The BEAM: Towards a first EAM Implementation

Ricardo Lopes, Vítor Santos Costa
{rslopes,vsc}@ncc.up.pt

LIACC, Universidade do Porto,
Rua do Campo Alegre, 823, 4150 Porto, Portugal

September 10, 1997

Abstract

Logic programming provides a high-level view of programming, giving implementors a vast latitude in what techniques to research towards obtaining the best performance for logic programs. One of the holy grails of logic programming has been to design computational models that could be executed efficiently and would allow for both 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. We report on the design of a first implementation for the Extended Andorra Model with Implicit Control, the BEAM.

1 Introduction

Logic programming 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 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. The high-level nature of logic programs gives implementors a vast latitude in what techniques to research towards obtaining the best performance for logic programs. Advanced compilation techniques [10] have been used to obtain performance close to imperative languages for some benchmarks [9]. Parallelism [6] allows logic programs to transparently exploit the advantages of the new parallel machines that are more than ever gaining in popularity. Extensions such as co-routining, constraints, and tabulation allow for substantial reductions in search space over the traditional Prolog selection function, and are a basis on which to build advanced features and support novel applications. 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 for both a reduction of the search space and for exploiting all the available parallelism in the application. These goals have been presented by David H. D. Warren [12] as, in order of priority:

- *Minimum number of inferences*: this is achieved by trying never to repeat the same execution step of one inference in different locations of the execution tree.

- *Maximum parallelism*: this is achieved by allowing goals to be executed as more independently as possible, and by doing the combination of all solutions as late as feasible.

These goals have motivated the design of the Extended Andorra Model. The model originates from the research on the Basic Andorra Model (BAM) [11]. The Andorra-I prototype [7] demonstrated that the BAM can successfully exploit two forms of parallelism, or-parallelism and dependent and-parallelism, and that it can reduce the search space for a large category of programs. On the other hand, Andorra-I depends on finding determinate applications. If determinacy cannot be found efficiently, applications with strong parallelism may need to be run sequentially. The Extended Andorra Model was proposed as an extension of the Andorra Model that could address the major problems of the Basic Andorra Model, and approximate the goals of minimising the search space and maximising the parallelism. The original proposal was based in the concept of rewriting and- and or-boxes. Research then evolved in two directions. David H. D. Warren and researchers at Bristol concentrated on the"1991" *Extended Andorra Model with Implicit Control* [12], where the goal was to obtain the advantages of the Extended Andorra Model with least effort from the programmer. Haridi, Janson and researchers at SICS concentrated on the AKL [3], 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. Research in AKL concentrated on including novel features such as ports [2], important for concurrent programming, and on obtaining good performance through sequential [4] and parallel implementations, such as Penny [5], with good results. More recently, the AKL researchers have shifted their interest to Oz [8], a language developed at DFKI which provides similar features to AKL. Research on the Extended Andorra Model with Implicit Control, that will refer to more shortly as *EAM*, followed a rather different approach. Research started by trying to prove the feasibility of the model by developing an interpreter. Gupta's interpreter [1] showed that the EAM was indeed feasible, but demonstrated difficulties in giving the best performance for some programs and queries. Towards obtaining maximum performance, Warren proposed solutions such as lazy copying and eager producers to give finer control over search when required. We report on the design of a first implementation for the EAM. As a first step we concentrate on the original rewriting rules of the EAM. This subset of the EAM will be implemented on our first prototype, the *BEAM*. We present the major data structures and execution algorithms that are required for efficient execution. The paper is organised as follows. We first give an overview of the EAM, and present an example for better understanding of the model. Next, we discuss the basic data structures required to implement the model. Last, we discuss some issues on extending the EAM to avoid infinite computations and present one solution.

2 The Extended Andorra Model

The EAM was designed to generate all answers to problems coded in Horn with a minimum number of inferences. The EAM tries to achieve this goal by performing as many steps in parallel as possible. 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.

2.1 Execution Principles for the EAM

The EAM is formally defined using rewrite rules which manipulate and-or-trees. These are represented through *and*, *or* and *choice-boxes*.

- **And-box:** $[\exists X_1, ..., X_m : \sigma \& C_1 \& ... \& C_n]$

  And-boxes are created when launching a new clause, the variables X_1 to X_n represent the variables created in the box and σ represents the constraints on external variables imposed by the and-box.

- **Or-box:** $\{C_1 \vee ... \vee C_n\}$

  Or-boxes are created when launching a new goal, each C_i represents an alternative clause for the goal.

- **Choice-box:** $\{G_1 \% C_1 \vee ... \vee G_n \% C_n\}$

  Choice-boxes are special or-boxes 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, |. We shall not discuss choice-boxes further here.

There are four main operations in the Extended Andorra Model:

- **Reduction:**

$$G \rightarrow \{[\exists Y_1 : \sigma_1 \% C_1] \vee ... \vee [\exists Y_n : \sigma_1 \% C_n]\}$$

  Reduction expands a goal G in an and-box into an or-box.

- **Promotion:**

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

  Promotion promotes constraints from a inner-box to the outer and-box.

- **Substitution:**

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

  Substitution applies a constraint on a variable.

- **Forking:**

$$[\exists X : \{(C_1 \vee ... \vee C_n) \ \& \ G] \rightarrow \{[\exists X : C_1 \& G] \vee ... \vee [\exists X : C_n \ \& \ G]\}$$

  Forking distributes a conjunction across a disjunction.

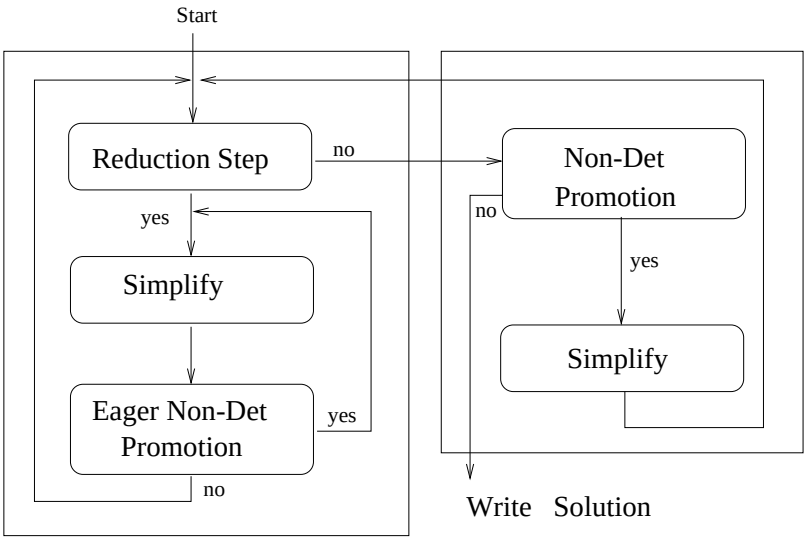

Figure 1: Execution.

2.2 Control in the EAM

The EAM uses a control scheme where an and-box suspends if it constrains an external variable. Figure 1, taken from Gupta's report [1], shows the control scheme used in the EAM interpreter. The systems tries to apply reduction and promotion steps as much as possible. *Simplification* steps are also used in the interpreter to maintain the data structures.

Forking is the operation that can lead to duplication of work, and it thus is only applied when all the other operations cannot be applied further. In the EAM interpreter, if several goals can produce a non-determinate binding for a variable, the leftmost one will be chosen. Notice that, before forking is started, a binding for variable X has to be generated by one of the C_i. After forking, this binding can be promoted in the newly created and-box. A forking step followed by a promotion is called *global forking*. The EAM allows the parallel early execution of non-determinate goals. In some cases, this may create speculative and-work, as some other goals might fail. This may result in a larger search space or even lead to non-termination, as the EAM allows any computation to go ahead as long as it does not bind variables. 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. Section 5 discusses this problem in more detail. One of the aims of the Extended Andorra Model is to perform the least number of reductions to obtain the solutions for a goal. To obtain this effect one often wants to do computations for different goals as independently as possible. Notice that in order to obtain maximum independence between goals it is necessary to limit the number of goals that can be allowed to delay global forking. That is, global forking should happen as soon as the goals that can bind the variable, and only this goals, cannot apply any other operations. A solution to classify these goals as used in the interpreter is to give scope to variables, possibly obtained through rewriting clauses in mini-scope form [1].

2.3 An Example of EAM Execution

This section presents an simple example that shows the essential ideas of how the computation of an Prolog program proceeds in EAM. Consider the very simple Prolog program:

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

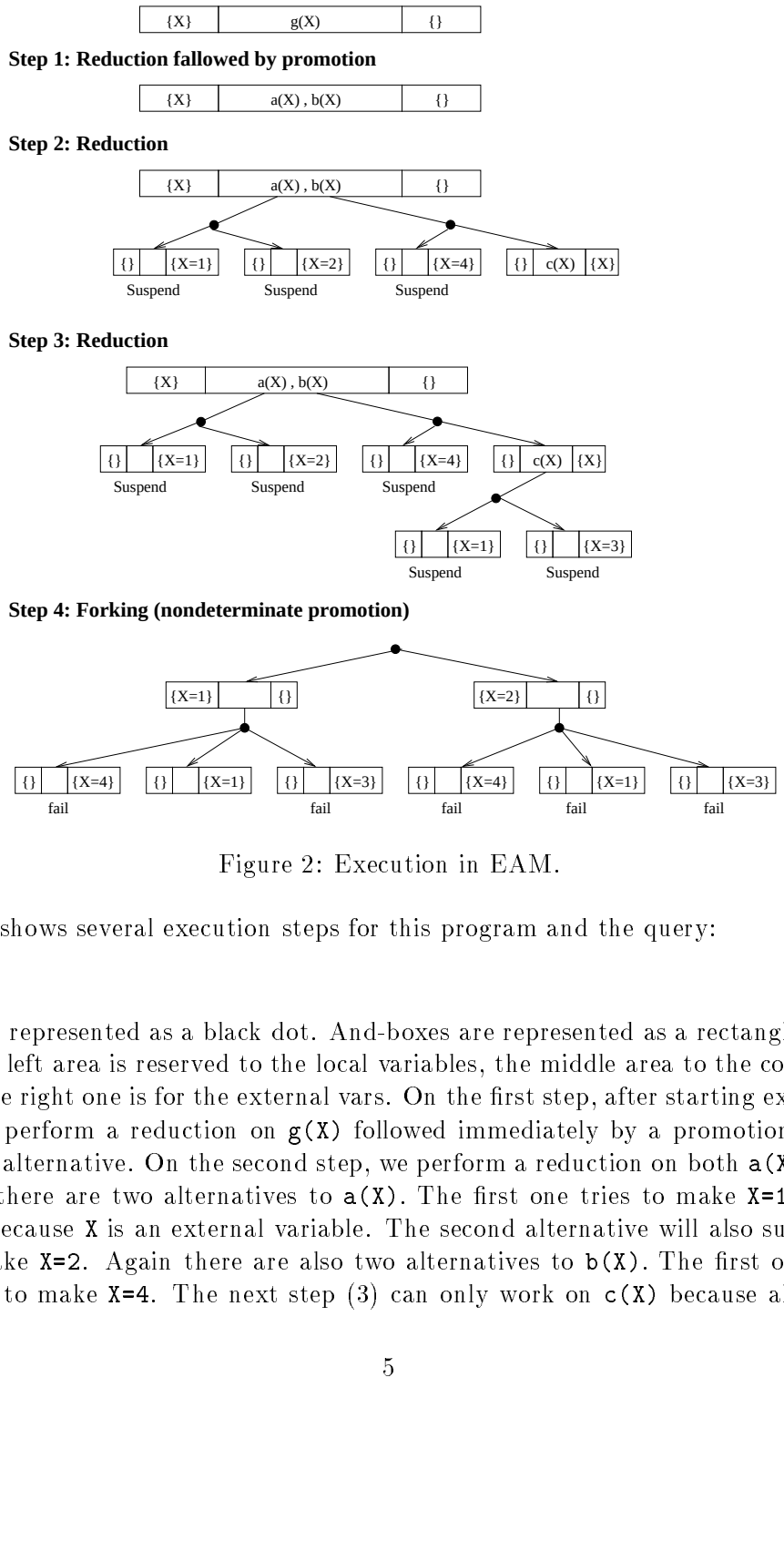

Figure 2: Execution in EAM.

Figure 2 shows several execution steps for this program and the query:

```
?- g(X).
```

Or-boxes are represented as a black dot. And-boxes are represented as a rectangle divided in 3 areas. The left area is reserved to the local variables, the middle area to the conjunction of goals, and the right one is for the external vars. On the first step, after starting execution, the EAM would perform a reduction on `g(X)` followed immediately by a promotion since `g(X)` only has one alternative. On the second step, we perform a reduction on both `a(X)` and `b(X)`. Notice now there are two alternatives to `a(X)`. The first one tries to make `X=1`, leading to suspension because `X` is an external variable. The second alternative will also suspend when trying to make `X=2`. Again there are also two alternatives to `b(X)`. The first one suspends when trying to make `X=4`. The next step (3) can only work on `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 `X`. At this point, all possible alternatives are suspended, which leaves only execution strategy, applying a nondeterminate promotion. We choose the or-box that is the leftmost on the tree. After forking, we can see the tree divided in two, with a or-box on top. 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. As you can conclude from this very simple example, the representation of and-boxes plus or-boxes are very important in a EAM implementation, because all the EAM operations are to be applied on and-boxes and or-boxes. Another important point is the representation of variables, because before any assignment is necessary to know if a variable is local or external.

2.4 Lazy Copying

In the original EAM a goal will always suspend when nondeterminately trying to bind an external variable, irrespective of whether it is the producer (the goal that generates bindings for) or the consumer (the goal that reads bindings for) that variable. One important observation is that if the goal is the producer, there is no need for it to suspend. The EAM does in fact try to obtain extra parallelism by letting the producer continue. To avoid duplicating work, this is implemented by creating pointers from the producer to the consumer goal and not by creating a physical copy for every alternative of the producer. This copy of the consumer is now done in lazy fashion, hence the name *lazy copying*. The forking rule now becomes:

$$[\exists X : \{C_1 \vee ... \vee C_n \& G] \rightarrow \{[\exists X : C_1 \& G_p] \vee ... \vee [\exists X : C_n \& G_p]\}$$

where G_p is a reference to a single goal. One can do even better, by copying parts of G_p as soon as they suspend. To support this scheme the EAM now includes 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, global forking with lazy copying is performed. In contrast, global forking on non-guessable variables is only performed if there is no other way to restart the computation.

3 Data Structures

An efficient implementation of the EAM requires data structures capable of performing the four main operations of the EAM quickly and efficiently. We next present a first design for the data structures that we intend to use on the BEAM.

3.1 Local Variables

Each and-box may create several local variables, corresponding to each individual subgoal. Local variables are represented with the attributes shown in Figure 3:

- *Value*: the actual value of the variable, may be bound or unbound. The representation is done with the help of tags as used in WAM. Unbound variables are represented as self-pointing references.

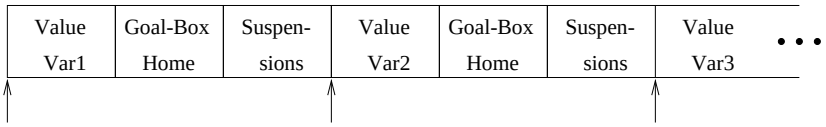

Figure 3: Local Vars representation.

- *Goal-Box_home*: pointer to the goal-box on the and-box where the variable was created.
- *suspensions*: pointer to a list of goals-boxes (inside and-boxes) that have suspended on this variable.

For each Goal we have a collection of Local variables. These variables are organised sequentially, in a vector, as in Figure 3.

3.2 External Variables

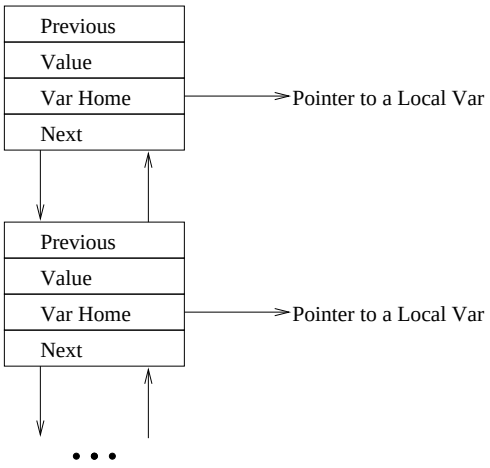

Figure 4: External Vars representation.

Accesses to variables external to the current and-box require special data structures. Each external variable has the attributes shown in Figure 4:

- *Value*: the value that has been assigned to the variable by the local and-box.
- *Var_Home*: pointer to the home of the local variable (as example consider one of the pointers in Figure 3)

If a goal changes a value to a variable that is not local, the assignment is recorded in an external variable struct and on the local variable. This way, whenever a descendent and-box wants to use the value of this external-variable, it can do it simply by accessing to the local variable directly. The external_variables structure is very important, as it allows to save possible assignments to variables that will be used ahead to know if the execution should proceed or not. They also work as an advanced kind of trail. If a assignment is made to a variable that belongs to a upper and-box, it's recorded in the External_Vars structure. If later the computation fails, or if the computations needs to suspend, the external struct will be used to recover all assignments made to non-local vars.

3.3 Or-Box

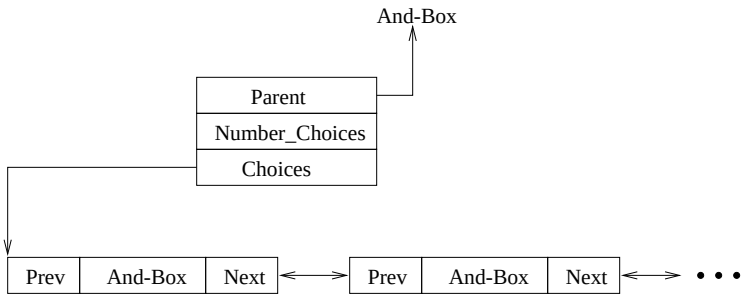

Figure 5: Or-Box representation.

Figure 5 shows how to implement an or-box. Each or-box has the following attributes:

- *Parent*: a reference to the parent goal that is represented in the upper level and-box.
- *Number_choices*: an integer that has two functions. The first is to indicate the number of alternatives that have been expanded. The second function is to, after all alternatives have been expanded, keep track of the number of choices still available. It works like this, imagine a clause with 4 choices. The field Number_choices is initialised with -4. Each time an alternative is expanded, the counter is incremented. When all alternatives have been expanded, this counter reaches 0. At this point it moves to the second function, and it's value goes to 4. When each alternative is removed, the counter is decremented.
- *Choices*: a pointer to a double-linked list of and-boxes

Note that each or-box contains a pointer, the Choices pointer, to a doubly-linked list of alternatives, each one being an and-box. This list must be double-linked because it has to support arbitrary insertion and deletion of and-boxes. Despite the fact that the integer *Number_choices* can be deduced from the doubly-linked list, it's representation is very important because it allows a quick access whenever we need to know the number of existent alternatives. For example, whenever the Number_choices field decreases to one, there is no need to keep the or-box. In this case, a promotion can be applied to that (only) choice so that the or-box disappears and the choice and-box goes up one level to be combined with the parent's and-box. The parent goal that was pointing to the or-box can be eliminated.

3.4 And-Box

And-boxes require a more complex structure, as they must store information on what goals are active, and on external and internal variables. Figure 6 shows an and-box with the following attributes:

- *Parent* : a reference to an or-box
- *Level* : a integer that indicates the level in the tree of the current and-box. If a and-box has level `n`, then the parent has level `n-1`, and the descendents have level `n+1`. This field is used to find the nearest and-box common to two and-boxes. For example, suppose that you have two and-boxes, A and B. Suppose that A is in level 4 and B in level 6.

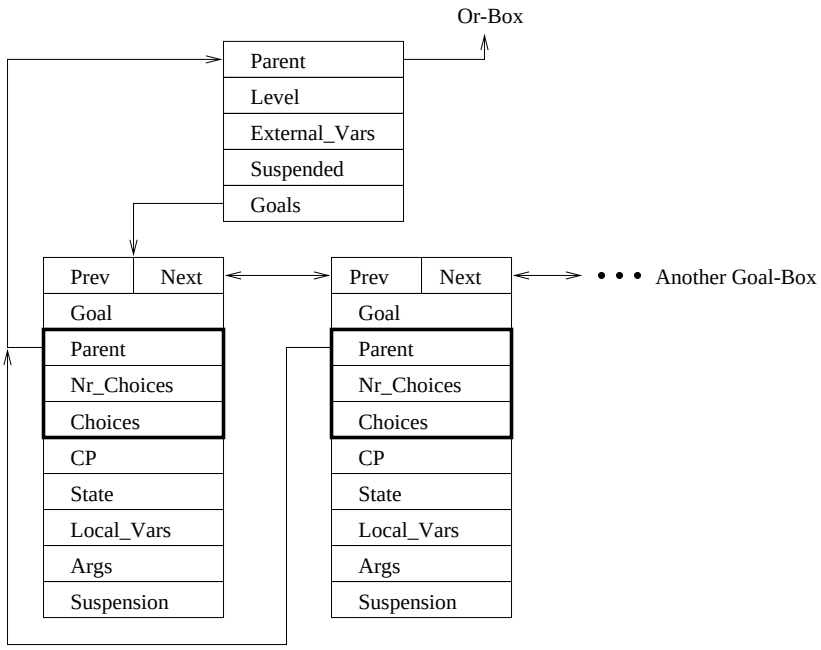

Figure 6: And-Box representation.

Then B can't be parent of A. Going up two levels in the tree of B, we get to know if A is parent of B. If not we can go up in the tree once in a time to find the common parent to both of the and-boxes.

- *External_Vars*: a pointer to a double-linked list of external variables as shown in Figure 4. These are the external variables used in the and-box. Note that all variables that are external in this and-box are necessarily local variables in some upper and-box.
- *Suspended*: this field is used to indicate when the and-box is suspended. If this value is false, then it means that it still exists work to be done in the current and-box. If this value is true, then it means that the current and-box has suspended.
- *Goals*: a pointer to a double-linked list of structures that represent the goals created in clause, and have information about a goal or one or-box with several alternatives. This structure, that we call goal-box, is divided in the following attributes:
  - *Goal*: pointer to the goal structure.
  - *Parent, Number_Choices, Choices*: these 3 fields represent an Or-Box that is created for the goal.
  - *CP*: sometimes the goal-boxes aren't all expanded at the same time. This field is used in conjunction with the pointer to the and-box to create the next goal-box.
  - *state*: can be:
    - *ready*: when the goal is ready to start execution
    - *active*: the goal is running
    - *fail*: the goal has failed
    - *success*: the goal has succeed

    - *suspended*: the goal has suspended in some point
    - *suspended_on_end*: the goal has suspended on the end. This is a special case of the suspended case. The general suspended case needs to try to continue the goal execution. This special case knows that the execution has already ended, so if the suspension can be activated, the goal can skip immediately to a success state.
  - *Local_Vars*: pointer to the start of a vector with the Local Vars, as shown in Figure 3.
  - *Args*: pointer to the start of the vector of arguments.
  - *Suspension*: pointer to a doubly-linked list of continuation record, each one with info about in which variable a goal suspended, and in which part of the goal.

Whenever there are goal-boxes suspended the and-box is not necessarily suspended. The and-box is only suspended when the field suspended is true. This field can only be true when at least one of the goal-boxes has suspended. Note that the Or-Box is physically included on the goal-box. This way the arguments can be accessed by the or-box directly.

4 Execution Model

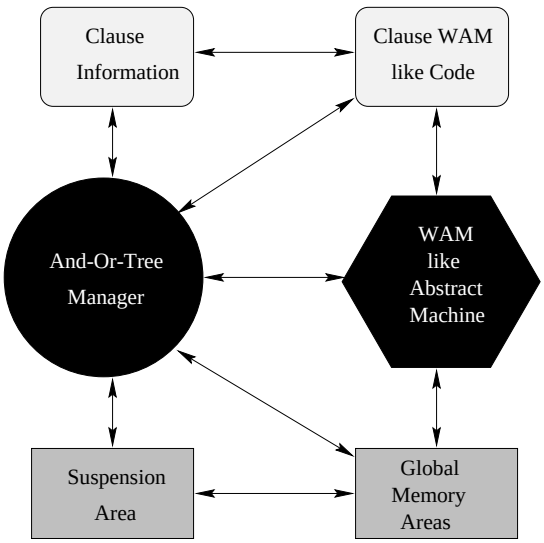

Figure 7: Execution Model.

There are six main components in the execution model of BEAM, as shown in Figure 7. They are:

- *And-Or-Tree Manager*: This module manipulates the and-boxes plus the or-boxes using the EAM rewriting rules, and starts WAM-like execution for the selected goal.
- *WAM like abstract machine*: This module runs WAM-like instructions. These include WAM instructions such as `puts`, `gets` and `unifys`. Instructions such as `calls` and `trys` may need to call the And-Or-Tree manager area if complex rewriting is necessary or if the current expansion must suspend.

- *Clause Information*: This area contains all the information related to the program's code. It contains all the information about the clauses available in the program. It includes the next area.
- *Clause WAM like Code*: It contains the clauses WAM like code.
- *Suspension Area*: This area is responsible for managing the suspended clauses or boxes of the EAM trees. The decision to wake suspended boxes is responsibility of the And-Or-Tree manager.
- *Global Memory Areas*: It operates like the heap in the WAM. Is used for saving variables values, compound terms like lists and structures, and even Prolog atoms. It is also used for advanced unification and most important, this area serves the and-or-boxes memory requests.

The execution model presented has two main areas. The first and most complex, is responsible for managing the and-or-box tree. It uses the Clause information area to know how many alternatives one clause have and how many goals one clause calls. Using this information it constructs the tree and uses the EAM rewriting rules to manage it. It request memory for the boxes in the Global Memory Areas, and it uses the Suspension Area if one of the boxes needs to suspend. The WAM like Abstract Machine is called by the And-Or-tree Manipulation 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 Manipulation transforms the `p(X,Y)` into one and-box with `g` and `f` as goals-boxes. During this creation it calls the WAM like Abstract Machine to manage the arguments of `g` and `f`.

The And-Or-Tree Manipulation area uses the EAM rewriting rules to work with the constructed tree. The most important rewriting rules are: the `Reduction`, the `Promotion` and `Forking`. Their implementation is explained individually in the following sections.

4.1 Reduction

The reduction operation expands a goal `P` into an or-box where each component of the or-box is the body of a clause whose head unifies with the goal. See example in Figure 8.

Note that in the example presented all the alternatives to `P` were expanded after the reduction, but in a real execution this could become very expensive. Sometimes to avoid overheads, the reduction first expands only the left alternative, and only when needed, it expands the second one, and so on.

4.2 (Deterministic) Promotion

If an and-box contains a or-box with only one alternative, then all the constraints within the lower and-box can be promoted to the upper and-box and finally the or-box can be removed. See example in Figure 9.

Note that some of the external variables to `P` (represented as `External_Vars2`) could be local in the parent and-box. So after the promotion this kind of variables wouldn't be in the external variables structure.

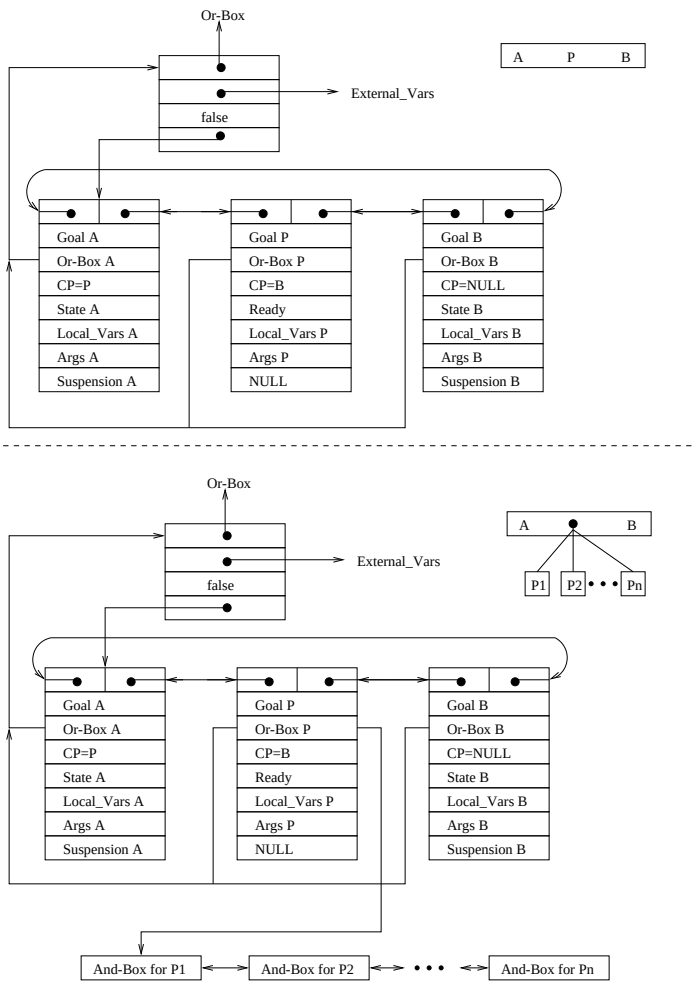

Figure 8: Applying Reduction.

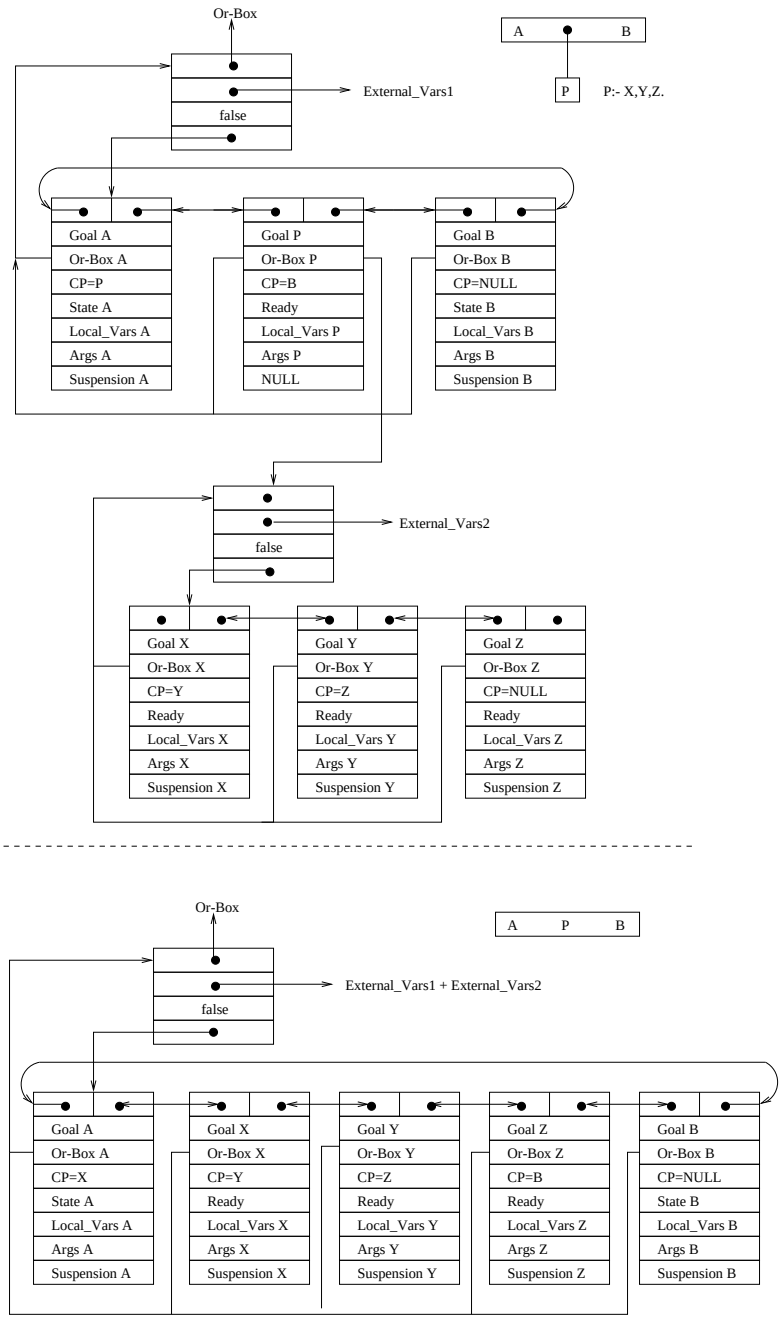

Figure 9: Applying Promotion.

4.3 Forking (Non-Deterministic Promotion)

This operation consists of distributing a goal across a disjunction. See example in Figure 10.

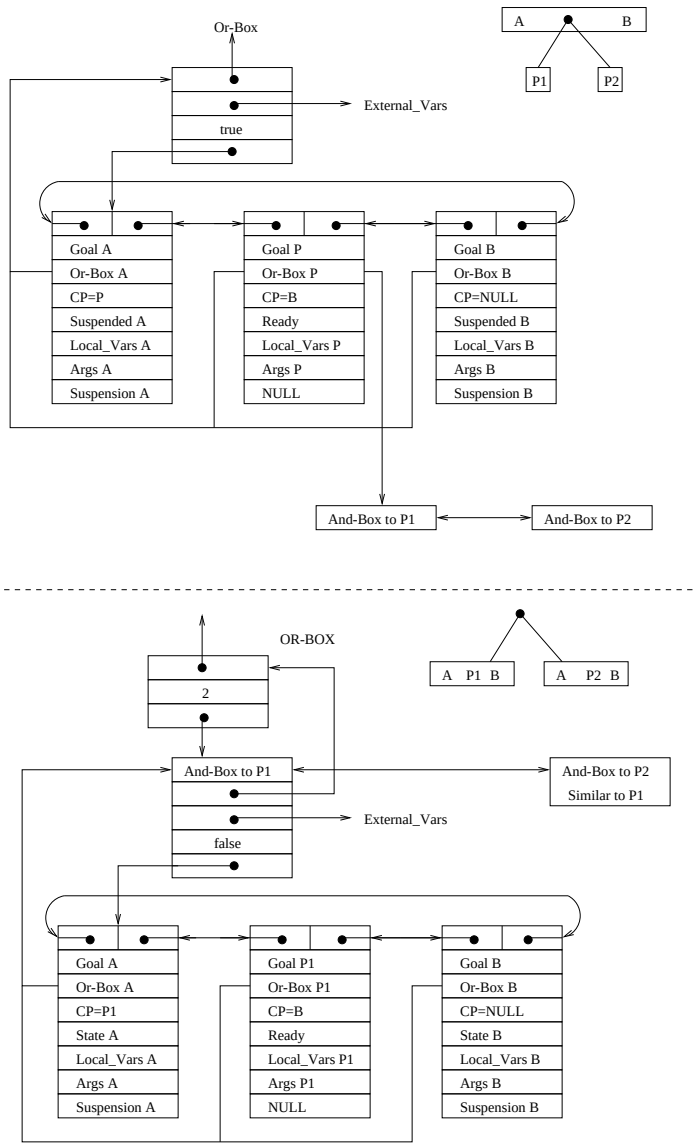

Figure 10: Applying Forking.

Note that some of the external variables to `P1` or `P2` could be local in the parent and-box. So after the forking this variables wouldn't be in the external variables structure. Note also that `A` and `B` are suspended, but after forking they can become ready if eventually they were suspended on variables that would change because of `P1` or `P2`.

5 Dealing with Non-Termination

In most cases the EAM should be able to significantly reduce the execution space for logic programs. Unfortunately, this is not always the case. In this section we present an example that shows this problem in the current definition of the EAM, and we propose a solution to avoid non-termination without the need for expensive global analysis or for user intervention. Consider the following Prolog program:

```
   ancestor(X,Y):- parent(X,Y).
   ancestor(X,Z):- parent(X,Y), ancestor(Y,Z).
   parent(a,fa).     parent(a,ma).    parent(b,fb).   parent(b,mb).
   parent(ma,mma).   parent(ma,fma).  parent(mb,c).   parent(mb,d).
   parent(fa,c).     parent(fa,d).    parent(fb,mfb). parent(fb,ffb).
?- ancestor(a,Z), ancestor(b,Z).
```

To simplify the notation we use `a(X)` for `ancestor(X)` and `p(X)` as `parent(X)` in Figure 11 and the following discussion. Figure 11 shows how the EAM leads to non-termination in this example.

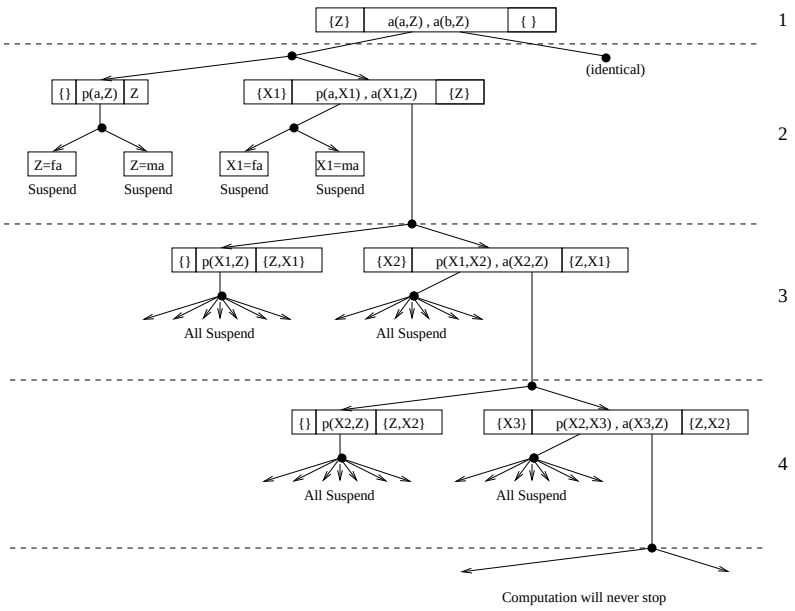

Figure 11: The ancestor example that leads to Non-Termination in EAM.

When the goal `p(X,Y)` is called, its execution is always suspended because an non-determinate assignment is needed. The only call that does not suspend is the goal `a(X,Y)`. This will lead to non-termination, since no value will ever be assigned. Warren and Gupta proposed a solution to this problem using Eager Non-Determinate promotion [1]. 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 very elegant since one of the aims of EAM is to obtain the maximum performance with the minimum control information from the user (Implicit Control). The solution that we propose to

solve these problems is the combination of Eager Non-Determinate promotion with tabling. 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 allows the reuse of goals. This would be particularly useful in exploiting independent and-parallelism. With the Eager forking and tabling the execution would change this way. On the level 3, the call `a(X2,Z)` would suspend because in level 2 was already made a similar call `a(X1,Z)`. Then, an eager forking could be made on `p(a,X1)`, assigning `X=fa` on one side and `X=ma` on the other side. This assignments would be sufficient to make the computations step out of the cycle, and eventually would lead to the solutions `Z=c` or `Z=d` (see Figure 12).

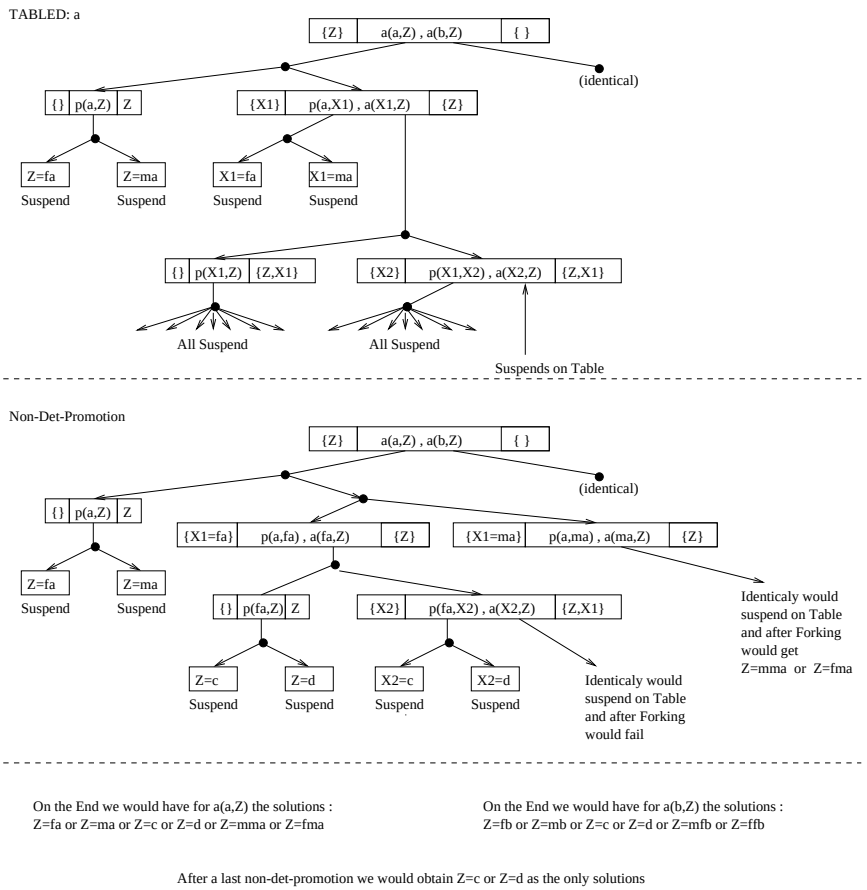

Figure 12: The ancestor example with tabulated goal.

Note that termination analysis should still be useful in detecting which goals would be required to tabulate.

6 Conclusions

In this paper we have presented a feasible data structures to be used in the BEAM, a first implementation of the EAM. We have also presented a novel solution to deal with Non-Termination on the EAM. We believe that this solution can also be useful to implement reuse of goals in the EAM. Implementation work has already started on the implementation of a sequential version of the BEAM.

Acknowledgments

The authors would like to acknowledge and thank the contribution and support that Fernando Silva gave to this work. Our work has been partly supported by JNICT under the Melodia project (JNICT contract PBIC/C/TIT/2495/95). Ricardo Lopes thanks PRAXIS (Praxis Grant BD/9277/96) for supporting his research.

References

[1] G. Gupta and D.H.D. Warren. An Interpreter for the Extended Andorra Model. Internal report, University of Brsitol, 1991.

[2] Sverker Janson. *AKL – A Multiparadigm Programming Language.* SICS Dissertation Series 14, Uppsala University, 1994.

[3] 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.

[4] Sverker Janson and Johan Montelius. Design of a Sequential Prototype Implementation of the Andorra Kernel Language. Sics research report, in preparation, Swedish Institute of Computer Science, 1992.

[5] Johan Montelius and Khayri A. M. Ali. An And/Or-Parallel Implementation of AKL. *New Generation Computing*, 13(4), 1995.

[6] V. Santos Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I.* PhD thesis, University of Bristol, August 1993.

[7] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[8] 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.

[9] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.

[10] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *The Journal of Logic Programming*, 19/20, May/July 1994.

[11] David H. D. Warren. The Andorra model. Presented at Gigalips Project workshop, University of Manchester, March 1988.

[12] David H. D. Warren. The Extended Andorra Model with Implicit Control. Presented at ICLP'90 Workshop on Parallel Logic Programming, Eilat, Israel, June 1990.