Universidade de Évora

Honesto Estudo com Longa Experiência Misturado

Back-end GNU Prolog para EAM (tabulação e distribuição)

Paulo Ricardo Lopes André

<prla@uevora.pt>

Orientador: *Prof. Doutor Salvador Abreu*

Évora, December 2, 2014

Mestrado em Engenharia Informática

Back-end GNU Prolog para EAM (tabulação e distribuição)

Paulo Ricardo Lopes André

<prla@uevora.pt>

Orientador: *Prof. Doutor Salvador Abreu*

Évora, December 2, 2014

Acknowledgments

First of all, I'd like to thank my supervisor, Salvador Abreu, for putting up with all my distractions, for doing his best to keep me focused when the going got tough and most of all for another great learning experience.

I would also like to thank my parents (and grandparents) for their unconditional love, for always believing in me and having provided me with everything I ever needed. No better parents exist.

To Cláudio, who I deeply respect, for accompanying me this whole journey of ten years. When there was something I needed to hear, he was there to tell me and that's what you can expect from a true friend.

And to *Rocky*, for having been the toughest little creature ever to grace the face of the planet. May you rest in peace.

Abstract

"GNU/Prolog Backend for the EAM (Tabling and Distribution)"

Logic programming provides a very high-level view of programming, which comes at the cost of some execution efficiency. Improving performance of logic programs is thus one of the holy grails of Prolog system implementations and a wide range of approaches have historically been taken towards this goal. Designing computational models that both exploit the available parallelism in a given application and that try hard to reduce the explored search space has been an ongoing line of research for many years. These goals in particular have motivated the design of several computational models, one of which is the Extended Andorra Model (EAM). In this thesis, we present a preliminary specification and implementation of the EAM with Implicit Control, the `WAM2EAM`, which supplies regular WAM instructions with an EAM-centered interpretation.

Resumo

A programação em lógica dá uma visão de muito alto nível da programação, o que acontece à custa de alguma eficiência na execução dos programas. Melhorar o desempenho dos programas em lógica é por isso um dos maiores objectivos das implementações de sistemas Prolog e um vasto leque de abordagens já foi utilizado no passado com vista a alcançar esta meta. Desenhar modelos computacionais que explorem tanto o paralelismo disponível numa dada aplicação e que tentam reduzir ao máximo o espaço de pesquisa tem sido uma linha de investigação muito activa durante muitos anos. Estes objectivos motivaram em particular a criação de vários modelos computacionais, entre eles o Extended Andorra Model (EAM). Nesta tese, apresenta-se uma especificação e implementação preliminares da EAM com Controlo Implícito, a `WAM2EAM`, que confere uma interpretação baseada na EAM a vulgares instruções WAM.

Contents

List of Figures

Chapter 1

Introduction

The work presented in this thesis is the result of a two years graduate masters student research which concerns itself with the general problem of enhancing the efficiency of logic program execution. In fact, this was spawned by one of the tasks of Project STAMPA at University of Porto[1] which aims at the first implementation of a system that fully supports tabled logic programs running within the David H. D. Warren's Extended Andorra Model environment. Breaking down this big task into two smaller ones, we chose to focus on the EAM side of the question initially, laying the groundwork to enhance the future compiler with tabling later on.

While initially STAMPA's task was geared towards integreating tabling with the BEAM, an early implementation of the EAM, plans shifted to a more radical approach in which we would take an existing Prolog compiler, GNU/Prolog, take its WAM output and compile with an EAM-centric execution in mind. Perhaps then a more appropriate title for this thesis might be "Casting The WAM as the EAM".

And in fact, part of the work described herein has already been published under that title in the proceedings of the Joint Workshop on Implementation of Constraint Logic Programming Systems and Logic-based Methods in Programming Environments, CICLOPS-WLPE'10, see [1].

---

[1]http://www.dcc.fc.up.pt/stampa/

1.1 Contextualization

Logic programming is an abstract and high-level view of programming in which programs are expressed as a collection of facts and predicates that define a model of the problem at hand and against which questions may be asked. The most well-known example of this paradigm of programming is Prolog, which has been sucessfully used in applications of many different areas. One line of work that has been followed to address performance issues is parallel execution: parallelism allows logic programs to transparently exploit multi-processor environments while extensions like co-routing, constraints and tabling go a long way towards reducing the problem's inherent search space. Some or all of these together act as the foundation on which to build more advanced techniques towards obtaining maximum performance.

From the experience gained in implementing the Basic Andorra Model, D.H.D. Warren made a more radical proposal, the Extended Andorra Model, or EAM [14], in which the conditions in which independent computations might be carried out are eagerly sought. In this thesis, we present a concrete implementation of the Extended Andorra Model, the `WAM2EAM`, which differs from other approaches taken in the past because we are compiling straight WAM code into C,[2] adopting an EAM computational model, resorting to GCC extensions.

1.2 Thesis Structure

This thesis is structured as follows: Chapter 2 presents a short survey on the road leading up to our current implementation as far as the EAM is concerned, from the Andorra Principle to the BEAM. Chapter 3 describes the EAM in more detail and lays down the theoretical groundwork of the `WAM2EAM` and delves more deeply into its practical implementation from WAM code compilation to the data structures and execution control of the EAM-based generated C code. Finally, Chapter 4 uses a concrete Prolog example and succintly describes its compilation and execution from start to finish.

---

[2]We are targetting C with GCC extensions, such as label values and indirect jumps.

Chapter 2

State of the Art and Related Work

A significant body of research on Logic Programming has been directed towards improving the performance of Prolog. One important line of research towards this goal is the exploitation of the different forms of implicit parallelism, present in Prolog programs. Several approaches have been devised over the years but we shall focus on the systems which allow for the transparent parallel goal execution, in particular the "Andorra" family of languages which includes Andorra-I, AKL and the BEAM.

2.1 The Andorra Principle

David H. D. Warren proposed the **Basic Andorra Model** (BAM),[1] geared towards the execution of logic programs, in which a goal is called *determinate* if it has at most one candidate clause. In this model, deterministic goals should be executed first, thereby reducing the nondeterminate "guesswork" to the minimum possible. Only then, once no deterministic goal remain to be executed, should a non-deterministic goal be selected for execution.

---

[1]Not to be confused with Van Roy's Berkeley Abstract Machine, used in the Aquarius Prolog system [7].

A system incorporating the Andorra Principle reduces the search space of logic programs by having deterministic goals execute first and only once, rather than have them re-executed several times in different points of the search space. This behavior is also known as "sidetracking." Also, as a desirable consequence, deterministic goals may generate constraints (bindings) which may further reduce the number of alternatives in other (nondeterministic) goals, possibly even making them deterministic.

Another interesting advantage is how all deterministic goals can execute in parallel, so long as they do not run into binding conflicts. Parallelism in the BAM comes in two flavours:

- *AND-Parallelism* - deterministic goals run in parallel

- *OR-Parallelism* - the exploration of different alternatives to a goal is done in parallel

The BAM may also alter the semantics of programs, in that the order of the solutions for a given goal may be different from that resulting from sequential Prolog execution. This may cause otherwise nonterminating programs to reach a solution.

There are, however, a few issues inherent to this sort of computational model:

- Finding which goals are deterministic can sometimes be difficult as predicates with more than one clause may actually have a single matching clause for a given query.

- Concurrency may break Prolog semantics, for instance by executing a pruning directive (e.g. cut) too early.

The best-known implementation of the Basic Andorra Model is Andorra-I [4, 3]. It exploits OR-parallelism and determinate dependent AND-parallelism while fully supporting Prolog, however, despite good results, the system is limited by the fact that co-routining and AND-parallelism can only be exploited between determinate goals.

Shortly after, Warren went further and proposed the **Extended Andorra Model** (EAM) which improved upon the ideas of the BAM, namely by trying to explore independent AND-parallelism. This lead to a two major approaches:

- **AKL**: The Andorra Kernel Language (AKL) [5, 8] was designed by Haridi and Janson and was the course followed at SICS. It concentrated on the idea that a new language was needed, based on the advantages of the EAM, which would subsume both Prolog and committed-choice languages. AKL distinguished itself by featuring an *explicit* control scheme, as programs were written using guarded clauses, where the guard was separated from the body with a sequential conjunction, cut or commit operator.

- **EAM with Implicit Control**: In contrast to AKL, David H. D. Warren and other researchers at Bristol worked towards an implementation of the EAM with implicit control. Its main goal was to take advantage of the Andorra Principle while alleviating the burden on the programmer.

2.2   The BEAM

The Boxed EAM (BEAM) is an implementation of the EAM design with implicit control, developed at University of Porto, Portugal [9, 10, 11, 12]. The beam's initial goal was to prove the feasibility of Warren's design for the EAM, and as a first step it concentrated on the original rewriting rules of the EAM, so formally it was defined through rewrite rules that manipulate AND-OR trees as well as simplification and optimization rules used to simplify the tree and discard boxes. It also made use of a general control strategy, which is used to decide when and how to apply each rule.

The main operations of the BEAM are:

- **Reduction** expands a goal G into and or-box.

- **Promotion** promotes constraints from the an inner AND-box to an outer AND-box.

- **Propagation** propagates constraints from an outer AND-box to the inner-boxes.

- **Splitting** distributes a conjunction across a disjunction.

Adding to these are a few simplification and optimization rules, all of which are described in [11].

Apart from AND- and OR-boxes, there's also another kind of box contemplated in the BEAM which is the choice-box. These are special OR-boxes created when the clauses defining a procedure include a pruning operator, generically designated by %. The original EAM supports two pruning operators, cut and commit.

The EAM tries to keep the control implicit as much as possible, contrary to AKL for instance. Therefore, in the BEAM, the control decisions are based exclusively on information implicitly extracted from the program. Moreover, one of the main goals of the EAM is to perform the least possible number of reductions to obtain the solutions to a goal. BEAM's control strategy is geared towards this goal.

The BEAM also does not attempt to do all the work by itself, instead relying on the output of an existing Prolog compiler, in this case YAP Prolog. The BEAM was built as an extension to YAP. It differs from the work reported herein in that the BEAM is meant to be an interpreter, whereas `WAM2EAM` takes WAM code and compiles it to C.

2.2.1 Non-termination

A central problem found by the developers of the BEAM was a consequence of EAM's execution scheme: as long as they do not bind any (external) variables, the EAM allows the early parallel execution of nondeterminate goals. In the worst case, this may lead to non-termination for certain recursive predicates. The proposed solution was based on both eager non-determinate promotion and tabling which, on the one hand guarantees that

the computation ends in programs that have finite solutions and on the other hand, with tabling, allows for the reuse of solutions to goals.

Chapter 3

The Extended Andorra Model and `WAM2EAM`

The Extended Andorra Model (EAM) is the foundation for the work we carried out with `WAM2EAM`. The ideia is to perform as much work as possible in parallel, exploiting all the avaliable forms of parallelism:

- *Or-parallelism*, related to exploring the various alternatives of any given goal.

- *Indendent AND-parallelism*, within a conjunction of goals that do not share any variables.

- *Dependent AND-parallelism*, between goals that *do* share variables.

The main extension of the EAM over the BAM is that non-deterministic goals are allowed to execute in parallel so long as they do not bind any external variables.

Our purpose is to provide a concrete implementation of the EAM with implicit control. It departs from existing work because it compiles regular WAM code into C, using an EAM runtime specification. Therefore, the biggest challenge and arguably the most interesting aspect of this work, is going from one paradigm (Prolog compiled onto the WAM) to a different one (EAM) with a single tool.

Based on a configuration AND-OR tree at all times, the way to evolve this configuration is by using one of several *rewrite* rules on it and an execution control scheme to manage the application of these rules. The next section describes each of these rules in detail.

3.1 Rewrite Rules

The rewrite rules are yet another variation on the usual scheme already found in other implementations such as AKL [6] or the BEAM [12].

A few comments on the notation used to express rewrite rules in `WAM2EAM`:

- The letters A_i and O_i stand for an AND-box and an OR-box, respectively.

- We shall use α_i to denote a (possibly empty) sequence of AND-boxes and ω_i for a possibly empty sequence of OR-boxes.

- The symbol . denotes the list constructor and is used to specify the structure of sequences or sets. It can be used to single out an element from a sequence (eg. $A.\alpha$ for a sequence of AND-boxes or $G_L.Ca$ for a sequence of goals that make up an AND-continuation).

- The symbol + is used to indicate a list append operation.

- The symbol ϵ is used to denote the empty sequence and applies to AND-boxes, OR-boxes and continuations.

- The symbol $\perp$ represents the FAIL box and applies to both kinds of boxes.

- Transitions are expressed as follows:

$$\frac{\text{(present configuration)}}{\text{(next configuration)}} \bigg|_{\text{(optional conditions)}}$$

  Should there be a precondition which must be satisfied for a rule to apply, it will be indicated on the right of the rule.

9

Rules with no preconditions apply whenever the present configuration matches the rule.

Note also that access to non-local stores is read-only: should a worker try to bind a non-local variable (ie. one which resides in an and-box different from the one it's presently executing in) it will block.

AND-try This rule augments an AND-box by progressing in the evaluation of its AND-continuation: it removes an item from the AND-continuation and acts upon it, creating a child OR-box with an empty set of descendent AND-boxes and an appropriate value for its OR-continuation.

$$\frac{\mathsf{AND}_{\mathsf{V}_\sigma}(\mathsf{Os}; \mathsf{G}_\mathsf{L}.\mathsf{Ca}; \mathsf{Cs})}{\mathsf{AND}_{\mathsf{V}_\sigma}(\mathsf{Os}' = \mathsf{OR}^\mathsf{L}(\epsilon; \mathsf{Co}).\mathsf{Os}; \mathsf{Ca}; \mathsf{Cs})} \tag{3.1}$$

Binding (successful binding) The binding rule (or *constraint imposition rule*) sets the value for a previously unbound variable. It occurrs in a situation similar to that of the AND-try rule, being different from it in that the subgoal to be tried is a simple store operation.

$$\frac{\mathsf{AND}_{\mathsf{V}_\sigma}(\mathsf{Os}; \mathsf{G}_{\mathsf{op}}.\mathsf{Ca}; \mathsf{Cs})}{\mathsf{AND}_{\mathsf{V}_{\sigma\theta}}(\mathsf{Os}; \mathsf{Ca}; \mathsf{Cs})}\Bigg|_{\mathsf{G}_{\mathsf{op}}\equiv\theta} \tag{3.2}$$

Where G_{op} binds a set of variables $\overline{\mathsf{X}} \subset \mathsf{V}$ via a substitution θ. This transition is expected to represent the bulk of what is performed in a clause's execution.

Suspension (external binding) The suspension rule can be applied under circumstances similar to those in which the binding rule applies: it would set the value for a previously unbound variable. It differs from the *successful binding* rule in that the variable which is being bound is external: this rule causes the operation to suspend.

$$\frac{\mathsf{AND}_{\mathsf{V}_\sigma}(\mathsf{Os}; \mathsf{G}_{\mathsf{op}}.\mathsf{Ca}; \mathsf{Cs})}{\mathsf{AND}_{\mathsf{V}_\sigma}(\mathsf{Os}; \mathsf{Ca}; \mathsf{Cs} \cup \{(\overline{\mathsf{X}}, \theta)\})}\Bigg|_{\mathsf{G}_{\mathsf{op}}\equiv\theta\wedge\overline{\mathsf{X}}\not\subset\mathsf{V}} \tag{3.3}$$

10

Where $\mathsf{G_{op}}$ attempts to bind a set of variables $\overline{\mathsf{X}} \not\subset \mathsf{V}$ via a substitution θ. θ then becomes suspended on $\overline{\mathsf{X}}$ and the pair $(\overline{\mathsf{X}}, \theta)$ is added to the set Cs.

AND-collapse (failed binding) Analogous to the binding rule, there is a transition which is applicable when the attempted store operation fails:

$$\frac{\mathsf{AND}_{\mathsf{V}\sigma}(\mathsf{Os};\mathsf{G_{op}}.\mathsf{Ca};\mathsf{Cs})}{\bot}\Bigg|_{\mathsf{V}\sigma \nvdash \mathsf{G_{op}}} \tag{3.4}$$

$\bot$ represents a *failed AND-box*. This transition is taken whenever the attempted operation $\mathsf{G_{op}}$ fails in the store provided by V and σ.

OR-try Similarly to AND-try, this rule augments an OR-box by progressing in the evaluation of its OR-continuation.

$$\frac{\mathsf{OR}^{\mathsf{K}}(\mathsf{As};\mathsf{G_o}.\mathsf{Co})}{\mathsf{OR}^{\mathsf{K}}(\mathsf{As}\cup\{\mathsf{A}\};\mathsf{Co})} \tag{3.5}$$

Where $\mathsf{G_o}$ is the first element of the OR-box's OR-continuation and A is the AND-box created by $\mathsf{G_o}$. Co represents the remainder of the OR-continuation. Initially we'll have $\mathsf{A} = \mathsf{AND}_\emptyset(\epsilon;\mathsf{Ca};\epsilon)$, i.e. the OR-box's context is passed on to the new AND-box, which starts its existence with no variables or bindings, no OR-box children and no suspended continuations.

Promotion This rule applies when an OR-box has a single successor node, which is then moved into the OR-box's parent node. More formally:

$$\frac{\mathsf{AND}_{\mathsf{V}\sigma}(\mathsf{OR}(\mathsf{AND}_{\mathsf{V}'\sigma'}(\mathsf{Os}';\mathsf{Ca}';\mathsf{Cs}'));\mathsf{Ca};\mathsf{Cs})}{\mathsf{AND}_{(\mathsf{V}\cup\mathsf{V}')\sigma\sigma'}(\mathsf{Os}';\mathsf{Ca}.\mathsf{Ca}';\mathsf{Cs}\cup\mathsf{Cs}')} \tag{3.6}$$

The promotion rule is contracting and it ensures that configurations remain shallow. The context from the AND-box which is being promoted is transfered onto the corresponding AND-continuations and suspensions.

Split This rule is the basis for non-determinism in `WAM2EAM` and can be thought of as a rule for distribution of ANDs over ORs:

$$\frac{\mathsf{OR}(\alpha_0 \mathsf{AND}_{\mathsf{V}_\sigma}(\omega_1 \mathsf{OR}(\alpha_1 \alpha_2)\omega_2)\alpha_3)}{\mathsf{OR}(\alpha_0 \mathsf{AND}_{\mathsf{V}_\sigma}(\omega_1 \mathsf{OR}(\alpha_1)\omega_2)\alpha_3 \mathsf{AND}_{\mathsf{V}_\sigma}(\omega_1 \mathsf{OR}(\alpha_2)\omega_2)\alpha_3)} \quad (3.7)$$

This is the "classical" choice split as found in AKL and the BEAM, in which an OR-box has one of its successor AND-boxes singled out, thereby making it a candidate for promotion by rule 3.6. The present formulation for this rule allows for different strategies to be selected, depending on the lengths of α_1 and α_2: should one of these be of length 1 it will be selected for promotion. In order to preserve Prolog-like solution order, $|\alpha_1| \leq |\alpha_2|$ must hold.

The AND-box containing α_2 as is a copy of the original AND-box containing $\alpha_1 \alpha_2$. The contexts are also copied to wherever it is appropriate.

3.2 The `WAM2EAM`

The major challenge in `WAM2EAM` certainly is to go from a WAM program and re-interpret it from an EAM point of view. To accomplish that, we take the GNU Prolog's textual WAM output and proceed from there. The idea is to generate C code for an EAM runtime. This entails doing things quite differently from previous work such as WAMCC [2] or B-Prolog [13]. `WAM2EAM` has two major aspects to it:

1. the compiler, comprising the parser and the C code generator,

2. the runtime, a collection of data structures, logic and execution control that implements the EAM execution model.

However, this is not enough to actually get answers from a Prolog program. `WAM2EAM` acts an intermediate step in the Prolog compilation pipeline, coming in between GNU/Prolog and GCC as fig. 3.2 ilustrates.
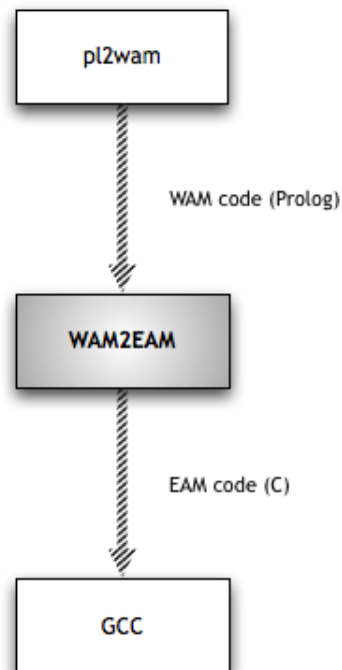The remainder of this section discusses design and implementation of the compiler and runtime.

Figure 3.1: Where `WAM2EAM` fits.

3.3 Parsing WAM instructions

We used GNU Prolog because its compilation passes are fairly simple and it is easy to materialize the WAM representation of Prolog programs. The following is a snippet of code which is the GNU Prolog WAM representation of the `p/1` predicate from the earlier example.

```
predicate(p/1,5,static,private,user,[
    allocate(1),
    get_variable(y(0),0),
    put_value(y(0),0),
    call(q/1),
    put_value(y(0),0),
    deallocate,
    execute(r/1)]).
```

Alas, in the name of efficiency and convenience later on and a more hands-on approach, we chose to go with C so to parse GNU/Prolog's WAM output we need a different route. The obvious candidate is the tried and true flex/bison tag team, which allows for flexible scanning and parsing which integrates well with custom C code. Despite a few extraneous meta bits of information to be found in the intermediate WAM representation, the bulk of the work is extracting the actual WAM instructions so that is the one major concern of the scanner. Other than that, every atom is internalized with a convenient C representation for further processing later on. Instructions are special in that they have arguments, varying in number and type, which must be parsed so a corresponding C WAM instruction representation must to be built. This is accomplished using a complete WAM instruction Bison grammar and allocating a tailor-made C structure for each instruction we find along the way, including its arguments. Therefore, the parsing stage comprises two major, consecutive steps:

1. For each predicate, we internalize every instruction found on the WAM intermediate representation - that is, building an appropriate C representation for each WAM instruction. This act as a staging area for the next step.

2. Generate a pattern of C code for each WAM instruction we find.

3.4   C Code Generation

An interesting aspect of `WAM2EAM` is how its input is pure text, just like its output is pure text. In particular, it compiles the Prolog WAM intermediate representation into C code, switching to an EAM perspective in the meantime. To accomplish this, the idea is to generate a small and well-defined pattern of C code for each WAM instruction, this being the vehicle to generate an EAM-based representation of the source program. For instance, while a pure WAM implementation would look at a choice-point manipulation instruction such as `try_me_else` as an order to produce... well, a choice point, `WAM2EAM` insteads subverts this perspective and creates the EAM analogue, in

14

this case setting up an OR-box alternative using the label given as argument. Be that as it may, a lot of the WAM instruction set translates as-is to the EAM representation. Simpler instructions, such as `put_value` for instance, are supposed to do exactly the same thing in the WAM and in the EAM and the same goes for indexing instructions like `switch_*`. In a few cases, such as `proceed`, `WAM2EAM` simply disregards the instruction as not being useful in the EAM setting.

At closer inspection of the WAM instruction set, the major difference in paradigm impacting the C code generation concerns the instructions dealing with non-determinism. Whereas the WAM deals with choice points, creating and destroying them as needed, the EAM, by doing away with the WAM's stack-based representation and using an AND-OR tree based configuration instead, deals with OR-boxes when it comes to setting up and exploring alternatives.

Once every detail of the original program has a C representation – an abstract parse tree – the idea is to walk through it and emit a bit of C for each predicate and for every WAM instruction inside it. For each internalized predicate, a block of C code is generated, setting up a new AND-box which contains a suitable number of allocated local variables,[1] binding those variables to its parent OR-box corresponding predicate arguments and defining each of those variables' *home* as the very AND-box that is being created. The output code is generated by this code in the compiler:

```
emit(8, "a = new_and_box (o, %d, ab_id++);\n", max_var_idx+1);
for (i = 0; i < n; i++)
  emit(8, "bind (a->locals[%d], o->args[%d]);\n", i, i);
for (i = 0; i < max_var_idx+1; i++)
  emit(8, "ASREF(a->locals[%d])->home = a;\n", i);
```

`max_var_idx` reflects the maximum number of variables used in this predicate, accounting for possible temporaries in all of its clauses, potentially a single one if deterministic. Looking now at the C code for a clause with two local variables, it might look something like this:

---

[1]The exact number is determined by inspection of the WAM code in the body.

```
a = new_and_box (o, 2);
bind (a->locals[0], o->args[0]);
bind (a->locals[1], o->args[1]);
```

This allocates a new AND-box with two local variables, as a child of the
current OR-box (whose address is kept in `o`) and both of those variables
are then immediately bound to whatever are the first two parent OR-box
arguments. This creates variable chains across the AND-OR tree, reflecting
the same concept found in Prolog clauses where a newer variable might refer
to an older one.

A second pass through the WAM instructions for the clause is needed to
generate code for each actual WAM instruction by traversing the list built
by the parser.

```
while (instrs) {
  print_instr (instrs->head, (*a)->name, n, max_var_idx+1, FALSE)
  instrs = instrs->tail;
}
```

`print_instr` then goes through a large `switch` instruction that finds the
appropriate bit of C code to emit for each WAM instruction, having the
EAM execution scheme in mind. WAM instructions, which by now we regard
as EAM instructions in their own right, are roughly divided in three major
groups:

3.4.1 Choice point manipulation

These are the `try*`, `retry*` and `trust*` instructions. We no longer think
in terms of choice point frames, instead looking at managing non-determinism
by way of OR-boxes. A predicate with only one clause consists of an OR-box
with a single alternative (and thus a single descendant AND-box) whereas a
non-deterministic predicate (ie. having more than one clause) is translated as
an OR-box with as many children AND-boxes as there are possible clauses.
A more in-depth description of how OR-boxes actually deal with alternatives

will be given after we introduce the major data structures used throughout
`WAM2EAM`. In practice, an instruction like `try_me_else (L)` (or `retry_me_else`
(L), for that matter) for predicate `q(1)` simply defines the next alternative
in the current OR-box, generating the following bit of C code:

```
o->alt = &&P_q_1_C4;
```

3.4.2   Execution Control

The `call` and `execute` instructions are responsible for predicate calling,
in effect jumping to the appropriate place in the code where to start executing
the called predicate. They also need to setup a return address for when this
predicate is done executing. This is accomplished by emiting a C label and
configuring the current AND-box continuation to that label, using GCC's
label address extension. With this, once the called predicate is done, it
will `proceed` to whatever AND-continuation is available in its AND-box, in
effect returning here and resuming execution. The difference between `call`
and `execute` is precisely what to do after the called predicate is done with.
Whereas in the former case, it simply continues executing whatever is left in
the current predicate, the latter means this was the last goal in the current
clause and it should look for a continuation above, in the Prolog execution
chain. Here's how the `call` instruction is translated to C: For example, the
pattern of code generated for calling the goal `q(X)` in our example is:

```
/* call(q/1) */
q_enqueue(a->and_conts,&&R1); // setup AND-continuation
o = new_or_box(a,1);          // create new OR-box
o->args[0] = a->locals[0];    // preload A registers on the new OR-box

goto P_q_1;                   // jump to the predicate's code

R1:                           // return label
/* further code.. */
```

3.4.3 Variable manipulation and unification

This type of instructions is also handled quite differently within the EAM. Simple instructions such as `put_value` or `get_variable` are basically the same, but unification needs to be looked at more carefully, as trying to bind variables which are not local to the current AND-box leads to suspension of execution and triggers a search for work, elsewhere in the code. AND-box suspension and the `WAM2EAM` execution scheme will be looked upon in a bit more detail shortly.

3.5 Generated code structure

Since we're generating a valid C program, ready to be compiled by GCC, there's a question of what layout this code will use. One important constraint is that we must be able to jump back and forth between different predicates, in order to implement predicate calling and returning. Also, we need to jump to random places in the code when attempting to resume a suspension. Considering that it is illegal to use C's `goto` between different functions,[2] generating one C function per predicate is not an option, no matter how tidy and comfortable that would be from a structure point of view.

One possible alternative then is to implement the entire program as a single function and delimiting predicates using unique labels. This way, jumping from one point in the code to another remains within the bounds of the one function and correct indentation when emitting the code will hopefully not make it a burden to look at. We also must be careful when jumping to a point of code from out of nowhere, since the correct environment must be replaced, namely the current AND- and OR-boxes. Other than that, all it takes for jumping around the code is the address to jump to and making good use of GCC's labels as values extension.

```
int program ()
{
```

---

[2]We may not reenter an existing C stack frame.

```
/* ... */
P_p_1: {
    a = new_and_box(o,1);
    /* ... */
    o = new_or_box(a,1);
    goto P_q_1;

/* ... */
P_q_1: {
    a = new_and_box(o,1);
    /* ... */
}
```

3.6   Runtime Data Structures

The runtime half of `WAM2EAM` is itself broken into two major steps and these are where we significantly depart from the WAM way of doing things and completely focus on EAM. First, executing the C code previously generated by the compiler will incrementally build the *configuration*, an AND-OR tree that gets constructed, modified and pruned as execution of the code proceeds. The way for this to happen is by applying in turn the different AND-OR tree rewrite rules.

The most important data structure in `WAM2EAM` is the AND-OR tree, also known as the *configuration*. An AND-OR tree is so called because it is composed of two kinds of nodes: the AND nodes, corresponding to Prolog clauses and the OR nodes, consisting of Prolog goals. We'll shortly get into more detail on how both these nodes are structured and how they interact with each other. For now, it's important to note that no two nodes, or *boxes*, of the same type are directly connected in an AND-OR tree, so any path from the root to any leaf is always made of alternate types of boxes. A parent OR-box has AND-box children, each of which has descendent OR-boxes, and so on. Moreover, the root is always an OR-box.

3.6.1 AND-boxes

They represent clauses, so there is one AND-box in the configuration for every clause in the Prolog source code. So, for instance, a non-deterministic predicate having four different clauses, would consist of four AND-boxes, one for each clause. AND-boxes are a lengthy structure in `WAM2EAM` in that they play a critical role. They are home to the clause's local variables, they need to keep track of their continuations (e.g. where to find the code for the next goal in the clause once the current goal is done with) and they also may or may not be suspended at any point in time. Finally, promotion also impacts AND-boxes directly, so they also have mechanisms to deal adequately with that. And, of course, they spawn (and in turn descend from) OR-boxes corresponding to the reduction of their body goals. AND-boxes are represent internally using the following C `struct`:

```c
struct _andbox {
  orbox *parent;
  int nlocals; // number of local vars
  term **locals; // array of vars local
  int num_or_boxes;
  orbox **or_boxes; // child OR-boxes
  queue *and_conts; // AND-continuations
  void *susp;
  int is_susp;
  andbox *clone;
  andbox *head; // top of this box's grouping
  andbox *rest; // next on this box's grouping
  int id;
};
```

An important aspect of AND-boxes is their AND-continuations, meaning the addresses in the code where execution should proceed in this box. AND-continuations are needed when a goal suceeds and more work needs to be

found in order for execution to continue. In other words, AND-continuations are used up when a called goal terminates and control goes back to the callee. This is accomplished through the implementation of the `execute` WAM instruction, which sets up a return label right *after* the jump to the called predicate code.

```
q_enqueue(a->and_conts,&&R2); // setup continuation
o = new_or_box(a,1); // create new OR-box

// preload A registers on the new OR-box
o->args[0] = a->locals[0];

goto P_r_1; // jump to r/1

R2:
    (...) // further code
```

3.6.2   OR-boxes

These represent goals and are created everytime a new goal is executed. Their primary concern is dealing with non-determinism by managing goal alternatives, namely holding an address for the next alternative for the current goal at all times. They also carry the goal's arguments when the goal gets called in order to pass them initially to each clause's AND-box as initial values. OR-boxes thus spawn an AND-box for each clause they invocate. Their representation on the WAM2EAM is as follows:

```
struct _orbox {
  andbox *parent;
  void *alt; // next alternative
  andbox **and_boxes;
  int num_and_boxes;
  term **args;
  int arity;
```

```
  orbox *clone;
  int id;
};
```

3.6.3   Suspensions

As we have seen before, caution must be taken when an attempt to bind a variable is made. Only in case the variable is local to the current AND-box will binding be allowed to occur. Otherwise, the AND-box is said to be suspended on the offending variable and execution proceeds elsewhere, namely to the next alternative in the current OR-box. Execution can only return to this AND-box when certain conditions are met, namely when the variable becomes local to the current AND-box or it gets bound from elsewhere. In the latter case, when the suspension is resumed, the attempted binding that triggered the suspension in the first place is retried and it either checks OK or it fails against the prevailing (earlier) binding.

In order to correctly deal with these situations, we need to wrap instructions wherein a suspension might occur with some code that actually checks for "offending" binding attempts, namely trying to bind a non-local variable. We do this by having every unification instruction check whether the dereferenced variable is already bound and if not, whether it is local or external to the current AND-box. The result of this verification is then returned as a meanigful code to a wrapping CHECK() macro, which then acts accordingly. Faced with a unification attempt, the outcome can then be any one of:

BIND_SUSP the variable is not bound yet and it's not local to the current AND-box either. The current AND-box suspends on this variable.

BIND_OK the variable is not bound and it's local, so the binding succeeds.

CHECK_OK the variable is bound and its value is the same as the one being attempted in the binding, so execution may proceed.

CHECK_FAIL the variable is bound and its value differs with the one being tried. The configuration branch rooted in the current AND-box fails and is pruned off the tree.

Because of suspensions, for every non-trivial program it's easy to see that we quickly arrive at what we call a *stuck configuration*, an AND-OR tree where all leaf AND-boxes are suspended. As we don't stop execution anytime a box suspends, it is only when no more code is left to execute that we have a problem. At this time we try to apply one of the rewriting rules, in particular giving priority to determinate rules such as determinate promotion. By promoting an inner AND-box into an outer AND-box, the variables inside it are also promoted which means they become closer to the AND-box where they will actually be local, eventually allowing for bindings to happen or suspensions to resume.

3.7   Deterministic Promotion

As explained in the previous section, actions (or rules) that *contract* the configuration are desirable. On the other hand, expanding goals also expands the configuration, as AND-boxes give way to OR-boxes which in turn give way to more AND-boxes and so forth. Deterministic promotion, being the only rule that eliminates boxes, is highly sought after. This rule is only applicable to OR-boxes with a single alternative.

Implementation-wise, promoting an AND-box context (variables, suspensions and continuations) into another requires maintaining their environments coherent. In other words, if the resulting AND-box contains the union of both sets of locals variables from the two AND-boxes involved in the suspension, then what was the first variable in the inner (promoted) AND-box is probably no longer the first variable in the outer (resulting) AND-box after promotion. This lends itself to all kinds of mayhem when code still refers `a->locals[0]` (WAM register `X(0)`) when the actual variable is now at `a->locals[1]`.

To cope with this problem, we opted to introduce the concept of AND-box groupings. Each AND node in the configuration is actually a group of one or more complete AND-boxes, forward-connected among themselves by a pointer which indicates the next box in the group. Moreover, every box in the group is also linked to the first - the *head*. This situation is illustrated in
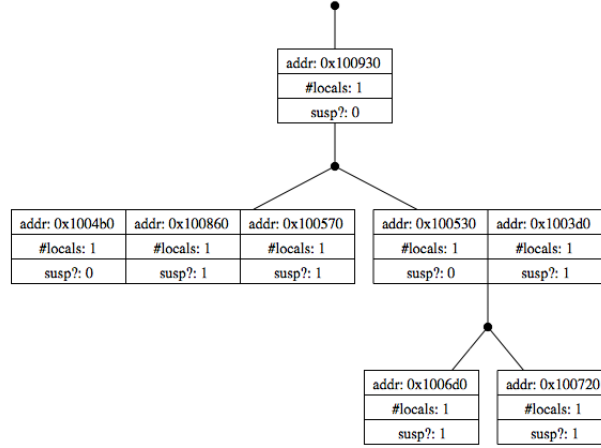
23

figure 3.2.

Figure 3.2: On the left: an AND-box grouping made of 3 different AND-boxes.

This way, each box environment remains pristine, as originally constructed, and it's safe to resume from a suspension point as far as accessing local variables is concerned. It's important to note that a variable is local to the current AND-box if, after dereferencing, its home AND-box is in the same group, i.e. has the same head.

3.8   OR-split and non-deterministic promotion

Desirable as deterministic promotion might be, its occurence is heavily constrained as we have shown in the previous section. The OR-box must have a single alternative and for predicates with multiple clauses that's frequently not the case. It is quite common for a configuration to get stuck with no chance for deterministic promotions to occur. When it comes to this, there is no other choice than to perform what we call an *OR-split* which forces a situation where a determinate promotion may happen.

Simply put, we elect an OR-box with more than one alternative to act as the root of a subtree to be *cloned*. In the original subtree, only one alternative

remains, while in the cloned subtree, *every other* alternative is present. This way, all alternatives remain in the overall configuration, ensuring correctness of the program, yet an opportunity for deterministic promotion now exists. Note that if the selected OR-box contains only two alternatives, we arrive at the special case where the OR-split induces two different deterministic promotion possibilities: one in the original box and another in the cloned box.

The choice of OR-box to split may be guided by heuristics, yet at this early stage we're simply going with the leftmost OR-box suitable for splitting. Also, from the chosen box's alternatives, we're picking the leftmost one to remain in the original branch and all others to be moved to the cloned subtree. Actual cloning is thus only needed for the parent AND-box and any siblings of the chosen OR-box. OR-split is the least desirable rule, because with cloning entire branches of the tree, it quickly becomes expensive.

3.9   The scheduler

The need to decide which rule to apply led to the implementation of a scheduler. This scheduler is called the first time after all alternatives and continuations are exhausted and no answers were produced. In other words, when the tree is stuck we ask the scheduler for guidance.

The implementation of the scheduler is part of the runtime code and is implemented as a C macro. It basically follows a hierarchy of possible events and acts accordingly for each outcome. First of all, in the event that a variable that had suspensions got bound, it tries to resume from any suspension pending on that variable. If none are found, it looks for an alternative in the current OR-box. If found, it continues execution from there, otherwise it tests the tree to see if it's stuck. If it is, it tries to apply deterministic promotion in order to try to move on or, if that fails, it resorts to applying non-deterministic promotion, by way of an OR-split. Putting this as the last choice makes sense, because it is also the most expensive operation.

It's interesting to note the reason why the scheduler is implemented as a

macro instead of a function, despite being a little involved and lengthy, it is because it may involve jumping to any point in the code, be it a suspension point, a continuation or an OR-alternative. Again, we are faced with the problem of not being able to jump between different C functions, so its being a macro is sufficient. The control flow for the scheduler is depicted in figure 3.3.
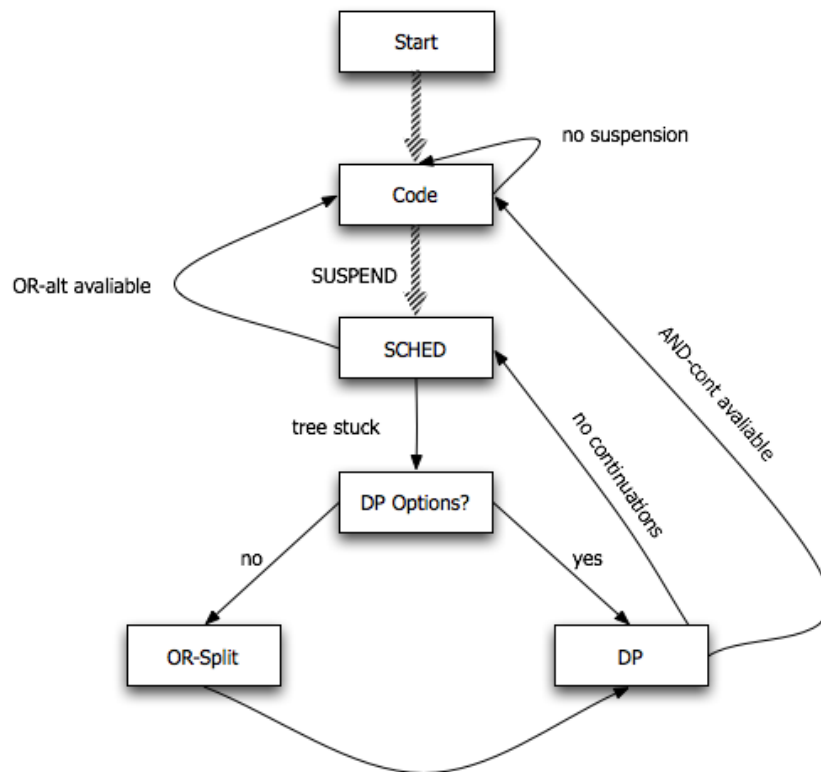

Figure 3.3: The scheduler's flow diagram.

Chapter 4

Example Execution

In this chapter, we'll use a running example based on the sample Prolog code shown in figure 4.1.

```
main :- p(X).

p(X) :- q(X), r(X).

q(1).  q(2).
r(2).  r(3).
```

Figure 4.1: Sample program

Keeping this example in mind, we now give an overview of how `WAM2EAM` goes from WAM code to EAM execution.

As previously mentioned, `WAM2EAM`-produced C code, when executed, comprises two different phases. The first one is made of consecutive reduction steps, expanding the AND-OR configuration as execution continues through the code. For each called predicate, a new OR-box is allocated, spawning a child AND-box for each of the predicate's clauses. Only trivial examples won't lead to suspension, as variable chaining between different predicates immediately induces external variables on some AND-boxes. This means that, at first, almost any binding attempt will lead to AND-box suspension, forcing execution to look elsewhere in the code, namely in the current OR-box's next alternative. So, in our example, it's easy to see how variable X
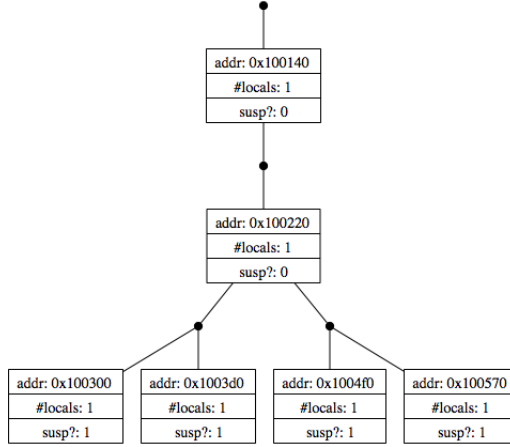
Figure 4.2: Stuck configuration.

is only local to the AND-box corresponding to the only clause for the main predicate and thus every fact for `q` and `r` will lead to suspension over X. After all clauses are executed, we get to a *stuck configuration* as seen on figure 4.2.

Implementation-wise, we rely on a function that takes the configuration's root OR-box and determines recursively whether the configuration is stuck. If every leaf node is suspended then we're faced with a stuck configuration indeed. Then the scheduler needs to choose one of two paths. It either performs deterministic promotion, if there are nodes ready to have this type of promotion performed on them, or it chooses to apply non-deterministic promotion somewhere in the tree:

```
if (is_stuck(root) && !q_empty(prom_ready))
    DP();
else
    NDP();
```

From now on, the configuration is modified by repeated application of rewriting rules, managed by the `WAM2EAM` scheduler. In this case, as no OR-box contains a single (suspended AND-box) alternative, no deterministic promotion is possible, so we need to resort to applying the OR-split rule on the leftmost OR-box, parent to suspended AND-boxes.
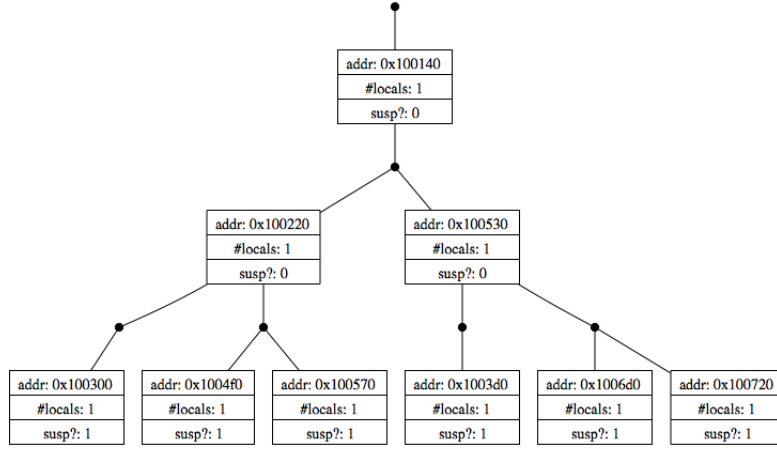
Figure 4.3: After OR-split.

Because there were only two alternatives to the split OR-box, it means one stays in the original branch while the other is moved to to the cloned branch and two determinstic promotions spots now exist (Figure 4.3). Were there more alternatives and only one deterministic promotion opportunity (in the original branch), the cloned branch would hold two or more alternatives, and thus not be ready for deterministic promotion. So we apply deterministic promotion to the leftmost AND-box, resulting in the configuration shown in figure 4.4.

After promotion, we attempt to restart the promoted and previously suspended box, but it immediately suspends again as its local variable, when dereferenced, still belongs to an another AND-box - the `main` AND-box in this example. Applying promotion to the analogous case in the cloned branch leads to exactly the same outcome, so we're again at a stuck configuration scenario. From here on, it's easy to see that repeated application of the `WAM2EAM` rules will result in a sequence of *OR-split, promotion, suspension* until an AND-box suspension is restarted and the variable in it that caused suspension is finally local to the current AND-box group. When that happens, the binding succeeds (or fails) and every AND-box suspended on this variable is "awakened". Then, there are two possible outcomes:
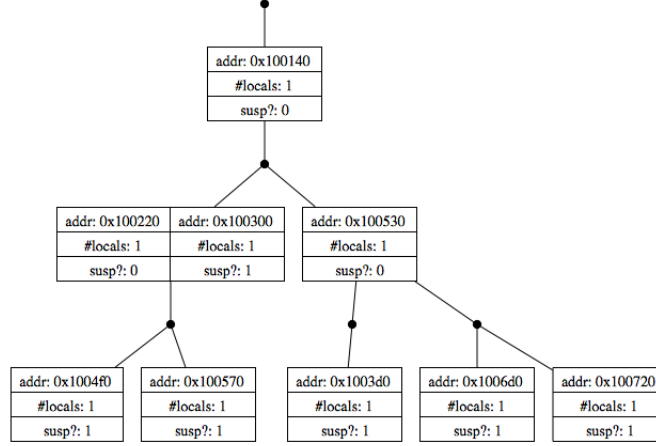
Figure 4.4: After deterministic promotion.

- **CHECK_OK** – the attempted binding at the suspension point unifies with the one already in place. This generates an answer to the program query and in our example that answer is `X=2`.

- **CHECK_FAIL** – the attempted binding fails to unify and that means this entire branch rooted on the current AND-box group, simply *fails* and is pruned from the configuration. Execution then looks to the scheduler for where to proceed.

As the EAM (and in turn `WAM2EAM`) does not contemplate explicit backtracking, the way to generate other answers for any given program, is to continue exploring different branches of the configuration looking for other successful bindings. In this case, none could be found as the other branch would also have a conflicting binding, leading to its pruning off the tree.

Chapter 5

Concluding Remarks & Future Work

We are convinced that our goal of generating a program following EAM semantics from a classical WAM one has been met, even if with some restrictions for the time being. Performance is not yet an issue but will become one as we develop further aspects of this implementation. It is interesting to see that it is feasible to have an EAM execution model without the Prolog compiler being aware of the fact.

Further work is to focus on the introduction of pruning operators – in the case of *cut* this is straightforward to recognize from the WAM code but for *commit* special measures will have to be taken as it is not inherently accounted for by the Prolog-to-WAM compiler of GNU Prolog.

One of the driving motivations for generating AND-OR trees and having them manipulated as per the EAM was to bridge this computational model to one with tabling, as found in XSB or YAP Prolog. Although we haven't begun to do so, this goal remains valid.

There are not many EAM implementations; we need to experimentally assess our work comparing it to the BEAM and other Prolog implementations in terms of performance, particularly when we work towards a parallel version of `WAM2EAM`.

Appendix A

Example Code Generation

As we have seen before in the body of this thesis, WAM instructions are reinterpreted under an EAM light. These original instructions can be roughly divided in three main classes:

- Execution control

- Choice point manipulation

- Unification

In this appendix, a few generated code patterns for the more crucial instructions are presented, along with the reasoning behind their implementation in terms of the EAM. We rely on the running example from Chapter 4 to illustrate the various instructions and their corresponding C code.

A.1   Execution Control

These instructions are responsible for controlling most of the execution that goes on in `WAM2EAM`, in that they induce the creation of boxes and look for where to go next when more work is needed.

A.1.1 `call`

This instruction is directly responsible for calling another predicate from within the execution of an AND-box. As calling a predicate entails jumping off to another place in the code, getting back after it executes, `call` needs to setup a return label so execution can proceed from there afterwards. This is accomplished by setting up an *AND-continuation* on the current AND-box, in effect enqueueing the address of the return label in its continuations queue. For the `call(q/1)` WAM instruction, the corresponding C snippet would then look like the following:

```
q_enqueue(a->and_conts,&&R1);
o = new_or_box(a,1);
o->args[0] = a->locals[0]; // preload A registers on the new OR-box
goto P_q_1;
```

```
R1:
a = o->parent;
o = a->parent;
```

R1 is statically setup *after* the jump to the predicate's code and, making use of `gcc`'s labels as values extension, we enqueue its address in `a->and_conts`. A new OR-box for the called predicate is then created and the only thing left to do before the jump is preloading the AND-box's current local registers on the new OR-box, so it will have its default values ready.

A.1.2 `execute`

The `execute` instruction works exactly like `call` in that it is used to call another predicate's code, yet it shows up when the called goal is the last in the current AND-box. Because of this, when execution gets back to the callee, it should look for where to proceed in the *parent* AND-box's continuations queue, instead of simply carrying on executing code as is the case with `call`.

```
q_enqueue(a->and_conts,&&R2);
```

```
o = new_or_box(a,1);

// preload A registers on the new OR-box
o->args[0] = a->locals[0];

goto P_r_1;

R2:
a = o->parent;
o = a->parent;
goto *(get_and_cont(a));
```

It should be noted that a continuation may not be found higher up in the tree
and if the configuration is stuck it may not even be anywhere to be found.
Control then goes back to the scheduler for it to apply some appropriate
rewrite rule on the configuration.

A.2   Choice Point Manipulation

Whereas in the WAM the management of alternatives was done by means
of *choice points* and *backtracking*, we have now seen that in the EAM, because
of its tree-based configuration, alternatives are fundamentally implemented
as different tree branches, rooted in an OR-box. Therefore, alternative-
management instructions, as we shall now call them, act upon the OR-
alternative field of an OR-box, which holds the address of whichever point
in code for trying the next defined clause.

A.2.1   try_me_else

This instruction sets up the initial alternative in the current OR-box.
Doing so for the instruction try_me_else(4) inside predicate r/1 is achieved
with the following snippet of C code:

```
o->alt = &&P_r_1_C4;
```

In this case, if the current clause suspends, a jump to `o->alt` is attempted, which means jumping to the `P_q_1_C4` label, where the code for the next clause is defined.

A.2.2 `retry_me_else`

`retry_me_else` simply updates the OR-alternative in the current OR-box but must also create a new AND-box in the process. The reason for its cousin `try_me_else` not having to do so is because the initial AND-box for the first clause is already created when the predicate is entered the first time, right after it gets called and control jumps to the appropriate point in the code. So compiling `retry_me_else(5)` would yield:

```
o->alt = &&P_r_1_C5;
a = new_and_box(o, 2); // arity is '2'

// bind vars to the parent OR-box's args
bind(a->locals[0], o->args[0]);
bind(a->locals[1], o->args[1]);

// define new locals' home
ASREF(a->locals[0])->home = a;
ASREF(a->locals[1])->home = a;
```

A.2.3 `trust_me_else_fail`

`trust_me_else_fail` indicates that it is dealing with the last alternative in the clause so this poses the question of what should be defined as the current OR-box alternative. As we ran out of those at this point, control should then look for what to do in the parent's AND-box continuation. This means the implementation is pretty much similar, differing in the alternative's target:

```
o->alt = (void *) get_and_cont(o->parent);
a = new_and_box(o, 2); // arity is '2'
```

```
// bind vars to the parent OR-box's args
bind(a->locals[0], o->args[0]);
bind(a->locals[1], o->args[1]);

// define new locals' home
ASREF(a->locals[0])->home = a;
ASREF(a->locals[1])->home = a;
```

A.3   Unification

By unification instructions, we mean any instruction that may induce variable binding and value management. These are any get_*, put_* or unify_* instructions. It's important to understand that a very common occurence in the EAM is having an AND-box suspend because of an attempt to bind an external variable and that only comes about through one of these unification instructions. As an example, we present the code pattern generated for the get_integer(2,0) instruction:

```
CHECK(get_integer(C_INT(2), a->locals[0], a), a->locals[0], 5);
```

The implementation of get_integer is an example of the runtime of WAM2EAM, code that is written to support the runtime execution, namely the body of these unification instructions. This one in particular looks like the following bit of code:

```
int get_integer (term_int *ti, term *var, andbox *a)
{
  var = deref(var);

  if (var->type == REF) {
    // unbound
    if (is_local(ASREF(var),a->head)) {
      term *i = ASTERM(C_INT(ti->value));
```

```
        ASREF(var)->ref = i;
        ASINT(i)->home = var;
        return BIND_OK;
    }
    return BIND_SUSP; // not local
  } else if (var->type == INT) {
    // bound
    if (X_INT(var) == ti->value)
      return CHECK_OK;
    return CHECK_FAIL;
  } else {
    return CHECK_FAIL;
  }
}
```

The outer conditional distinguishes between the case where the variable is not bound yet (a REF) and the case where it is already bound, and in this case it must be an INT otherwise the check fails immediately. In case it is not bound and it is local, it binds the variable to the given value, internalized as an integer. But if it's not local, it suspends the current AND-box. Finally, in the case the variable is already bound, it just checks its current value against the new value, failing if they don't match.

Note that the function shown above returns a meaningful macro value for any of its possible outcomes. This value is to be used by the CHECK macro wrapper in the caller, which takes appropriate measures on each of these outcomes:

```
#define CHECK(OP,VAR,R)
  do {
  retry_me##R:
    switch (OP) {
    case BIND_SUSP:
      ADD_SUSP (VAR, &&retry_me##R, a);
      goto START_SCHED;
```

```
        break;
    case CHECK_FAIL:
        remove_and_box(a);
        a = root->and_boxes[0];
        goto START_SCHED;
        break;
    case BIND_OK:
        goto START_SCHED;
        break;
    case CHECK_OK:
        goto DONE;
        break;
    }
} while (0)
```

In a nutshell, if the binding induces a suspension, the relevant info about it is saved to allow execution to get back here when conditions are right later on. If the check fails, the entire tree branch rooted on the current (failed) AND-box is pruned off the configuration and if the binding works out OK, then control resumes from wherever the scheduler tells it to. Finally, if it checks OK, then we have an answer to the query we're working on.

As the WAM2EAM is still in its initial stages of development, not every WAM instruction has a corresponding C snippet being generated, yet this appendix should be enough to give an overview of how it goes about compiling WAM instructions.

Bibliography

[1] Paulo André and Salvador Abreu. Casting the wam as an eam. In *2010 Joint Workshop on Implementation of Constraint Logic Programming Systems and Logic-based Methods in Programming Environments*, 2010.

[2] Philippe Codognet and Daniel Diaz. `wamcc`: Compiling Prolog to C. In *12th International Conference on Logic Programming*. The MIT Press, 1995.

[3] 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 *PPOPP*, pages 83–93, 1991.

[4] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The andorra-i engine: A parallel implementation of the basic andorra model. In *ICLP*, pages 825–839, 1991.

[5] Torkel Franzén, Seif Haridi, and Sverker Janson. An overview of the andorra kernel language. In Lars-Henrik Eriksson, Lars Hallnäs, and Peter Schroeder-Heister, editors, *ELP*, volume 596 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 1991.

[6] Torkel Franzén, Seif Haridi, and Sverker Janson. An Overview of the Andorra Kernel Language. In *Extensions of Logic Programming, Second International Workshop*, pages 163–179.

[7] Bruce K. Holmer, Barton Sano, Michael Carlton, Peter Van Roy, Ralph C. Haygood, William R. Bush, Alvin M. Despain, Joan M. Pendleton, and Tep Dobry. Fast Prolog with an Extended General Purpose

Architecture. In *Proceedings of the 17th International Symposium on Computer Architecture*, 1990.

[8] Sverker Janson and Seif Haridi. Programming paradigms of the andorra kernel language. In *ISLP*, pages 167–183, 1991.

[9] Ricardo Lopes and Vítor Santos Costa. The beam: A first eam implementation. In Maria Chiara Meo and Manuel Vilares Ferro, editors, *APPIA-GULP-PRODE*, pages 425–440, 1999.

[10] Ricardo Lopes, Vítor Santos Costa, and Fernando M. A. Silva. A novel implementation of the extended andorra model. In I. V. Ramakrishnan, editor, *PADL*, volume 1990 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2001.

[11] Ricardo Lopes, Vítor Santos Costa, and Fernando M. A. Silva. On the beam implementation. In Fernando Moura-Pires and Salvador Abreu, editors, *EPIA*, volume 2902 of *Lecture Notes in Computer Science*, pages 131–135. Springer, 2003.

[12] Ricardo Lopes and Vítor Santos Costa. The BEAM: Towards a first EAM Implementation. In *Proceedings of the Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, 1997.

[13] Paul Tarau. The BinProlog Experience: Implementing a High-Performance Continuation Passing Prolog Engine. Technical report, BinNet Corp., 1998.

[14] David H. D. Warren. The Extended Andorra Model with Implicit Control. ICLP90 Preconference Workshop, June 1990.