamental, but pre-compilation to Prolog may be expensive and difficult. We believe that the advanced search features of the EAM can be most useful for these applications.

The BEAM opens up a design space of novel mechanisms for improving the control of (constraint) logic programs. We are researching this space into providing an extensive framework for the effective execution of logic programs. As a result we have recently proposed the *LIGHT-BEAM* model, which addresses two major limitations of the BEAM by including support for tabling and for scoping of computing.

We are also pursuing work in the parallel execution of Extended Andorra Model programs. We plan to have an implementation of the proposed parallel model, the RAINBOW, in a near future.

Finally, we also intend to use Abreu's work on improving backward execution in the Andorra family of languages [3, 2], to optimize BEAM's performance. His work has attained good results in the AKL, and we believe that BEAM will also benefit from it.

10.3 Final Remark

Several systems have proved that parallel logic programming can be useful and practical. Several different strategies have been followed but none has been able to explore all forms of parallelism with good results. The Extended Andorra Model as other models show that parallel logic programming is taking steps to remove any restrictions that may exist on exploring all different forms of parallelism. The ultimate goal is to achieve maximum parallelism with minimal user intervention.

Appendix A

Benchmarks

The benchmark are available from the first author home page at:

http://www.ncc.up.pt/~rslopes/BEAM/Bench.html.

The **cal benchmark**

```
/*
Calendar program:
This algorithm was published in comp.programming and comes from
    Andy Lowry, lowry@watson.ibm.com, (914) 784-7925
    IBM Research, P.O. Box 704, Yorktown Heights, NY 10598
*/
% Last 10000 FoolsDays arithmetic benchmark:
cal:-         day_of_week(1993,4,9,Day),
        write('April 9, 1993 is: '),
        write(Day), nl.
% We associate a number with each day of the week:
dow(0, sun).
dow(1, mon).
dow(2, tue).
dow(3, wed).
dow(4, thu).
dow(5, fri).
dow(6, sat).
% The day of week computation is rather arcane, but it works.
% Note the correction for leap years.
day_of_week(Year,Month,Day, DayOfWeek):-
        cal_key(Month, Key, LeapC),
        compute_it(Year,Day,Key,LeapC,DayOfWeek).
compute_it(Year,Day,Key,LeapC,DayOfWeek):-
        Century is Year // 100,
        YearInCentury is Year - Century * 100,
        DOW0 is (Century * 5 + Century // 4 +
                 YearInCentury + YearInCentury // 4 +
                 Day + Key)
                 mod 7,
        leap_year(Year,DOW0,LeapC,DayOfWeek).
% A leap year is any year which is divisible by 4; if it is also
% divisible by 100 then it must also be divisible by 400 (thus,
% 1600 and 2000 are leap years; 1700, 1800, and 1900 are not).
leap_year(Year,DOW0,_,DayOfWeek):-
        0 =\= Year mod 4,
```

```
        !,
        dow(DOW0,DayOfWeek).
leap_year(Year,DOW0,LeapC,DayOfWeek):-
        0 =\= Year mod 100,
        !,
        DOW is DOW0-LeapC,
        dow(DOW,DayOfWeek).
leap_year(Year,DOW0,_,DayOfWeek):-
        0 =\= Year mod 400,
        !,
        dow(DOW0,DayOfWeek).
leap_year(_,DOW0,LeapC,DayOfWeek):-
        DOW is DOW0-LeapC,
        dow(DOW,DayOfWeek).
/*
There is a special key value for each month and a
correction factor for January and February in leap years.
*/

cal_key( 1, 6, 1).
cal_key( 2, 2, 1).
cal_key( 3, 2, 0).
cal_key( 4, 5, 0).
cal_key( 5, 0, 0).
cal_key( 6, 3, 0).
cal_key( 7, 5, 0).
cal_key( 8, 1, 0).
cal_key( 9, 4, 0).
cal_key(10, 6, 0).
cal_key(11, 2, 0).
cal_key(12, 4, 0).
cal_key(jan, 6, 1).
cal_key(feb, 2, 1).
cal_key(mar, 2, 0).
cal_key(apr, 5, 0).
cal_key(may, 0, 0).
cal_key(jun, 3, 0).
cal_key(jul, 5, 0).
cal_key(aug, 1, 0).
cal_key(sep, 4, 0).
cal_key(oct, 6, 0).
cal_key(nov, 2, 0).
cal_key(dec, 4, 0).
cal_key('January', 6, 1).
cal_key('February', 2, 1).
cal_key('March', 2, 0).
cal_key('April', 5, 0).
cal_key('May', 0, 0).
cal_key('June', 3, 0).
cal_key('July', 5, 0).
cal_key('August', 1, 0).
cal_key('September',4, 0).
cal_key('October', 6, 0).
cal_key('November',2, 0).
cal_key('December', 4, 0).
```

The deriv benchmark

```
deriv:- ops8(R1),divide10(R2),log10(R3),times10(R4).
ops8(R) :- d((x+1)*((^(x,2)+2)*(^(x,3)+3)),x,R).
divide10(R) :- d((((((((((x/x)/x)/x)/x)/x)/x)/x)/x)/x,x,R).
log10(R) :- d(log(log(log(log(log(log(log(log(log(log(x)))))))))),x,R).
times10(R) :- d((((((((((x*x)*x)*x)*x)*x)*x)*x)*x)*x,x,R).
d(U+V,X,DU+DV) :- !, d(U,X,DU), d(V,X,DV).
d(U-V,X,DU-DV) :- !, d(U,X,DU), d(V,X,DV).
d(U*V,X,DU*V+U*DV) :- !, d(U,X,DU), d(V,X,DV).
d(U/V,X,(DU*V-U*DV)/(^(V,2))) :- !, d(U,X,DU), d(V,X,DV).
d(^(U,N),X,DU*N*(^(U,N1))) :- !, integer(N), N1 is N-1, d(U,X,DU).
d(-U,X,-DU) :- !, d(U,X,DU).
d(exp(U),X,exp(U)*DU) :- !, d(U,X,DU).
d(log(U),X,DU/U) :- !, d(U,X,DU).
d(X,X,1) :- !.
d(_,_,0).
```

The qsort benchmark

```
qsort:- qsort([27,74,17, ... , 40,53,59, 8],X,[]), write(X).
qsort([X|L],R,R0) :- partition(L,X,L1,L2), qsort(L2,R1,R0), qsort(L1,R,[X|R1]).
qsort([],R,R).
partition([X|L],Y,[X|L1],L2) :- X =< Y, partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :-       X > Y, partition(L,Y,L1,L2).
partition([],_,[],[]).
```

The serialise benchmark

```
% generated: 17 November 1989
% option(s):
%
%   serialise
%
%   David H. D. Warren
%
%   itemize (pick a "serial number" for each
%   unique integer in) a list of 25 integers
ser:- serialise("ABLE WAS I ERE I SAW ELBA",R), write(R), fail.
ser.
serialise(L,R) :- pairlists(L,R,A), arrange(A,T), numbered(T,1,_).
pairlists([X|L],[Y|R],[pair(X,Y)|A]) :- pairlists(L,R,A).
pairlists([],[],[]).
arrange([X|L],tree(T1,X,T2)) :-
split(L,X,L1,L2),
arrange(L1,T1),
arrange(L2,T2).
arrange([],void).
split([X|L],X,L1,L2) :- !, split(L,X,L1,L2).
split([X|L],Y,[X|L1],L2) :- before(X,Y), !, split(L,Y,L1,L2).
split([X|L],Y,L1,[X|L2]) :- before(Y,X), !, split(L,Y,L1,L2).
split([],_,[],[]).
before(pair(X1,_),pair(X2,_)) :- X1 < X2.
numbered(tree(T1,pair(_,N1),T2),N0,N) :-
numbered(T1,N0,N1),
N2 is N1+1,
numbered(T2,N2,N).
numbered(void,N,N).
```

The rev benchmark

```
rev(U) :- rev([0,1,2, ... , 999],[],U).
rev([],Y,Y).
rev([H|T],Y,Z):- rev(T,[H|Y],Z).
```

The nreverse benchmark

```
nreverse(U) :- nreverse([1,2, ... , 1000],U), write(U).
nreverse([],[]).
nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
concatenate([],L,L).
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
```

The kkqueens benchmark

```
/*-------------------------------------------------------------------------------
Program:  N-Queens (candidates/noncandidates)
Author:   K. Kumon
Modified: E. Tick
Date:     July 9 1989
KL1 vers: T. Chikayama
Date:          May 13 1993
Notes:
Input number of queens.
Number of solutions and statistics will be printed out.
-------------------------------------------------------------------------------*/
main:- kkqueens(9,M),  write(M), nl.
kkqueens(N,M) :-
    gen(N,L),
    queen(L,[],[],X,[]),
    count(X, M).
queen([], [], L, I, O):- I=[L|O].
queen([], [_|_], _, I, O):- I=O.
queen([P|U], C, L, I0, I2):-
check(L, P, 1, U,C, L, I0, I1),
queen(U, [P|C], L, I1, I2).
check([], T, _D, U,C, B, I, O):-
concat(U, C, N),
queen(N, [], [T|B], I, O).
check([P|R], T, D, U,C, B, I, O):-
        T =\= P+D,
        T =\= P-D, !,
        D1 is D+1,
        check(R, T, D1, U,C, B, I, O).
check([P|_R],T, D,_U,_C,_B,I, O):- T =:= P+D, !, I=O.
check([P|_R],T, D,_U,_C,_B,I, O):- T =:= P-D, !, I=O.
concat([], Y, Z) :- Z = Y.
concat([W|X], Y, WZ) :- WZ = [W|Z], concat(X, Y, Z).
gen(N, X) :- N > 0, !, X = [N|Xs], M is N-1, gen(M,Xs).
gen(N, X) :- N =:= 0, !, X = [].
count(L,N) :- count(L,0,N).
count([],M,N) :- N = M.
count([_|Xs],M,N) :- M1 is M+1, count(Xs,M1,N).
```

The tak benchmark

```
% generated: 17 November 1989
% option(s): SOURCE_TRANSFORM_1
%
%   tak
%
%   Evan Tick (from Lisp version by R. P. Gabriel)
%
%   (almost) Takeuchi function (recursive arithmetic)
tak:- tak(18,12,6,U), write(U).
tak(X,Y,Z,A):- X =< Y, Z=A.
tak(X,Y,Z,A):-
        X > Y,
        X1 is X - 1,
        tak(X1,Y,Z,A1),
        Y1 is Y - 1,
        tak(Y1,Z,X,A2),
        Z1 is Z - 1,
        tak(Z1,X,Y,A3),
        tak(A1,A2,A3,A).
```

The ancestor benchmark

```
ancestor:- ancestor(a,X), ancestor(b,X), write([X]), nl, fail.
ancestor.
ancestor(X,Y):-   parent(X,Y).
ancestor(X,Z):-   parent(X,Y),  ancestor(Y,Z).
parent(a,fa).     parent(a,ma).
parent(ma,mma).   parent(ma,fma).
parent(fa,c).     parent(fa,d).
parent(b,fb).     parent(b,mb).
parent(mb,c).     parent(mb,d).
parent(fb,mfb).   parent(fb,ffb).
```

The houses benchmark

```
houses:- go_hs(U), write(U), fail.
houses.
memb( X2, [ X1 | X0 ] ) :- memb( X2, X0 ).
memb( X1, [ X1 | X0 ] ).
testp( X3, X2, [ X3 | X1 ], [ X2 | X0 ] ).
testp( X5, X4, [ X3 | X2 ], [ X1 | X0 ] ) :- testp( X5, X4, X2, X0 ).
on_right( X4, X3, [ X4 | X2 ], [ X1, X3 | X0 ] ).
on_right( X5, X4, [ X3 | X2 ], [ X1 | X0 ] ) :- on_right( X5, X4, X2, X0 ).
on_left( X4, X3, [ X2, X4 | X1 ], [ X3 | X0 ] ).
on_left( X5, X4, [ X3 | X2 ], [ X1 | X0 ] ) :- on_left( X5, X4, X2, X0 ).
beside( X3, X2, X1, X0 ) :- on_right( X3, X2, X1, X0 ).
beside( X3, X2, X1, X0 ) :- on_left( X3, X2, X1, X0 ).
go_hs( config( X27, X26, X25, X24, X23 ) ) :-
        X27=[ norway, X22, X21, X20, X19 ],
        X26=[ X18, X17, X16, X15, X14 ],
        X25=[ X13, X12, X11, X10, X9 ],
        X24=[ X8, X7, milk, X6, X5 ],
        X23=[ X4, X3, X2, X1, X0 ],
        testp( england, red, X27, X26 ),
        testp( japan, craven, X27, X23 ),
        beside( norway, blue, X27, X26 ),
        on_right( white, green, X26, X26 ),
        testp( yellow, kool, X26, X23 ),
        testp( spain, dog, X27, X25 ),
        testp( old_gold, snail, X23, X25 ),
        testp( gitane, wine, X23, X24 ),
        testp( ukr, tea, X27, X24 ),
        testp( green, coffe, X26, X24 ),
        beside( chesterfield, fox, X23, X25 ),
        beside( kool, horse, X23, X25 ),
        memb( zebra, X25 ),
        memb( water, X24 ).
```

The query benchmark

```
main:- query(U), write(U), fail.                        main.
query([C1,D1,C2,D2]) :- density(C1,D1), density(C2,D2),
      D1 > D2,  T1 is 20*D1,  T2 is 21*D2,  T1 < T2.
density(C,D) :-  pop(C,P), area(C,A),  D is (P*100)//A.
```

```
% populations in 100000's
pop(china,8250).
pop(india,5863).
pop(ussr,2521).
pop(usa,2119).
pop(indonesia,1276).
pop(japan,1097).
pop(brazil,1042).
pop(bangladesh, 750).
pop(pakistan, 682).
pop(w_germany, 620).
pop(nigeria, 613).
pop(mexico, 581).
pop(uk,         559).
pop(italy, 554).
pop(france, 525).
pop(philippines, 415).
pop(thailand, 410).
pop(turkey, 383).
pop(egypt, 364).
pop(spain, 352).
pop(poland, 337).
pop(s_korea, 335).
pop(iran, 320).
pop(ethiopia, 272).
pop(argentina, 251).
```

```
% areas in 1000's of square miles
area(china,     3380).
area(india,     1139).
area(ussr,      8708).
area(usa,       3609).
area(indonesia,  570).
area(japan,      148).
area(brazil,    3288).
area(bangladesh,  55).
area(pakistan,   311).
area(w_germany,   96).
area(nigeria,    373).
area(mexico,     764).
area(uk,  86).
area(italy,      116).
area(france,     213).
area(philippines, 90).
area(thailand,   200).
area(turkey,     296).
area(egypt,      386).
area(spain,      190).
area(poland,     121).
area(s_korea,     37).
area(iran,       628).
area(ethiopia,   350).
area(argentina, 1080).
```

The zebra benchmark

```
main:-  houses(Houses),
        member(house(red, english, _, _, _), Houses),
        member(house(_, spanish, dog, _, _), Houses),
        member(house(green, _, _, coffee, _), Houses),
        member(house(_, ukrainian, _, tea, _), Houses),
        right_of(house(green,_,_,_,_), house(ivory,_,_,_,_), Houses),
        member(house(_, _, snails, _, winstons), Houses),
        member(house(yellow, _, _, _, kools), Houses),
        Houses = [_, _, house(_, _, _, milk, _), _,_],
        Houses = [house(_, norwegian, _, _, _)|_],
        next_to(house(_,_,_,_,chesterfields), house(_,_,fox,_,_), Houses),
        next_to(house(_,_,_,_,kools), house(_,_,horse,_,_), Houses),
        member(house(_, _, _, orange_juice, lucky_strikes), Houses),
        member(house(_, japanese, _, _, parliaments), Houses),
        next_to(house(_,norwegian,_,_,_), house(blue,_,_,_,_), Houses),
        member(house(_, _, zebra, _, _), Houses),
        member(house(_, _, _, water, _), Houses),
        print_houses(Houses), fail.
main.
houses([house(_, _, _, _, _), house(_, _, _, _, _), house(_, _, _, _, _),
        house(_, _, _, _, _), house(_, _, _, _, _)]).
right_of(A, B, [B, A | _]).   right_of(A, B, [_ | Y]) :- right_of(A, B, Y).
next_to(A, B, [A, B | _]).    next_to(A, B, [B, A | _]).
next_to(A, B, [_ | Y]) :- next_to(A, B, Y).
member(X, [X|_]).             member(X, [_|Y]) :- member(X, Y).
print_houses([]).             print_houses([A|B]) :- write(A), nl, print_houses(B).
```

The puzzle4x4 benchmark

```
main:- pz4x4_go8(X), write(X).
pz4x4_go8(S):-
        pz4x4_problem8(Y),
        pz4x4_solve(8,Y,S).
pz4x4_go9(S):-
        pz4x4_problem9(Y),
        pz4x4_solve(9,Y,S).
pz4x4_go10(S):-
        pz4x4_problem10(Y),
        pz4x4_solve(10,Y,S).
pz4x4_go11(S):-
        pz4x4_problem11(Y),
        pz4x4_solve(11,Y,S).
pz4x4_go12(S):-
        pz4x4_problem12(Y),
        pz4x4_solve(12,Y,S).
pz4x4_problem8([ 1, 2, 3, 8,
           5, 6,11, 7,
           9, o,10,16,
          13,14,12,15]).
pz4x4_problem9([ 1, 2, 3, 8,
           5, 6,11, 7,
           o, 9,10,16,
          13,14,12,15]).
pz4x4_problem10([ 1, 2, 3, 8,
            5, 6,11, 7,
           13, 9,10,16,
            o,14,12,15]).
pz4x4_problem11([ o, 2, 3, 8,
            1, 6,11, 7,
            5, 9,10,16,
           13,14,12,15]).
pz4x4_problem12([ 2, o, 3, 8,
            1, 6,11, 7,
            5, 9,10,16,
           13,14,12,15]).
pz4x4_move(1,0).
pz4x4_move(2,1).
pz4x4_move(3,2).
pz4x4_move(4,3).
pz4x4_move(5,4).
pz4x4_move(6,5).
pz4x4_move(7,6).
pz4x4_move(8,7).
pz4x4_move(9,8).
pz4x4_move(10,9).
pz4x4_move(11,10).
pz4x4_move(12,11).
pz4x4_solve(_,[ 1, 2, 3, o,
          5, 6, 7, 8,
          9,10,11,12,
         13,14,15,16],[]):-  true.
pz4x4_solve(N, [P11,  o,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m11|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[  o,P11,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                  o,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m11|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[  o,P12,P13,P14,
                P11,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[  o,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m12|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P12,  o,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,  o,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m12|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,  o,P12,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m12|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,  o,P13,P14,
                P21,P12,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,  o,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m13|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P13,  o,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,  o,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m13|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,  o,P13,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m13|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,  o,P14,
                P21,P22,P13,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,  o,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
```

```
                P41,P42,P43,P44],[m14|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P14,  o,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,  o,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m14|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,  o,
                P21,P22,P23,P14,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[  o,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m21|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P21,P12,P13,P14,
                  o,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m21|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                  o,P21,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                  o,P32,P33,P34,
                P41,P42,P43,P44],[m21|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                  o,P22,P23,P24,
                P21,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,  o,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m22|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P22,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                  o,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m22|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P22,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,  o,P33,P34,
                P41,P42,P43,P44],[m22|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P22,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m22|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,  o,P22,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,  o,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m23|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P23,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m23|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P23,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,  o,P34,
                P41,P42,P43,P44],[m23|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,  o,P24,
                P31,P32,P23,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,  o,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m23|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,  o,P23,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,  o,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m24|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P24,
                P21,P22,P23,  o,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m24|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P24,  o,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
```

```
                P21,P22,P23,P24,
                P31,P32,P33, o,
                P41,P42,P43,P44],[m24|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23, o,
                P31,P32,P33,P24,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                  o,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m31|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P31,P22,P23,P24,
                  o,P32,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,  o,P33,P34,
                P41,P42,P43,P44],[m31|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                  o,P31,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                  o,P42,P43,P44],[m31|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                  o,P32,P33,P34,
                P31,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m32|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P32,P23,P24,
                P31,  o,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                  o,P32,P33,P34,
                P41,P42,P43,P44],[m32|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P32,  o,P33,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,  o,P34,
                P41,P42,P43,P44],[m32|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,  o,P32,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,  o,P43,P44],[m32|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,  o,P33,P34,
                P41,P32,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m33|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P33,P24,
                P31,P32,  o,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,  o,P33,P34,
                P41,P42,P43,P44],[m33|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P33,  o,P34,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,  o,
                P41,P42,P43,P44],[m33|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,  o,P33,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,  o,P44],[m33|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32, o,P34,
                P41,P42,P33,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,  o,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m34|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P34,
                P31,P32,P33,  o,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,  o,P34,
                P41,P42,P43,P44],[m34|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P34,  o,
                P41,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,  o],[m34|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,  o,
                P41,P42,P43,P34],L).
```

```
pz4x4_solve(N,[P11,P12,P13,P14,
               P21,P22,P23,P24,
                 o,P32,P33,P34,
               P41,P42,P43,P44],[m41|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P41,P32,P33,P34,
                 o,P42,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P33,P34,
               P41,  o,P43,P44],[m41|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P33,P34,
                 o,P41,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,  o,P33,P34,
               P41,P42,P43,P44],[m42|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P42,P33,P34,
               P41,  o,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P33,P34,
                 o,P42,P43,P44],[m42|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P33,P34,
               P42,  o,P43,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P33,P34,
               P41,P42,  o,P44],[m42|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P33,P34,
               P41,  o,P42,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,  o,P34,
               P41,P42,P43,P44],[m43|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P43,P34,
               P41,P42,  o,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P33,P34,
               P41,  o,P43,P44],[m43|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P33,P34,
               P41,P43,  o,P44],L).
pz4x4_solve(N,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P33,P34,
               P41,P42,P43,  o],[m43|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P33,P34,
               P41,P42,  o,P43],L).
pz4x4_solve(N,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P33,  o,
               P41,P42,P43,P44],[m44|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P33,P44,
               P41,P42,P43,  o],L).
pz4x4_solve(N,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P33,P34,
               P41,P42,  o,P44],[m44|L]):-
      pz4x4_move(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,P33,P34,
               P41,P42,P44,  o],L).
```

The Send-More-Money benchmark

```
send_money:- solve(A,B,C,D,E,F,G,H), print_out(A,B,C,D,E,F,G,H).
difflist([A|B]) :-  not_el(A,B), difflist(B).
difflist([]).
digi(9).  digi(8).  digi(7).  digi(6).  digi(5).
digi(4).  digi(3).  digi(2).  digi(1).  digi(0).
not_el(A,[]).
not_el(A,[B|C]) :-  A\==B, not_el(A,C).
carry(1,1,1).  carry(1,1,0).  carry(1,0,1).  carry(1,0,0).
carry(0,1,1).  carry(0,1,0).  carry(0,0,1).  carry(0,0,0).
generate(A,B,C,D,E,F,G,H) :-
        digi(E),  digi(A),  digi(F), digi(B),
        digi(C),  digi(G),  digi(D), digi(H),
        difflist([A,B,C,D,E,F,G,H]).
print_out(A,B,C,D,E,F,G,H) :-
        write('    SEND'),  nl,
        write(' +  MORE'),  nl,
        write(-----------), nl,
        write('   MONEY'),  nl, nl,
        write('The solution is:'), nl, nl,
        write('   '), write(A), write(B), write(C), write(D), nl,
        write(' + '), write(E), write(F), write(G), write(B), nl,
        write(-----------), nl,
        write('  '),  write(E), write(F), write(C), write(B), write(H), nl.
solve(A,B,C,D,E,F,G,H) :-
        carry(I,J,K),
        E=1,
        generate(A,B,C,D,E,F,G,H),
        A>0,
        L is D+B,
        M is H+10*I,
        L=M,
        N is I+C+G,
        O is B+10*J,
        N=O,
        P is J+B+F,
        Q is C+10*K,
        P=Q,
        R is K+A+E,
        S is F+10*E,
        R=S.
```

The Queens benchmark

```
queens(L,C):- perm(L,P), pair(L,P,C), safe([],C).
perm([],[]).
perm(Xs,[Z|Zs]):- select(Z,Xs,Ys), perm(Ys,Zs).
select(X,[Y|Ys],[Y|Zs]):- select(X,Ys,Zs).
select(X,[X|Xs],Xs).
pair([],[],[]).
pair([X|Y],[U|V],[p(X,U)|W]):- pair(Y,V,W).
safe(X,[]).
safe(X,[Q|R]):- test(X,Q), safe([Q|X],R).
test([],X).
test([R|S],Q):- test(S,Q), nd(R,Q).
nd(p(C1,R1),p(C2,R2)):-
        C is C1-C2,
        R is R1-R2,
        C =\= R,
        NR is R2-R1,
        C=\=NR.
```

The check_list benchmark

```
list(Size):- get_list(Size,P), list_is_correct(P), write(P).
%Get_List(Size, List)
get_list(8,[(1,1),(1,2),(2,2),(3,2),(4,3),(4,4),(4,3),(3,3)]).
get_list(8,[(1,1),(1,2),(2,2),(3,2),(4,3),(4,4),(3,4),(3,3)]).
get_list(9,[(1,1),(1,2),(2,2),(3,2),(4,2),(4,3),(4,4),(4,3),(3,3)]).
get_list(9,[(1,1),(1,2),(2,2),(3,2),(4,2),(4,3),(4,4),(3,4),(3,3)]).
get_list(10,[(1,1),(1,2),(2,2),(3,2),(3,1),(4,1),(4,2),(4,3),(3,3),(4,3)]).
get_list(10,[(1,1),(1,2),(2,2),(3,2),(3,1),(4,1),(4,2),(4,3),(4,4),(3,4)]).
get_list(11,[(1,1),(1,2),(2,2),(3,2),(3,1),(4,1),(4,2),(4,3),(4,4),(4,3),(3,3)]).
get_list(11,[(1,1),(1,2),(2,2),(3,2),(3,1),(4,1),(4,2),(4,3),(4,4),(3,4),(3,3)]).
get_list(15,[(1,1),(1,2),(2,2),(3,2),(3,1),(4,1),(4,2),(4,3),(4,4),(3,4),(2,4),
             (2,3),(1,3),(2,3),(3,3)]).
get_list(15,[(1,1),(1,2),(2,2),(3,2),(3,1),(4,1),(4,2),(4,3),(4,4),(3,4),(2,4),
             (1,4),(1,3),(2,3),(3,3)]).
list_is_correct([]).
list_is_correct([(X,Y)|R]):- is_not_in((X,Y),R), list_is_correct(R).
is_not_in((_,_),[]).
is_not_in((X,Y),[(X2,_)|R]):- X =\= X2, is_not_in((X,Y),R).
is_not_in((X,Y),[(_,Y2)|R]):- Y =\= Y2, is_not_in((X,Y),R).
```

The ppuzzle benchmark

```
main:- ppuzzle(4,Sol).
ppuzzle(N,P):- create_list(N,N,L), mix(L,L,R), permute(R,P),
               puzzleA(N,P), % or puzzleB or puzzleC
               write(P), nl, fail.
ppuzzle(_,_).
create_list(0,_,[]).
create_list(N,X,[Y|R]):- N>0, M is N-1, Y is X-M, create_list(M,X,R).
mix([],_,[]).
mix([X|R],L,R3):- mixer(X,L,R1), mix(R,L,R2), append(R1,R2,R3).
mixer(X,[],[]).
mixer(X,[Y|R],[p(X,Y)|L]):- mixer(X,R,L).
append([],X,X).
append([A|B],X,[A|B1]):- append(B,X,B1).
permute([],[]).
permute(L,[N|L2]):- take_off(L,N,R), permute(R,L2).
take_off([X|R],X,R).
take_off([Y|R],X,[Y|L]):-  take_off(R,X,L).
puzzleA(W,[p(X,Y),p(X,X),p(Y,X),p(Y,Y),p(Y,3),p(X,3),p(X,4),
  p(Y,4),p(3,4),p(3,3),p(3,Y),p(3,X),p(4,X),p(4,Y),p(4,3),p(4,4)]).
puzzleA(W,[p(4,4),p(4,3),p(4,Y),p(4,X),p(3,X),p(3,Y),p(3,3),
  p(3,4),p(Y,4),p(X,4),p(X,3),p(Y,3),p(Y,Y),p(X,Y),p(X,X),p(Y,X)]).
puzzleB(W,[p(X,Y),p(X,X),p(Y,X),p(Y,Y),p(Y,Z),p(X,Z),p(X,W),
  p(Y,W),p(Z,W),p(Z,Z),p(Z,Y),p(Z,X),p(W,X),p(W,Y),p(W,Z),p(W,W)]).
puzzleB(W,[p(W,W),p(W,Z),p(W,Y),p(W,X),p(Z,X),p(Z,Y),p(Z,Z),
  p(Z,W),p(Y,W),p(X,W),p(X,Z),p(Y,Z),p(Y,Y),p(X,Y),p(X,X),p(Y,X)]).
% Puzzle C - verifies all valid paths between two different
% points in a puzzle size N*N
puzzleC(_,L):- verify_all_paths(L).
verify_all_paths([]).
verify_all_paths([p(X,Y)|R]):- verify(R,p(X,Y)).
verify([],_).
verify([p(Z,Y)|R],p(X,Y)):- Z is X+1, verify(R,p(Z,Y)).
verify([p(Z,Y)|R],p(X,Y)):- Z is X-1, verify(R,p(Z,Y)).
verify([p(X,W)|R],p(X,Y)):- W is Y+1, verify(R,p(X,W)).
verify([p(X,W)|R],p(X,Y)):- W is Y-1, verify(R,p(X,W)).
```

References

[1] *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Japan, 1992. Association for Computing Machinery.

[2] S. Abreu, L. M. Pereira, and P. Codognet. Improving backward execution in the andorra family of languages. In Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 384–398, Washington, USA, 1992. The MIT Press.

[3] Salvador Abreu. *Improving the Parallel Execution of Logic Programs.* PhD thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 1994.

[4] Salvador Abreu. Towards the OAR Language and Computational Model. In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Languages*, July 2000.

[5] Salvador Abreu and Luis Moniz Pereira. Design for AKL with intelligent pruning. In *Extensions of Logic Programming*, pages 3–10, 1993.

[6] Abderrahamane Aggoun, David Chan, Pierre Dufresne, Eamon Falvey, Hugh Grant, Alexander Herold, Geoffrey Macartney, Micha Meier, David Miller, Shyam Mudambi, Bruno Perez, Emmanuel van Rossum, Joachim Schimpf, Periklis Andreas Tsahageas, and Dominique Henry de Villeneuve. *ECLiPSe 3.5 User Manual.* ECRC, December 1995.

[7] Hassan Aït-Kaci. *Warren's Abstract Machine — A Tutorial Reconstruction.* MIT Press, 1991.

[8] K. Ali and R. Karlsson. Scheduling Speculative Work in MUSE and Performance Results. *International Journal of Parallel Programming*, 21(6):449–476, December 1992.

[9] Khayri A. M. Ali and Roland 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.

[10] Reem Bahgat. *Pandora: Non-Deterministic Parallel Logic Programming.* PhD thesis, Department of Computing, Imperial College of Science & Technology, February 1991.

[11] Reem Bahgat and Steve Gregory. Pandora: Non-deterministic Parallel Logic Programming. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 471–486. MIT Press, June 1989.

[12] Uri Baron, Jacques Chassin de Kergommeux, Max Hailperin, Michael Ratcliffe, Philippe Robert, Jean-Claude Syre, and Harald Westphal. The Parallel ECRC Prolog System PEPSys: an Overview and Evaluation Results. In *International Conference on Fifth Generation Computer Systems 1988*, pages 841–850. ICOT, Tokyo, Japan, November 1988.

[13] Anthony Beaumont, S Muthu Raman, Péter Szeredi, and David 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.

[14] Patrice Boizumault. A General Model to Implement Dif and Freeze. In Ehud Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 585–592. Springer-Verlag, 1986.

[15] P. Borgwardt. Parallel prolog using stack segments on shared memory multiprocessors. In *International Symposium on Logic Programming*, pages 2–12, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.

[16] Francisco Bueno and Manuel V. Hermenegildo. An Automatic Translations Scheme from Prolog to the Andorra Kernel Language. In *International Conference on Fifth Generation Computer Systems 1992*, pages 759–769. ICOT, Tokyo, Japan, June 1992.

[17] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, University of Wisconsin-Madison Computer Sciences Department, June 1997.

[18] B. Carlson, Mats Carlsson, and D. Diaz. Entailment of finite domain constraints. In Pascal Van Hentenryck, editor, *ICLP'94, International Conference on Logic Programming*, MIT Press Series in Logic Programming, S. Margherita Ligure, Italy, 1994. The MIT Press.

[19] Björn Carlson, Seif Haridi, and Sverker Janson. AKL(FD)-A concurrent language for FD programming. In *Logic Programming: Proceedings of the 1994 International Symposium*, pages 521–535. MIT Press, 1994.

[20] Mats Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In Jean-Louis 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.

[21] Mats Carlsson. Internals of Sicstus Prolog version 0.6. Internal Report, Gigalips Project, November 1987.

[22] Mats 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.

[23] Mats Carlsson and Johan Widen. SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science, 1988. SICS Research Report R88007B.

[24] Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.

[25] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[26] Takashi Chikayama, Tetsuro Fujise, and Daigo Sekit. A Portable and Efficient Implementation of KL1. In *6th International Symposium PLILP*, pages 25–39, 1994.

[27] Andrzej Ciepielewski and Seif Haridi. A formal model for or-parallel execution of logic programs. In *IFIP 83 Conference*, pages 299–305. North Holland, 1983.

[28] Keith L. Clark and Steve 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.

[29] Keith L. Clark and Steve Gregory. PARLOG: Parallel Programming in Logic. *ACM TOPLAS*, 8:1–49, January 1986.

[30] Keith L. Clark, F. G. McCabe, and Steve Gregory. IC-PROLOG – language features. In K. L. Clark and S. Å. Tärnlund, editors, *Logic Programming*, pages 253–266. Academic Press, London, 1982.

[31] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[32] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *The Journal of Logic Programming*, 27(3):185–226, June 1996.

[33] 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.

[34] Alain Colmerauer. Theoretical Model of Prolog II. In Michel van Caneghen and David H. D. Warren, editors, *Logic Programming and its Applications*, pages 3–31. Ablex Publishing Corporation, 1986.

[35] Manuel Eduardo Correia, Fernando Silva, and Vítor Santos Costa. The SBA: Exploiting orthogonality in OR-AND Parallel Systems. In *Proceedings of the 1997 International Logic Programming Symposium*, pages 117–131. MIT Press, October 1997. Also published as Technical Report DCC-97-3, DCC - FC & LIACC, Universidade do Porto, April, 1997.

[36] James 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.

[37] James A. Crammond. The abstract machine and implementation of parallel parlog. *New Generation Computing*, 10(4):385–422, 1992.

[38] L. Damas, V. Santos Costa, R. Reis, and R. Azevedo. *YAP User's Guide and Reference Manual*, 1998.

[39] Arthur Delcher and Simon Kasif. Some Results on the Complexity of Exploiting Data Dependency in Parallel Logic Programs. *The Journal of Logic Programming*, 6(3):229–242, May 1989.

[40] P. Deransart, A. Ed-Dbali, L. Cervoni, and A. A. Ed-Ball. *Prolog, The Standard : Reference Manual.* Springer Verlag, 1996.

[41] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. 24(6), 1994.

[42] Inês 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.

[43] Inês C. Dutra. Design of a Flexible Scheduler for Andorra-I. Internal Report, PEPMA Project, January 1991.

[44] Inês C. Dutra. Strategies for Scheduling And- and Or-Work in Parallel Logic Programming Systems. In *Logic Programming: Proceedings of the 1994 International Symposium*, pages 289–304. MIT Press, 1994.

[45] Torkel Franzén, Seif Haridi, and Sverker Janson. An overview of the andorra kernel language. In P. Eriksson, L.-H.; Hallnäs, L.; Schroeder-Heister, editor, *Proceedings of the Second International Workshop on Extensions of Logic Programming (ELP'91)*, volume 596 of *LNAI*, pages 163–180, Stockholm, Sweden, January 1991. Springer Verlag.

[46] J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *Proceedings of the Eight International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in LNCS, pages 243–258, Aachen, Germany, September 1996. Springer-Verlag.

[47] J. Gabriel, T. Lindholm, E. L. Lusk, and Ross A. Overbeek. A Tutorial on the Warren Abstract Machine for Computational Logic. Research Paper ANL-84-84, Argonne National Laboratory, June 1985.

[48] C. Ghezzi and M. Jazayeri. *Programming Languages Concepts.* John Wiley & Sons, 1982.

[49] Gopal Gupta. *Parallel Execution of Logic Programs on a Shared Memory Multiprocessors.* PhD thesis, University of North Carolina at Chapel Hill, 1991.

[50] Gopal Gupta, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel Logic Programming: A Survey. Research Report, ftp://pub.cs.nmsu.edu/pub/papers/lp/survey.ps.Z, 1997.

[51] Gopal Gupta and Manuel 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.

[52] Gopal Gupta and Manuel V. Hermenegildo. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *LNCS 569, ICLP'91 Workshop on Parallelism*, pages 146–158, 1991.

[53] Gopal Gupta, Manuel V. Hermenegildo, Enrico Pontelli, and Vítor Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP'94*, pages 93–109. MIT Press, 1994.

[54] Gopal Gupta and Bharat Jayaraman. Analysis of or-parallel execution models. *ACM TOPLAS*, 15(4):659–680, 1993.

[55] Gopal Gupta and Enrico Pontelli. Extended dynamic dependent And-parallelism in ACE. In ACM, editor, *PASCO '97. Proceedings of the second international symposium on parallel symbolic computation, July 20–22, 1997, Maui, HI*, pages 68–79, New York, NY 10036, USA, 1997. ACM Press.

[56] Gopal Gupta and David H. D. Warren. An Interpreter for the Extended Andorra Model. Internal report, University of Bristol, 1991.

[57] D. Harel and A. Pnueli. *Logics and Models of Concurrent Systems*, chapter The development of reactive systems, pages 477–498. Springer-Verlag, 1985.

[58] Seif Haridi and Sverker Jansson. Kernel Andorra Prolog and its Computational Model. In David H. D. Warren and Péter Szeredi, editors, *Proceedings of the*

Seventh International Conference on Logic Programming, pages 31–46. MIT Press, 1990.

[59] Julie Sussman Harold Abelson, Gerald Jay Sussman, editor. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[60] B. Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG.* PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, March 1990.

[61] Bogumił Hausman. Hybrid implementation techniques in Erlang BEAM. In Leon Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 816–816, Cambridge, June 13–18 1995. MIT Press.

[62] Manuel V. Hermenegildo and Kevin 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.

[63] Manuel V. Hermenegildo and Kevin Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[64] Manuel V. Hermenegildo and Roger 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.

[65] R. Hill. LUSH-Resolution and its Completeness. Dcl memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.

[66] P. Hudak and S. Jones et al. Report on the Programming Language Haskell: a non-strict, purely functional language. *ACM Sigplan Notices*, 27(5), 1992.

[67] Sverker Janson. *AKL – A Multiparadigm Programming Language.* SICS Dissertation Series 14, Uppsala University, 1994.

[68] Sverker Janson. *AKL - A Multiparadigm Programming Language.* PhD thesis, SICS Swedish Institute of Computer Science, Uppsala University, 1994.

[69] Sverker Janson and Seif 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.

[70] Sverker Janson and Johan Montelius. Design of a Sequential Prototype Implementation of the Andorra Kernel Language. Sics research report, Swedish Institute of Computer Science, 1992.

[71] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996. Reprinted February 1997.

[72] Y. Kimura and Takashi 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.

[73] Feliks Kluźniak and Stanislaw Szpakowicz. *Prolog for Programmers*. Academic Press, 1985.

[74] Peter M. Kogge. *The Architecture of Symbolic Computers*. McGraw-Hill, 1991.

[75] Robert A. Kowalski. *Logic for Problem Solving*. Elsevier North-Holland Inc., 1979.

[76] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

[77] Ricardo Lopes, Vítor Santos Costa, and Fernando Silva. A novel implementation of the extended andorra model. *Lecture Notes in Computer Science*, pages 199–213, 2001.

[78] Ricardo Lopes and Vítor Santos Costa. *Parallelism and Implementation of Logic and Constraint Logic Programming*, chapter The BEAM: Towards a first EAM Implementation, pages 87–106. Nova Science, Inc., 1999.

[79] Ricardo Lopes and Vítor Santos Costa. The BEAM: a first EAM Implementation. In *Proceedings of AGP99: The 1999 Joint Conference on Declarative Programming*, pages 425–439, September 1999.

[80] Ricardo Lopes and Vítor Santos Costa. A Performance Analysis of the BEAM Memory Manager. In *Proceedings of AGP00: The 2000 Joint Conference on Declarative Programming*, December 2000.

[81] Ricardo Lopes and Vítor Santos Costa. Memory Management for the BEAM. In *CL2000 First Workshop on Memory Management in Logic Programs*, Technical Report of Dept. Comp. Science, K.U.Leuven, July 2000.

[82] Ricardo Lopes, Fernando Silva, Vítor Santos Costa, and Salvador Abreu. The RAINBOW: Towards a Parallel Beam. In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Languages*, July 2000.

[83] Ewing Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross A. Overbeek, Rick Stevens, David H. D. Warren, Alan Calderwood, Péter Szeredi, Seif Haridi, Per Brand, Mats Carlsson, Andrzej Ciepielewski, and Bogumił Hausman. The Aurora or-parallel Prolog system. In *International Conference on Fifth Generation Computer Systems 1988*, pages 819–830. ICOT, Tokyo, Japan, November 1988.

[84] Ewing Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross A. Overbeek, Rick Stevens, David H. D. Warren, Alan Calderwood, Péter Szeredi, Seif Haridi, Per Brand, Mats Carlsson, Andrzej Ciepielewski, and Bogumił Hausman. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[85] Peter S. Magnusson. A design for efficient simulation of a multiprocessor. In *Proceedings of MASCOTS*, pages 69–78, 1993.

[86] Peter S. Magnusson. Efficient instruction cache simulation and execution profiling with a threaded-code interpreter. In *Proceedings of the 97' Winter Simulation Conference*, 1997.

[87] Johan Montelius. *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*. PhD thesis, Uppsala University, 1997.

[88] Johan Montelius and Khayri A. M. Ali. An And/Or-Parallel Implementation of AKL. Proc. NSF/ICOT Workshop on Parallel Logic Programming and its Environments, CIS-94-04, University of Oregon, Mar. 1994.

[89] Johan Montelius and Khayri A. M. Ali. An And/Or-Parallel Implementation of AKL. *New Generation Computing*, 13(4), 1995.

[90] Johan Montelius and Peter Magnusson. Using SIMICS to evaluate the Penny system. In Jan Małuszyński, editor, *Proceedings of the International Symposium on Logic Programming (ILPS-97)*, pages 133–148, Cambridge, October 13–16 1997. MIT Press.

[91] Remco Moolenaar and Bart Demoen. A parallel implementation for AKL. In *Proceedings of the Programming Language Implementation and Logic Programming: PLILP '93, Tallin, Estonia*, pages 246–261, 1993.

[92] Remco Moolenaar and Bart Demoen. Hybrid tree search in the Andorra Model. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 110–123. MIT Press, June 1994.

[93] Lee Naish. *Negation and Control in Prolog.* Lecture notes in Computer Science 238. Springer–Verlag, 1985.

[94] Doug Palmer. The DAM: A Parallel Implementation of the AKL. Presented at the ILPS workshop on Parallel Logic Programming, October 1991.

[95] Doug Palmer and L. Naish. NUA-Prolog: an Extension to the WAM for Parallel Andorra. In Koishi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming.* MIT Press, 1991.

[96] Vijay A. Saraswat Pascal Van Hentenryck and Yves Deville. Design, implementation and evaluation of the constraint language cc(fd). Technical report no. cs-93-02, Dept. of Computer Science, Brown University, Jan 1993.

[97] Luís Moniz Pereira and António Porto. Intelligent Backtracking and Sidetracking in Horn Clause Programs - the Theory. Report 2/79, Departamento de Informatica, Universidade Nova de Lisboa, October 1979.

[98] Enrico Pontelli and Gopal Gupta. Dependent and Extended Dynamic Dependent And-parallelism in ACE. *Journal of Functional and Logic Programming*, 1999(Special Issue 1), April 1999.

[99] Enrico Pontelli, Gopal Gupta, and Manuel V. Hermenegildo. Implementation and performance of &ace: An independent and-para llel system. Technical report, NMSU, 1994.

[100] Enrico Pontelli, Gopal Gupta, and Manuel V. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.

[101] Enrico Pontelli, Gopal Gupta, Manuel V. Hermenegildo, Manuel Carro, and DongXing Tang. Efficient Implementation of And-Parallel Logic Programming Systems. *Computer Languages*, 22(2/3), 1996.

[102] A. Porto. Epilog: A language for extended programming in logic. In J. Campbell, editor, *Implementations of Prolog*, pages 268–278. Ellis Horwood, 1984.

[103] T. W. Pratt. *Programming Languages: design and implementation.* Prentice-Hall, 1975.

[104] *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.

[105] J. Reynolds. Transformational systems and algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–152, 1970.

[106] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(23):23–41, January 1965.

[107] R. Rocha, F. Silva, and V. Santos Costa. Or-Parallelism within Tabling. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, number 1551 in LNCS, pages 137–151, San Antonio, Texas, USA, January 1999. Springer-Verlag.

[108] R. Rocha, F. Silva, and V. Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Proceedings of the 2nd Conference on Tabulation in Parsing and Deduction*, pages 77–87, Vigo, Spain, September 2000.

[109] R. Rocha, F. Silva, and V. Santos Costa. On a Tabling Engine that Can Exploit Or-Parallelism. In *Proceedings of the 18th International Conference on Logic Programming*, Cyprus, November/December 2001. To appear.

[110] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. YapOr: an Or-Parallel Prolog System based on Environment Copying. In *LNAI 1695, Proceedings of EPPIA'99: The 9th Portuguese Conference on Artificial Intelligence*, pages 178–192. Springer-Verlag LNAI Series, September 1999.

[111] Kazuaki Rokusawa, Akihiko Nakase, and Takashi Chikayama. Distributed memory implementation of klic. *New Generation Computing*, 14(3):261–280, 1996.

[112] K. Sagonas. *The SLG-WAM: A Search-Efficient Engine for Well-Founded Evaluation of Normal Logic Programs.* PhD thesis, Department of Computer Science, State University of New York, Stony Brook, USA, August 1996.

[113] K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, May 1998.

[114] Kostis Sagonas, Terrance Swift, David S. Warren, Juliana Freire, and P. Rao. The XSB programmer's manual. Technical report, State University of New York at Stony Brook, 1997. Available at http://www.cs.sunysb.edu/˜sbprolog.

[115] Vítor Santos Costa, David H. D. Warren, and Rong Yang. Andorra-I Compilation. *New Generation Computing*, 14(1):3–30, 1996.

[116] Vítor Santos Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I.* PhD thesis, University of Bristol, August 1993.

[117] Vítor Santos Costa. *Encyclopedia of Computer Science and Technology*, chapter Parallelism and Implementation Technology for Logic Programming Languages. Marcel Dekker Inc, 2000 (to appear).

[118] Vítor Santos Costa, L. Damas, R. Reis, and R. Azevedo. *YAP User's Manual*, 2000. http://www.ncc.up.pt/˜vsc/Yap.

[119] Vítor Santos Costa, Ashwin Srinivasan, and Rui Camacho. A note on two simple transformations for improving the efficiency of an ILP system. In *Proceedings of ILP'2000*, July 2000. Proceedings to appear in Springer-Verlag LNAI series.

[120] Vítor Santos Costa, David H. D. Warren, and Rong 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.

[121] Vítor Santos Costa, David H. D. Warren, and Rong 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.

[122] Vítor Santos Costa, David H. D. Warren, and Rong 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.

[123] Vijay A. Saraswat. Partial Correctness Semantics for **CP**[$\downarrow$,|,&,;]. In *Proceedings of the Foundations of Software Technology and Theoretical Computer Science Conference*, pages 347–368, December 1985.

[124] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989. Available as Technical Report CMU-CS-89-108.

[125] 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.

[126] M. Sato, H. Shimizu, A. Matsumoto, K. Rokusawa, and A. Goto. KL1 Execution Model for PIM Cluster with Shared Memory. In Jean-Louis 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.

[127] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In A.H. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, vol. 874, pages 134–150, Orcas Island, Washington, USA, May 1994. Springer-Verlag.

[128] Sverker Janson Seif Haridi. Concurrent Constraint Programming at SICS with the Andorra Kernel Language (Extended Abstract). In *Proceedings First International Workshop on Principles and Practices of Constraint Programming*, 1993.

[129] Ehud Shapiro. A Subset of Concurrent Prolog and Its Interpreter. In Ehud Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 27–83. MIT Press, Cambridge MA, 1987.

[130] Ehud Shapiro. *Concurrent Prolog: Collected Papers.* MIT Press, 1987.

[131] Ehud Shapiro. The family of Concurrent Logic Programming Languages. *ACM computing surveys*, 21(3):412–510, 1989.

[132] Kish Shen. Initial Results from the Parallel Implementation of DASWAM. In Michael Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming.* The MIT Press, 1996.

[133] Kish Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3), 1996.

[134] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[135] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, 1986.

[136] T. Swift. *Efficient Evaluation of Normal Logic Programs.* PhD thesis, Department of Computer Science, State University of New York, Stony Brook, USA, December 1994.

[137] Akikazu Takeuchi. *Parallel Logic Programming.* PhD thesis, University of Tokyo, July 1990.

[138] Jiro Tanaka, Kazunori Ueda, Toshihiko Miyazaki, Akikazu Takeuchi, Yuji Matsumoto, and Koichi 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.

[139] 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.

[140] Evan Tick. *Parallel Logic Programming*. MIT Press, 1991.

[141] P. Tinker. Performance of an OR-parallel Logic Programming system. *International Journal of Parallel Programming*, 17(1):59–92, 1988.

[142] Kazunori Ueda and Masao 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.

[143] Henk Van Acker, Remco Moolenaar, and Bart Demoen. A parallel implementation of AKL. Presented at the ILPS workshop on Parallel Logic Programming, October 1991.

[144] M. H. van Emdem and G. J. de Lucena Filho. Predicate Logic as a Language for Parallel Programming. In K. L. Clark and S. Å. Tärnlund, editors, *Logic Programming*, pages 189–198. Academic Press, London, 1982.

[145] Peter Van Roy. Logic programming in Oz with Mozart. In Danny De Schreye, editor, *International Conference on Logic Programming*, pages 38–51, Las Cruces, NM, USA, November 1999. The MIT Press.

[146] David H. D. Warren. Implementing Prolog - Compiling Predicate Logic Programs. Technical Report 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[147] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[148] David H. D. Warren. Or-Parallel Execution Models of Prolog. In *TAPSOFT'87, The 1987 International Joint Conference on Theory and Practice of Software Development, Pisa, Italy*, pages 243–259. Springer-Verlag, March 1987.

[149] David H. D. Warren. The Andorra model. Presented at Gigalips Project workshop, University of Manchester, March 1988.

[150] David H. D. Warren. Extended Andorra model. PEPMA Project workshop, University of Bristol, October 1989.

[151] David H. D. Warren. The Extended Andorra Model with Implicit Control. Presented at ICLP'90 Workshop on Parallel Logic Programming, Eilat, Israel, June 1990.

[152] David 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.

[153] David H. D. Warren, Luís Moniz Pereira, and F. C. N. Pereira. Prolog—The Language and its Implementation Compared with Lisp. *ACM SIGPLAN Notices*, 12(8):109–115, 1977.

[154] David S. Warren. Efficient Prolog Memory Management for Flexible Control Strategies. In *International Symposium on Logic Programming*, pages 198–203, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.

[155] D.H.D. Warren. An improved prolog implementation which optimises tail recursion. Research Paper 156, Dept. of Artificial Intelligence, University of Edinburgh, 1980.

[156] Harald Westphal, Philippe Robert, Jacques Chassin de Kergommeaux, and Jean-Claude Syre. The PEPSys model: combining backtracking, and- and or-parallelism. In *The 1987 Symposium on Logic Programming, San Francisco, California*. IEEE, 1987.

[157] Rong Yang and Hideo Aiso. P-Prolog: a Parallel Logic Language Based on Exclusive Relation. In Ehud Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 255–269. Springer-Verlag, July 1986.

[158] Rong Yang, Anthony Beaumont, Inês C. Dutra, Vítor Santos Costa, and David 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.