An Abstract Machine for the Execution of DeLP Programs

A. G. Stankevicius A. J. García

Laboratorio de Investigación y Desarrollo en Inteligencia Artificial
Departamento de Ciencias de la Computación
Universidad Nacional del Sur
Bahía Blanca - Buenos Aires - ARGENTINA
e-mail: {ags, ajg}@cs.uns.edu.ar

Abstract

Defeasible Logic Programming (DeLP) is a knowledge representation and reasoning formalism that by combining Logic Programming with Defeasible Argumentation is able to represent incomplete and potentially contradictory information.

Within the field of Logic Programming, most of the implementations of Prolog and its variants are based on an abstract machine defined by D. Warren (nowadays known as WAM, standing for *Warren's Abstract Machine*), that sits between the program and the actual hardware executing it. This separation of concerns allows the developer to focus mainly on the aspects related to the language being implemented, and not on the distinctive characteristics of the available hardware.

In this paper we summarize how an abstract machine can also help in the context of DeLP, exploring the points of contact between WAM and a particular abstract machine defined for this theory called JAM.

Keywords: knowledge representation and reasoning, defeasible argumentation, abstract machines.

1 Motivations

In the past few years, research in *Logic Programming* has striven to capture nonmonotonic features, seeking to develop more powerful tools for knowledge representation and reasoning. *Defeasible Logic Programming* (DeLP) is one of those formalism, that by combining Logic Programming with Defeasible Argumentation is able to represent incomplete and potentially contradictory information. Ideas borrowed from Defeasible Argumentation [8] such as representing defeasible reasons as arguments or performing a full dialectical analysis before returning the answer of queries are carefully added to a knowledge representation language featuring Prolog-like rules.

Within the field of Logic Programming, most of the implementations of Prolog and its variants are based on an abstract machine defined by D. Warren [11] (nowadays known as WAM, standing for *Warren's Abstract Machine*), that sits between the program and the actual hardware executing it. The possibility of setting aside secondary aspects of the compiler/interpreter

design has compelled many developers to adopt this approach, making it the *de-facto* standard for implementing PROLOG [1]. Moreover, the most efficient implementations seldom translate PROLOG code directly into native code, taking advantage instead of an intermediary WAM-like abstract machine.

The situation is analogous in the context of DeLP: it would be useful to have an intermediary abstract machine sitting half way between a DeLP program and the hardware executing the actual queries. The work reported in [4] defined an abstract machine suitable for this language, which due to historical reasons was called *Justification Abstract Machine*, or JAM for short. Unfortunately, this abstract machine specifically designed for an earlier revision of the theory [2, 6] lacks the recent updates made to the syntax and semantics of the system in [3, 5].

In this paper we summarize how this abstract machine can be adapted to the context of the latest revision of the system [5], exploring the points of contact between WAM and JAM, and emphasizing the changes introduced to reflect this new semantics. It is structured as follows: the next section introduces the essential concepts of DeLP. Section 3, discusses the actual JAM operation, covering what aspects of the original JAM have been updated. Finally, Section 4 presents the conclusions reached in this work.

2 Defeasible Logic Programming

This section briefly introduces DeLP's essentials following its most recent formulation [5], referring the reader looking for a comprehensive presentation to [3].

The DeLP language is defined in terms of three disjoint sets: *facts*, *strict rules*, and *defeasible rules*. Following the standard nomenclature, a literal L can either be a ground atom A or its strong negation $\sim A$. In this context, a fact is just a literal; a strict rule is an ordered pair $Head \leftarrow Body$ whose first component, $Head$, is a literal, and whose second component, $Body$, is a finite non-empty set of literals. A strict rule with head L_0 and body $\{L_1, \ldots, L_n\}$ can also be written as $L_0 \leftarrow L_1, \ldots, L_n$. In a like manner, a defeasible rule is an ordered pair $Head \leftarrow Body$ whose first component, $Head$, is a literal, and whose second component, $Body$, is a finite non-empty set of literals. In this case, a defeasible rule with head L_0 and body $\{L_1, \ldots, L_n\}$ can also be written as $L_0 \prec L_1, \ldots, L_n$. Syntactically, the symbol '$\prec$' is all that distinguishes a defeasible rule from a strict one. This distinction is important because strict rules represent undisputed information while, in contrast, defeasible rules represent tentative information, good until we find one or more reasons challenging it.

In this formalism, the state of the world we are interested in modelling is captured as a *Defeasible Logic Program* (*de.l.p*), essentially a possibly infinite set of facts, strict rules and defeasible rules. In a given *de.l.p* $\mathcal{P}$, the subset of facts and strict rules is referred as Π, and the subset of defeasible rules as Δ. When required, the *de.l.p* $\mathcal{P}$ can also be noted as (Π, Δ). Considering that the set Π is used to represent non-defeasible information, it must express certain internal coherence: it is assumed that the set Π of every *de.l.p* $\mathcal{P}$ is non-contradictory, in the sense that no two complementary literals (such as $fly(fred)$ and $\sim fly(fred)$) be derivable at the same time. As usual, the literals that can be derived are obtained chaining as many rules as required:

Definition 1. (defeasible derivation) [5]
Let $\mathcal{P} = (\Pi, \Delta)$ be a *de.l.p* and L a ground literal. A *defeasible derivation* of L from $\mathcal{P}$, noted $\mathcal{P} \vdash L$, consists of a finite sequence $L_1, L_2, \ldots, L_n = L$ of ground literals, where each L_i, $1 \leq i \leq n$, satisfy that:

1. L_i is a fact, or

2. there exists a rule in $\mathcal{P}$ (either strict or defeasible) with a head L_i and body $B_1, B_2, \ldots, B_k$ such that every literal B_j, $1 \leq j \leq k$, already appears in the sequence before L_i.

■

Note that although the set Π must be non-contradictory, the set Δ, and hence $\mathcal{P}$ itself (*i.e.*, $\Pi \cup \Delta$), may be contradictory, as shown in the following example.

Example 1. The following is a valid *de.l.p*:

$$\Pi = \left\{ \begin{array}{l} \texttt{bird(X) <- penguin(X).} \\ \texttt{\sim fly(X) <- penguin(X).} \\ \texttt{bird(tweety).} \\ \texttt{bird(fred).} \\ \texttt{sick(fred).} \\ \texttt{penguin(pengo).} \end{array} \right\} \qquad \Delta = \left\{ \begin{array}{l} \texttt{fly(X) -< bird(X).} \\ \texttt{\sim fly(X) -< weak(X).} \\ \texttt{weak(X) -< sick(X).} \end{array} \right\}$$

In the previous example, it is possible to defeasible derive both $fly(fred)$ and $\sim fly(fred)$. However, out of the literals that can be defeasible derived in a given *de.l.p*, only those able to stand the dialectical analysis are entailed. This procedure is borrowed from the field of Defeasible Argumentation, where people speak in terms *arguments* instead of derivations. An argument is a tentative piece of reasoning supporting a given conclusion, formally defined in this theory as follows:

Definition 2. (argument structure) [5]
Let $\mathcal{P} = (\Pi, \Delta)$ be a *de.l.p* and h a literal. We say that $\langle \mathcal{A}, h \rangle$ is an *argument structure* for h, if $\mathcal{A}$ is a set of defeasible rules of Δ, satisfying that:

1. there exists a defeasible derivation for h from $\Pi \cup \mathcal{A}$,

2. the set $\Pi \cup \mathcal{A}$ is non-contradictory, and

3. $\mathcal{A}$ is minimal, in the sense that there is no proper subset $\mathcal{A}'$ of $\mathcal{A}$ satisfying the previous conditions.

Also, an argument structure $\langle \mathcal{B}, q \rangle$ is a *sub-argument structure* of $\langle \mathcal{A}, h \rangle$ when $\mathcal{B} \subseteq \mathcal{A}$. ■

For instance, there exists defeasible derivations for $fly(tweety)$ and $fly(pengo)$ in the program shown in the Example 1. However, only $fly(tweety)$ is backed up by an argument, since the set of ground rules used in the derivation of $fly(pengo)$ are inconsistent with Π and therefore no valid argument structure sanctions it. Yet, in the same program is still possible to build the argument structures $\langle \mathcal{A}, fly(fred) \rangle$ and $\langle \mathcal{B}, \sim fly(fred) \rangle$, where:

$$\mathcal{A} = \{fly(fred) \prec bird(fred)\}$$
$$\mathcal{B} = \{\sim fly(fred) \prec weak(fred), weak(fred) \prec sick(fred)\}$$

In DeLP, this kind of conflict arising between argument structures is settled defining under which conditions one argument structure has enough strength to *warrant* its conclusion. In short, a given literal is warranted if we are able to find an argument structure for that literal that remains undefeated.

Definition 3. (counter-argument) [5]
We say that $\langle \mathcal{A}_1, h_1 \rangle$ *counter-argues* (also *rebuts* or *attacks*) $\langle \mathcal{A}_2, h_2 \rangle$ *at the literal* h, if and only if there exists a sub-argument structure $\langle \mathcal{A}, h \rangle$ of $\langle \mathcal{A}_2, h_2 \rangle$ such that the set $\Pi \cup \{h, h_1\}$ is contradictory. ■

Definition 4. (defeater) [5]
The argument structure $\langle \mathcal{A}_1, h_1 \rangle$ is a *defeater* for $\langle \mathcal{A}_2, h_2 \rangle$, if and only if there exists a sub-argument structure $\langle \mathcal{A}, h \rangle$ of $\langle \mathcal{A}_2, h_2 \rangle$ such that $\langle \mathcal{A}_1, h_1 \rangle$ counter-argues $\langle \mathcal{A}_2, h_2 \rangle$ at h, and either:

- $\langle \mathcal{A}_1, h_1 \rangle$ is *preferred*[1] over $\langle \mathcal{A}, h \rangle$ (proper defeat), or
- $\langle \mathcal{A}_1, h_1 \rangle$ is *unrelated* by the preference criterion to $\langle \mathcal{A}, h \rangle$ (blocking defeat)

■

Since defeaters are in turn argument structures, there may exists defeaters for the defeaters, and so on. This sequence of argument structures, each one defeating the previous one, is called *argumentation line*, in the sense that this exchange of reasons is actually exploring a given topic of the controversy.

Definition 5. (argumentation line) [5]
Let $\mathcal{P}$ be a *de.l.p* and $\langle \mathcal{A}_0, h_0 \rangle$ an argument structure from $\mathcal{P}$. An *argumentation line* for $\langle \mathcal{A}_0, h_0 \rangle$ is a sequence of argument structures from $\mathcal{P}$, noted $\lambda = [\langle \mathcal{A}_0, h_0 \rangle, \langle \mathcal{A}_1, h_1 \rangle, \langle \mathcal{A}_2, h_2 \rangle, \ldots]$, where each $\langle \mathcal{A}_i, h_i \rangle$, $i > 0$, is a defeater of its predecessor $\langle \mathcal{A}_{i-1}, h_{i-1} \rangle$. ■

As it has been shown elsewhere [7], not every exchange of counter-arguments actually constitutes a valid pattern of reasoning. For instance, circular argumentation is a particular case of *fallacious reasoning* which should be avoided. The occurrence of cycles and other undesired situations are prevented imposing the following conditions over the potential argumentation lines, where λ_S denotes the set of argument structures occupying even positions in the line λ, and λ_I the set of argument structures occupying its odd positions.

Definition 6. (acceptable argumentation line) [5]
Let $\lambda = [\langle \mathcal{A}_0, h_0 \rangle, \langle \mathcal{A}_1, h_1 \rangle, \ldots, \langle \mathcal{A}_n, h_n \rangle]$ be an argumentation line. We say that λ is an acceptable argumentation line if and only if the following conditions are met:

- λ is a finite sequence.
- The set λ_S of argument structures supporting $\langle \mathcal{A}_0, h_0 \rangle$ and the set λ_I of argument structures interfering with $\langle \mathcal{A}_0, h_0 \rangle$ are concordant.[2]
- No argument $\langle \mathcal{A}_k, h_k \rangle$ in λ is a sub-argument structure of an $\langle \mathcal{A}_i, h_i \rangle$ appearing before in λ (*i.e.*, $i < k$).
- For all i such that the argument structure $\langle \mathcal{A}_i, h_i \rangle$ is a blocking defeater for $\langle \mathcal{A}_{i-1}, h_{i-1} \rangle$, whenever $\langle \mathcal{A}_{i+1}, h_{i+1} \rangle$ exists, then $\langle \mathcal{A}_{i+1}, h_{i+1} \rangle$ must be a proper defeater of $\langle \mathcal{A}_i, h_i \rangle$.

■

[1] according to the preference criterion being applied (*e.g.*, specificity).
[2] a set $\{\langle \mathcal{B}_1, q_1 \rangle, \langle \mathcal{B}_2, q_2 \rangle, \ldots, \langle \mathcal{B}_m, q_m \rangle\}$ is said to be concordant if $\Pi \cup \mathcal{B}_1 \cup \ldots \cup \mathcal{B}_m$ is non-contradictory.

Finally, the following definition captures the essence of the procedure that decides whether an argument structure warrants its conclusion. Once again, an argument structure becomes a warrant only if it is not defeated. This prompts for a complete *dialectical analysis*.

Definition 7. (dialectical tree) [5]
Let $\langle \mathcal{A}_0, h_0 \rangle$ be an argument structure of a *de.l.p* $\mathcal{P}$. A *dialectical tree* for $\langle \mathcal{A}_0, h_0 \rangle$, noted $\mathcal{T}_{\langle \mathcal{A}_0, h_0 \rangle}$, is recursively defined as follows:

1. The root of the tree is labeled with $\langle \mathcal{A}_0, h_0 \rangle$.
2. Let N be an inner node of the tree labeled $\langle \mathcal{A}_n, h_n \rangle$, and $\lambda = [\langle \mathcal{A}_0, h_0 \rangle, \langle \mathcal{A}_1, h_1 \rangle, \ldots, \langle \mathcal{A}_n, h_n \rangle]$ be the sequence of labels in the path from the root to N. Let $\langle \mathcal{B}_1, q_1 \rangle$, $\langle \mathcal{B}_2, q_2 \rangle$, $\ldots$, $\langle \mathcal{B}_k, q_k \rangle$ be all the defeaters of $\langle \mathcal{A}_n, h_n \rangle$. For each defeater $\langle \mathcal{B}_i, q_i \rangle$, $1 \leq i \leq k$, such that the argumentation line $\lambda' = [\langle \mathcal{A}_0, h_0 \rangle, \langle \mathcal{A}_0, h_0 \rangle, \ldots, \langle \mathcal{A}_n, h_n \rangle, \langle \mathcal{B}_i, q_i \rangle]$ is acceptable, then the node N must have a child labeled $\langle \mathcal{B}_i, q_i \rangle$. If there is no defeater for $\langle \mathcal{A}_n, h_n \rangle$ or there is no $\langle \mathcal{B}_i, q_i \rangle$ such that λ' is acceptable, then N is a leaf.

■

The outcome of this tree-structured dialectical analysis (*i.e.*, whether a certain argument structure is able to warrant its conclusion), can be easily determined following a simple recursive, bottom-up labeling:

Definition 8. (marking of a dialectical tree) [5]
Let $\langle \mathcal{A}, h \rangle$ be an argument structure and $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ its dialectical tree. The corresponding marked dialectical tree, noted $\mathcal{T}^{\star}_{\langle \mathcal{A}, h \rangle}$, is obtained marking every node in $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ as follows:

1. All the leaves in $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ are marked **U** in $\mathcal{T}^{\star}_{\langle \mathcal{A}, h \rangle}$.
2. Let $\langle \mathcal{B}, q \rangle$ be an inner node of $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$. Then $\langle \mathcal{B}, q \rangle$ should be marked **U** in $\mathcal{T}^{\star}_{\langle \mathcal{A}, h \rangle}$ if and only if every child of $\langle \mathcal{B}, q \rangle$ is marked **D** in $\mathcal{T}^{\star}_{\langle \mathcal{A}, h \rangle}$. Conversely, the node $\langle \mathcal{B}, q \rangle$ should be marked **D** in $\mathcal{T}^{\star}_{\langle \mathcal{A}, h \rangle}$ if and only if it has at least one child marked **U**.

■

Definition 9. (warrant) [5]
Let $\langle \mathcal{A}, h \rangle$ be an argument structure and $\mathcal{T}^{\star}_{\langle \mathcal{A}, h \rangle}$ its associated marked dialectical tree. The literal h is *warranted* if and only if the root node of $\mathcal{T}^{\star}_{\langle \mathcal{A}, h \rangle}$ is marked **U**. ■

Although the notion of warrant characterizes a set of literals that could be interpreted as the semantic of given *de.l.p*, this formalism takes into account more possibilities as answers to a given query $\multimap h$:

- YES, if h is warranted.
- NO, if $\overline{h}$ is warranted, where $\overline{h}$ denotes the complement of h with respect to strong negation.
- UNDECIDED, if neither h nor $\overline{h}$ are warranted.
- UNKNOWN, if h is not present in the signature of the *de.l.p* under consideration.

On the one hand, the query $fly(tweety)$ is warranted in the *de.l.p* of the Example 1, therefore it succeeds and gets a YES answer. On the other hand, the query $fly(coco)$ is not present in the program signature, hence it does not succeeds and gets an UNKNOWN answer. In addition, the query $fly(fred)$ gets an UNDECIDED answer (recall the competing arguments constructed above), and the query $\sim fly(tweety)$ gets a NO answer, since its complement is actually warranted.

After this succinct introduction to DeLP, the next section reviews the abstract machine proposed for executing its programs, upon which several implementations are based.

3 An Abstract Machine suitable for DeLP

In this section we review the particular abstract machine defined for DeLP called JAM [4], showing how it can be updated to the reflect the changes introduced along the DeLP evolution. Recall that the original JAM was specifically designed for an earlier revision of the formal theory, whose syntax and semantics is slightly different with respect to the formalism briefly stated in the previous section. To begin with, the changes introduced at the syntax level should be accounted for in the compiler, thus requiring no updates on the abstract machine. For instance, although JAM is able to deal with defeasible rules with an empty body (these rules are known as *presumptions*), the compiler should inform the user that no rule can have an empty body under the new syntax.

In contrast, the changes introduced at the semantic level do require further consideration. Since JAM was conceived as an extension of WAM, we first cover the conventional WAM operation, and then address the features only available in JAM. Notably, both execution models are based on the observation that any rule can be considered as a set of queries (its body) that, when satisfied, entails a new fact (its head). Under this conception, the whole operation of WAM, and its extension JAM, can be analyzed in terms of facts and queries.

Example 2. Consider the following PROLOG program:

```
a(X) <- b(X), c(X).     b(q).
a(X) <- d(X).           c(r).
                        d(s).
```

This simple program can be translated into the WAM code shown in Figure 1, along with the translation of an arbitrary query `?- a(X).`, whose code starts in the label `?:`.

The execution model of WAM starts with the query that must be solved. The *program counter register* (P) holds the address of the instruction about to be executed. This register is incremented after its execution in order to refer the next instruction, unless the instruction just executed were a control transference. For instance, a control transference takes place when a `call` instruction gets executed: this instruction loads the address of the predicate to be called in the register P and also stores the return address in the *continuation register* (CP). The address where the code associated to each predicate begins is stored in a *permanent symbol table*.

The WAM instructions `allocate` and `deallocate` are responsible for storing and recovering *environments* from the STACK. These environments hold the local variables that should be preserved between predicate invocations. A distinctive feature in logic programming is the possibility of defining the same predicate through several independent rules. When a particular

```
a/1:  try_me_else E1:        b/1:  just_me              ?:  allocate 1
      allocate 1                   allocate 0                 put_variable X1 A1
      get_variable X1 A1           get_constant q A1          call a/1
      put_value X1 A1              deallocate                 deallocate
      call b/1                     proceed                    proceed
      put_value X1 A1        c/1:  just_me
      call c/1                     allocate 0
      deallocate                   get_constant r A1
      proceed                      deallocate
 E1:  just_me                      proceed
      allocate 0             d/1:  just_me
      get_variable X1 A1           allocate 0
      put_value X1 A1              get_constant s A1
      call d/1                     deallocate
      deallocate                   proceed
      proceed
```

Figure 1: An example of actual WAM code

definition for a predicate is chosen, the remaining alternatives are stored inside a *choice-point*. These choice-points are maintained by the WAM's special instructions implementing *backtracking*: `try_me_else`, `retry_me_else`, and `trust_me`. For instance, the query in the Example 2 only succeeds when the second definition of the predicate `a/1` is chosen.

Regarding the memory architecture, even though JAM includes additional regions not present in the original WAM, its structuring remains straightforward. The memory architecture is composed of several independent regions, each associated with a set of special registers. These regions and their corresponding registers are sketched in Figure 2. In the first place, the CODE region keeps the compiled code of the rules that made the program. The region HEAP is a stack which stores all the structures dynamically created throughout execution, mostly in the unifications. The region STACK is also a stack which stores two types of objects: the environments and the choice-points, already mentioned above. The region T-HEAP (temporary heap) is analogous to HEAP, but for storing the terms of the temporary facts (whose use within JAM is discussed further bellow in Section 3.1). The next region, LINE, stores the current argumentation line being explored. Note that the last two memory regions are not present in WAM, since they deal with aspects of the dialectical analysis not required for solving Prolog queries. Finally, the region TRAIL stores references to the variables binded during those unifications that should be unbinded when backtracking.

In addition to those registers shown in Figure 2, there are two other families of registers: *parameter registers* and *term registers*. In the one hand, the parameter registers, denoted as A_i,

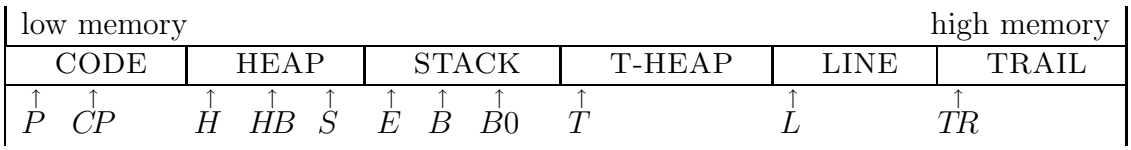

Figure 2: JAM memory architecture

are used to communicate the location of the parameters of a certain goal in the current query to the head of the clause with whom it will be unified. On the other hand, the term registers, denoted as X_i, are used in the construction of terms in the HEAP.

The HEAP is the memory region holding all the terms referred through the execution. Each cell in the HEAP contains a pair $(type, content)$, where *type* describes what is stored in *content*. The valid types are the usual ones: *reference*, *constant*, *structure*, and *list*. Since variables can either be binded to a term or unbinded, they are represented by the pair $(reference, address)$, where *address* denotes the location of the term binded to it and assuming that a self reference denotes an unbinded variable. Constants are easily represented by the pair $(constant, token)$, where *token* is the actual constant being stored. Finally, a structure of the form $f(t_1, \ldots, t_n)$ is represented by the pair $(structure, address)$, where *address* denotes the location of the pair (*constant*, f) which holds the functor name, and the following n cells contain the representation of its n subterms.

Insofar, the topics covered were common to both WAM and JAM. In what follows, we focus our attention on those aspects of JAM not present in WAM.

3.1 Finding argument structures

The construction of an argument for a query involves two main tasks: to find a defeasible derivation for that query, and then to check its consistency. The defeasible derivation can be obtained applying the same mechanism used in WAM when solving PROLOG queries (*i.e.*, the backward-chaining of rules). However, the consistency check requires some additional effort. Within JAM, the consistency of defeasible derivations is checked while the derivation is being built, making a clever use of backward-chaining. When a rule $c \prec L$ is about to be used, it is first confirmed whether $\Pi \cup \{c\}$ remains consistent. Naturally, no rule should be used in a derivation when its current bindings make it inconsistent with Π (recall the consistency requirement in argument structures imposed in Definition 2).

Let $cons(\mathcal{A})$ be the set of ground literals appearing as the consequences of the defeasible rules in an argument structure $\langle \mathcal{A}, h \rangle$. Any defeasible rule $c \prec L$ can be used in the construction of a given argument structure $\langle \mathcal{A}, h \rangle$ if and only if $\Pi \cup cons(\mathcal{A}') \cup \{c\}$ is consistent, where $\mathcal{A}'$ is the set of defeasible rules used up to this point in the construction of that argument structure (also note that $\Pi \cup cons(\mathcal{A}')$ is already consistent). These elements in $cons(\mathcal{A}')$ are what we call *temporary facts*. They are called facts because while being used they behave exactly like facts, and they are called temporary because their lifespan is restricted only to the construction of the current argument structure.

To sum up, JAM implements its consistency checks using the same machinery that finds defeasible derivations, also the very same procedure used for solving PROLOG queries within WAM. The correctness of this approach has been established in [2], from where we adapted the following lemma:

Lemma 1. Let Π be a consistent set of strict rules, and h be a literal. Then, the set $\Pi \cup \{h\}$ is inconsistent if and only if there exists a derivation of $\overline{h}$ from $\Pi \cup \{h\}$, allowing the use of some inverted rules.

The *inverted rules* are a third kind of rule that JAM uses internally for consistency checks (note that the knowledge engineer does not need to be aware of their existence). To distinguish inverted rules from the strict and defeasible ones, they are denoted as $L_0 \text{@—} L_1, L_2, \ldots, L_n$. This kind of rules play a key role within JAM, as shown in the next theorem, also adapted from [2].

Theorem 2. *Let $\mathcal{P} = (\Pi, \Delta)$ be a de.l.p. In this context, a defeasible rule $h \prec L$ can be used in the construction of an argument structure $\langle \mathcal{B}, q \rangle$ if and only if overlineh cannot be derived from $\Pi \cup cons(\mathcal{B}') \cup \{h\}$.*

Demonstration. The rule $h \prec L$ can be used in the construction of $\langle \mathcal{B}, q \rangle$ if and only if $\Pi \cup cons(\mathcal{A}') \cup \{h\}$ is consistent, and by virtue of Lemma 1, $\Pi \cup cons(\mathcal{A}') \cup \{h\}$ is consistent if and only if $\overline{h}$ can not be derived from $\Pi \cup cons(\mathcal{A}') \cup \{h\}$. □

Even though consistency checks within JAM are analogous to defeasible derivations, no inverted rules should be used in the construction of an argument structure. For this reason, two instructions, `prove_consistency_else` and `end_consistency_proof`, delimit the boundaries of the consistency check. These instructions set and clear the internal flags called `defeasibleProof` and `allowInvertedRules` that control the kind of rule that are allowed to be used in a given state. Within JAM, it is quite easy to identify the different types of rules by looking into the compiled code initially loaded in the CODE region, since different types of rules are compiled using a different sets of instructions, as shown in the following table:

Strict Rules	Defeasible Rules	Inverted Rules
`just_me`	`defeasible_just_me`	`just_me_checking_consistency`
`try_me_else`	`defeasible_try_me_else`	`try_me_else_checking_consistency`
`retry_me_else`	`defeasible_retry_me_else`	`retry_me_else_checking_consistency`
`trust_me`	`defeasible_trust_me`	`trust_me_checking_consistency`

Since temporary facts (*i.e.*, the head of every defeasible rules used up to this point) are indeed temporary, lasting only until the current argument structure is completed, they are stored in a separate structure called *temporary fact table*. In a like manner, the terms in these temporary facts are stored in a separate region (the T-HEAP), apart from the regular terms. The JAM instructions `save_srule_terms`, `save_drule_terms` and `save_fact` generate these temporary facts. Finally, as an example of the role of these concepts, Figure 3 depicts the JAM code associated to the rules $fly(X) \prec bird(X)$ and $\sim hen(X) \multimap \sim bird(X)$.

3.2 Computing answers

Since determining whether a given literal h is warranted is essential in order to answer queries about h, JAM must implement the dialectical analysis structured as a tree mentioned in Definition 7. The construction of this dialectical tree can be described in terms of the following standard PROLOG predicates:

```
warranted(Q)   :-  find_argument(A), /+ defeated(A).
defeated(A)    :-  find_defeater(A,D), /+ defeated(D).
```

Putting it into words, a literal `Q` is warranted when an argument structure for this literal can be constructed, provided that it is not defeated. In turn, an argument structure is defeated if we found a defeater for it, such that this new defeater is not defeated by another argument structure. Note that the goal `/+ defeated(A)` in the first clause is defined in terms of PROLOG's standard negation as failure. It succeeds when there are no defeaters for the argument structure `A`, and it fails when there exists at least one of those defeaters. Figure 4 shows how this procedure is implemented within JAM. First, an argument structure for the current query is constructed (following the procedures discussed in the previous section). Then, if the construction succeeds

```
fly/1:  defeasible_just_me
        allocate 2
        get_variable X2 A1
        save_drule_terms fly/1 X1
        forbid_inv
        put_value X2 A1
        call bird/1
        set_defeasible
        prove_consistency_else L:
        put_structure ~fly/1 A1
        set_value X2
        call naf/1
    L:  end_consistency_proof
        save_fact X1
        deallocate
        proceed
```

```
~hen/1:  just_me_checking_consistency
         allocate 0
         get_variable X1 A1
         put_value X1 A1
         call ~bird/1
         forbid_inv
         only_strong
         stop_only_strong
         deallocate
         proceed
```

Figure 3: JAM code for the rules `fly(X) -< bird(X)` and `~hen(X) @- ~bird(X)`

(it could be case that no argument structure support that literal), JAM looks for level one defeaters. If one of such defeaters is found, then JAM looks for level two defeaters, and so on.

In this example, the instruction `prepare_to_defeat k` dynamically generates the JAM code for the predicates `DefeatK/0`, exactly as if it were part of the original program loaded into the machine. In a sense, we are using the same JAM to execute the aforementioned piece of PROLOG code, a task it can easily achieve since the strict part of DeLP programs behaves almost like standard PROLOG. The purpose of these dynamic predicates is to find defeaters of the most recently constructed argument structure. When a defeater is found, it must be verified whether this defeater is in turn defeated, hence `DefeatK/0` is almost performing a parallel dialectical analysis. The predicate `DefeatK/0` follows the same strategy as before, trying to find an argument structure that defeats the one just constructed and then verifying that this defeater is not also defeated, this time considering defeaters in the next level, using the instruction `prepare_to_defeat K+1` and attempting the goal `/+ DefeatK+1/0`, and so on.

Throughout this process, the memory region called LINE holds the current argumentation line, containing every support and every interfering argument structure in the path from the

```
?:  allocate                      ; start of the query.
      ⋮                           ; query code.
    call q/n                      ; builds an argument.
    prepare_to_defeat 1           ; considers all its possible defeaters.
    put_constant Defeat1/0 A1     ; attempts to prove that...
    call naf/1                    ; ...no defeater can be found.
    deallocate                    ;
    end_query                     ; end of the query.
```

Figure 4: JAM's code for a query `q/n`

current node to the root of the dialectical tree. This information is used during the execution of those JAM instructions that verify whether the argumentation line under consideration is acceptable, namely `not_circular`, `not_less_specific`, `proper_defeat` and `blocking_defeat`. According to Definition 6, there are four conditions that characterize an acceptable argumentation line. The first condition is trivially met by any algorithm that returns an answer, like this one. For the second condition, the concordance check over the set of supporting and interfering argument structures present in the current argumentation line was also verified in the former theory, thus JAM already check this condition using the information stored in the temporary fact table. The same applies to the third condition, since circular argumentation, a long standing form of fallacious argumentation, was already averted in the previous formalization. Al last, the fourth condition can be asserted updating the semantic of the instructions `proper_defeat` and `blocking_defeat`. In the original JAM, blocking defeaters were only accepted as a defeater of the argument structure labeling the root of the dialectical tree, but in the new formulation they are allowed anywhere but as response of another blocking defeater.

It should be noted that the dialectical tree being constructed never gets represented as a complete tree, only the argument structures present in the current argumentation line are kept. However, when an argumentation line is exhausted, some of its argument structures might not be part of the new argumentation line about to be explored. This is implemented within JAM accessing LINE like as a stack, storing the register that holds the current top of this stack (called L) inside the choice-points, and restoring its value whenever backtracking.

Finally, being able to determine whether a given literal is warranted allows JAM to compute the right answer. For instance, given a literal h, just attempt to warrant both h and $\overline{h}$, and then return the answer shown in the following table:

h	$\overline{h}$	*right answer*
warranted	not warranted	YES
not warranted	warranted	NO
not warranted	not warranted	UNDECIDED
warranted	warranted	*(impossible)*

4 Conclusions

Defeasible Logic Programming provides an attractive framework for knowledge representation and reasoning, mainly due to its ability to represent incomplete and potentially contradictory information. Having an adequate implementation of this theory leads not only towards developing robust applications but also allowing the fine tuning of the formalism itself. To that end, in this papers we have shown how can an approach proven successful within Logic Programming, namely having an abstract machine sitting half way between the logic program and the hardware actually executing it, could also be adapted to context of DeLP.

An abstract machine suitable for DeLP was already introduced in a previous work, although it was specifically designed for an earlier revision of the theory, that lacks the recent updates to its syntax and semantics. The purpose of this paper was to highlight the main features of the internal operation of this abstract machine, clearly stating as well what must be changed in order to make it suitable for the newest revision of DeLP. In fact, this updated abstract machine actually underpins the DeLP implementation reported in [9, 10], which is available for download at `http://cs.uns.edu.ar/~ags/DLP`.

Acknowledgements

This research was partially supported by *Secretaría General de Ciencia y Tecnología de la Universidad Nacional del Sur* (24/N016) and by *Agencia Nacional de Promoción Científica y Tecnológica* (PICT 2002 No. 13096). The first author is also partially supported by a fellowship from *Comisión de Investigaciones Científicas* (CIC).

References

[1] Aït-Kaci, H. *Warren's Abstract Machine, a tutorial reconstruction*. MIT Press, 1991.

[2] García, A. J. La Programación en Lógica Rebatible: su definición teórica y computacional. Master's thesis, Departamento de Ciencias de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina, June 1997.

[3] García, A. J. *Programación en Lógica Rebatible: Lenguaje, Semántica Operacional, y Paralelismo.* PhD thesis, Departamento de Ciencias de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina, Dec. 2000.

[4] García, A. J., and Simari, G. R. Una extensión de la máquina abstracta de Warren para la argumentación rebatible. In *Proceedings of the III Congreso Argentino de Ciencias de la Computación* (La Plata, Oct. 1997), Universidad Nacional de La Plata.

[5] García, A. J., and Simari, G. R. Defeasible Logic Programming: An Argumentative Approach. *Journal of Theory and Practice of Logic Programming 4*, 1–2 (2004), 95–138.

[6] García, A. J., Simari, G. R., and Chesñevar, C. I. An Argumentative Framework for Reasoning with Inconsistent and Incomplete Information. In *Proceedings of the Workshop on Practical Reasoning and Rationality* (Brighton, Reino Unido, Aug. 1998), 13th European Conference on Artificial Intelligence, pp. 13–19.

[7] Simari, G. R., Chesñevar, C. I., and García, A. J. The Role of Dialectics in Defeasible Argumentation. In *Proceedings of the XIV Conferencia Internacional de la Sociedad Chilena para Ciencias de la Computación* (Concepción, Chile, Nov. 1994), Universidad de Concepción, pp. 111–121.

[8] Simari, G. R., and Loui, R. P. A Mathematical Treatment of Defeasible Reasoning and its Implementation. *Artificial Intelligence 53*, 1–2 (1992), 125–157.

[9] Stankevicius, A. G., García, A. J., and Simari, G. R. Una arquitectura para la ejecución de Programas Lógicos Rebatibles. In *Proceedings of the 5th International Congress on Informatics Engineering* (Capital Federal, Argentina, Aug. 1999), Universidad de Buenos Aires, pp. 450–461.

[10] Stankevicius, A. G., García, A. J., and Simari, G. R. Compilation Techniques for Defeasible Logic Programs. In *Proceedings of the 6th International Congress on Informatics Engineering* (Capital Federal, Argentina, Apr. 2000), Universidad de Buenos Aires.

[11] Warren, D. H. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, CA, Oct. 1983.