Compilation Techniques for Defeasible Logic Programs

A. G. Stankevicius* A. J. García† G. R. Simari

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

Abstract

Defeasible logic programming is a newly developed extension to conventional logic programming that captures some aspects of common-sense reasoning hard to model within the classical approach. Particularly,*defeasible logic programs* (**DLP**) are expressive enough to represent incomplete and potentially contradictory information, applying ideas from *defeasible argumentation* in order to decide between conflicting goals.

Nowadays, several architectures designed for executing **DLP**s are based on abstract machines; the compiler plays a key role in these architectures. Structural simplicity and good maintainability are valuable properties in compilers for recently developed languages like defeasible logic programming.

Predictive recursive compilers grant these desired properties. However, the translation of **DLP**s into instructions of the abstract machine is rather difficult in this setting, particularly when parsing terms and lists. In this paper, we propose some compilation techniques that deal with these problems, field-tested in a **DLP** compiler entirely written in **Java**.

Keywords: `defeasible logic programming, logic program compilation.`

1 Introduction

Whenever we attempt to use logic programming as a modeling tool for common-sense reasoning, we will realize, eventually, that the classical approach lacks some essential features required for the enterprise [5, 10]. This prompts the development of extensions to conventional logic programming that successfully address these inconveniences. *Defeasible logic programming* [4] is one of these newly developed extensions; it captures aspects of common-sense reasoning hard to model within the classical approach. Particularly,*defeasible logic programs* (**DLP**s) are expressive enough to represent incomplete

*Supported by a fellowship of Secretaría General de Ciencia y Tecnología, UNS.
†Partially supported by Fundación OSDIC.

and potentially contradictory information, applying ideas from *defeasible argumentation* in order to decide between conflicting goals [12].

Nowadays, several architectures designed for executing **DLP**s are based on abstract machines [4, 7, 11, 13] (like the majority of the systems for executing **Prolog** programs). As emphasized in [11], the compiler plays a key role in these architectures. Note that when implementing compilers for recently developed languages (*e.g.*, defeasible logic programming), developers seek structural simplicity and good maintainability, given that new ideas need be prototyped and tested constantly. For example, almost every **Prolog** compiler that exploits parallelism, are entirely written in **Prolog** [3].

Predictive recursive compilers [2] grant these desired properties. Unfortunately, the translation of **DLP** terms and lists into instructions of the abstract machine is rather difficult for a top-down compiler. Other compilers (*e.g.*, bottom-up) do not suffer this drawback, but they tend to be complex in structure and not so easy to maintain. The reader might notice that predictive recursive compilers cannot deal with dynamic operators (a powerful resource available in **Prolog**). Not withstanding, defeasible logic programming *is not* **Prolog**, and does not have dynamic operators, thus predictive recursive compilers are still applicable.

Naturally, writing the compiler entirely in **Prolog** gives us excellent modularity and maintainability. Moreover, with the expressive power of **Prolog** it is easy to deal successfully with the problems arising in term and list compilation. Despite of this, a compiler of this kind would be severely constrained in the size of the input allowed, and also in the efficiency of the translation process. Following [9], we decided to take a step towards an efficient translation without neglecting the other properties. In this trade-off, a good balance of efficiency versus maintainability can be achieved adopting a predictive recursive compiler.

The classical approach for compiling **Prolog** terms resorts to find one of its *flattened forms* [1]. However, this method cannot be directly applied to predictive recursive compilers. In this paper, we propose some compilation techniques for dealing with the problems that arise in the translation of **DLP**s in that type of compiler. These techniques were field-tested in a **DLP** compiler, entirely written in **Java**, that comprises an integral part of the visualization system reported in [13].

This paper is organized as follows: in the next section, we discuss the essentials of defeasible logic programming from point of view of the compiler. Section 3 overviews the compiler where we developed the compilation techniques. In section 4, we begin by reviewing the standard approach to term compilation, and we also define the new techniques. We conclude section 4 addressing the compilation of lists, a special type of term. Finally, section 5 summarizes our conclusions.

2 Defeasible Logic Programming

In what follows, we discuss the essentials of defeasible logic programming from point of view of the compiler. For a more comprehensive analysis we refer the interested reader to its original definition [4].

In the language of defeasible logic programming, a *term* is either a *variable* (denoted by an uppercase initial), a *constant* (denoted by a lowercase initial), or a *structure* $f(t_1, \ldots, t_n)$, where f is a *functor* (with the same restrictions as for constants) and the t_i are terms (also known as *subterms*). For each functor, the number of subterms is fixed and is called *arity*. Considering that the same functor could be associated with several arities we adopt the standard convention of denoting a functor f of arity n as f/n.

An *atomic predicate* is an expression of the form $p(t_1, \ldots, t_n)$, $n \geq 0$, where p is a predicate name (with the same restrictions as for constants) and the t_i are terms (also known as predicate *parameters*). Once again, we call the amount of parameters in predicate p its *arity*. We adopt the same convention as before when denoting predicate arity. A *literal* l is either an atomic predicate p, or its negation $\sim p$. Note that the symbol "$\sim$" denotes strong negation, not to be confused with the usual negation in conventional logic programming (negation as failure), denoted in this language as "`not`".

Finally, a *defeasible logic program* (**DLP**) is a finite set of *strict rules* and *defeasible rules*. A strict rule is a rule of the form "$l \leftarrow p_1, \ldots, p_n$", $n \geq 0$, where l is a literal, and each p_i is either a literal or the symbol "`not`" of negation as failure followed by a literal. If $n = 0$, we say that l is a *fact*, denoted "$l \leftarrow$ true". Likewise, a defeasible rule is a rule of the form "$l \prec p_1, \ldots, p_n$", $n \geq 0$, with the same considerations for l and the p_i as before. In this case, if $n = 0$ we say that l is a *presumption*, denoted "$l \prec$ true". The symbol "$\prec$" distinguishes defeasible from strict rules. This distinction is important because strict rules represent undisputed information, while in contrast, defeasible rules represent tentative information that can be used until its validity is questioned. For this reason, every **DLP** contains two disjoint set of rules: one that models strict knowledge, and the other that models defeasible knowledge.

For every rule in a **DLP**, the consequent l is called *head*, and the antecedent $p_1, \ldots, p_n$ is called *body*. Following this terminology, the symbols "$\leftarrow$" and "$\prec$" are called *neck*. Finally, a rule without head is called query.

Example 2.1. A **DLP** concerning the popular `tweety`.

```
fly(X) -< bird(X).          penguin(pengo) <- true.
~fly(X) -< ill(X).          bird(tweety) <- true.
~fly(X) <- penguin(X).      bird(fred) -< true.
bird(X) <- penguin(X).      ill(fred) -< true.
```

Recall that we are focusing our attention on the point of view of the compiler. Therefore, we do not address the "higher level" notions of *argument*, *defeat* and *justification* that have been explored elsewhere [8].

3 Compiler overview

In this section we overview the compiler used as test-bed for demonstrating the proposed compilation techniques. This compiler is one of the key components of the system reported in [13]. It takes a **DLP** and translates it into instructions of an abstract machine that was specifically developed for defeasible logic programming. The abstract machine, called *Justification Abstract Machine* (**JAM**) [4, 7], is an extension of another well-known abstract machine defined for **Prolog** execution, devised by D. Warren and nowadays known as *Warren's Abstract Machine* (**WAM**) [14, 1].

In what follows, we detail the compilation phases, giving a brief introduction to the abstract machine used as target of the compilation process.

```
       program  →  clause program | EOF
        clause  →  head neck body . | head .
          head  →  literal
       literal  →  atom | ~atom
          atom  →  functor parameter-list
       functor  →  lower-id
parameter-list  →  ( term-list ) | ε
     term-list  →  term , term-list | term
          term  →  number | atom | list
      variable  →  upper-id | anon-id
          list  →  variable | [ term-list | list ] | [ term-list ] | []
          neck  →  :- | <- | -<
          body  →  query-list
    query-list  →  query , query-list | query
         query  →  !  | literal | not literal
```

Figure 1: Preprocessor initial grammar.

3.1 Compilation phases

The compiler is grounded on a well established type of top-down parser, namely predictive recursive parsers [2]. Note that there are several tough tasks that should be performed by any **DLP** compiler besides the translation itself. Among others, the generation of *inverted rules* [6, 4] is demanding. For this reason, we decided to split the compiler work in two compilation phases: the preprocessing phase, and the translation phase. Initially, the **DLP** to be compiled is fed into the preprocessor. In turn, its output is fed to the translator obtaining as a result the original program expressed in terms of **JAM** instructions. Given that every rule is independent from each other by definition, the task of translating **DLP**s can be reduced to the task of translating a single rule.

Every predictive recursive parser is built on top of a predictive recursive grammar. Figure 1 shows the initial grammar used as the source language (*i.e.*, the language accepted by the preprocessor). Recall that the final grammar should be factorized, and left recursion should be eliminated before trying to apply this parser-construction method. The resulting grammar is portrayed in Figure 2.

It may seem strange to consider variables —denoted in Figure 2 by the terminal symbols `upper-id` and `anon-id`— as lists instead of as terms. We have chosen this grammar because it solves the dilemma faced by the compiler when dealing with alternative representations for lists. For example, both `[a]` and `[a|[]]` denote the *same list*, namely the one containing only the constant `a`. We shed more light on this matter in section 4.2.

Note that the language accepted by the parser of the translator differs from the language of defeasible logic programming. For instance, the preprocessor generates inverted rules that are not allowed as rules of any **DLP**. The translator's grammar is discussed in section 4.1.

3.2 Justification abstract machine (JAM)

As stated before, the compiler translates **DLP**s into **JAM** instructions. In order to gain a deeper understanding of the translation process, we introduce in what follows the basic concepts of the abstract machine.

In a **DLP**, every rule can be considered as a set of queries (its body) that when

```
       program  →  clause program | EOF
        clause  →  literal clause'
       clause'  →  neck query-list . | .
       literal  →  atom | ~atom
          atom  →  lower-id parameter-list
parameter-list  →  ( term-list ) | ε
     term-list  →  term term-list'
    term-list'  →  , term-list | ε
          term  →  number | atom | list
          list  →  upper-id | anon-id | [ term list' ] | []
         list'  →  , term list' | | list | ε
          neck  →  :- | <- | -<
    query-list  →  query query-list'
   query-list'  →  , query-list | ε
         query  →  !  | literal | not literal
```

Figure 2: Preprocessor predictive recursive grammar

satisfied entails a new fact (its head). This particular view allows the whole treatment of the abstract machine in terms of facts and queries. The execution of **JAM** code always starts with a query. Thereafter, some rule such that its head matches with a goal in the current query is selected. Next then, that goal is removed from the current query (it has been met), and the body of the selected clause is added to the current query. Finally, the process starts all over again, this time with the new query.

JAM's memory architecture is simple: it contains several separate regions, and a set of special registers associated with each of them (see Figure 3). The **CODE** region keeps the code of the compiled rules of the program. The **HEAP** is a stack that stores all the structures created throughout execution (particularly in the unification). The **STACK** is also a stack that stores two types of objects: *environments* (where the information local to each rule is stored), and *choice-points* (where backtracking information is stored). The **T-HEAP** (temporary heap) is analogous to the **HEAP**, but for keeping the so-called *temporary facts* [7]. The next region, the **LINE**, stores the current argumentation line. These last two regions are not present in the **WAM**, as they deal with certain aspects of the argumentation process not present in **Prolog**. Finally, the **TRAIL** stores references to variables binded during unification that should be unbinded if a backtrack is performed.

The abstract machine also has two types of registers: *parameter registers*, and *term registers*. The parameter registers are used to inform the head of the rule selected for unification about the location of the parameters belonging to a certain goal present in the current query (*i.e*, implementing the parameter passing). The term registers are used in the construction of the term representation inside the **HEAP**. Our particular usage of term register is explained in the next section.

low memory					high memory
CODE	**HEAP**	**STACK**	**T-HEAP**	**LINE**	**TRAIL**
↑ P ↑ CP	↑ H ↑ HB ↑ S ↑ E	↑ B ↑ $B0$ ↑ T	↑ L	↑ TR	

Figure 3: Abstract machine's memory architecture.

4 Compilation techniques

In this section we define the proposed compilation techniques. We start by reviewing term compilation, and in the sequel the compilation of a special type of term, the lists.

4.1 Term compilation

The terms of each rule can be compiled independently from the rest. For this compiler, we have elaborated from the intuitions behind the compiler reported in [6], namely, we use a temporary symbol table for storing the partial translation of the rule being compiled.

Each entry in the temporary symbol table contains the following fields[1]:

1. **Goal Position:** keeps the position of the current goal in the rule. It is used to distinguish permanent from temporary variables.
2. **Identifier:** stores the name associated with the current entry.
3. **Arity:** only meaningful when the current identifier is a predicate or a structure.
4. **Type:** describes the kind of identifier stored (*e.g.*, variable, constant, etc.)
5. **Opcode: JAM** instruction associated with this entry.
6. **Opcode Parameters:** only meaningful when the stored opcode has parameters.
7. **Parameter Register Reservoir:** holds the parameter register assigned to the opcode.
8. **Term Register Reservoir:** holds the term register assigned to the opcode.

As stated before, the memory region **HEAP** holds every term that appears in a **DLP**. Each of the **HEAP** cells contains a pair $(type, content)$, where *type* describes what is stored in *content*. The valid types are: *reference*, *constant*, *structure*, and *list*. Let us analyze in detail the representation of each type of term.

A variable can be either binded to a term, or unbinded. Hence, a variable is represented by the pair $(reference, address)$, where $address$ is in turn the pair $(region, offset)$ that denotes the location of the term binded to it. We assume the convention that self reference denotes an unbinded variable. Constants are easily represented by the pair $(constant, token)$, where $token$ is the actual constant. Finally, a structure of the form $f(t_1, \ldots, t_n)$ is represented by the pair $(structure, address)$, where $address$ is in turn the pair $(region, offset)$. This pair denotes the location of the pair $(constant, f/n)$ and the following n cells are the representation of its n subterms.

Example 4.1. Assuming the offset n as a starting address in the **HEAP**, we can represent the term $f(s(X), X)$ —actually located at $n+4$— in the following way:

n	reference	$(\mathbf{HEAP}, n)$	X
$n+1$	structure	$(\mathbf{HEAP}, n+2)$	$s($
$n+2$	constant	$s/1$	$\vdots$
$n+3$	reference	$(\mathbf{HEAP}, n)$	$X)$
$n+4$	structure	$(\mathbf{HEAP}, n+5)$	$f($
$n+5$	constant	$f/2$	$\vdots$
$n+6$	reference	$(\mathbf{HEAP}, n)$	$X,$
$n+7$	structure	$(\mathbf{HEAP}, n+2)$	$s(X))$

[1] a deeper analysis can be found in [4, 13].

location	1^{st} occur.	in struct.	**WAM** & **JAM** opcodes
body	yes	yes	`put_variable Vn Ai`
body	yes	no	`set_variable Vn`
body	no	yes	`put_value Vn Ai`
body	no	no	`set_value Vn`
head	yes	yes	`get_variable Vn Ai`
head	yes	no	`unify_variable Vn`
head	no	yes	`get_value Vn Ai`
head	no	no	`unify_value Vn`

Table 1: **WAM** and **JAM** instructions for variables.

The **JAM**, as a **WAM** extension, has a larger set of instructions. However, the instructions for handling terms coincide in both abstract machines. Clearly, a term can occur either in the head, or in the body of a rule; different **WAM** (**JAM**) instructions are used in each case. Moreover, there are different instruction for each type of term. In what follows, we describe how the compiler generates the proper instructions for constants, variables, and structures.

There are several factors that determine which is the right instruction to output when translating a variable: whether it is the first occurrence, whether it appears in the head or in the body of a rule, etc. Table 1 summarizes the different alternatives. The compilation of constants is easier, only four instructions are involved; the right one is chosen according to Table 2.

When it comes to compiling structures, the hidden complexity of this task arises. We will follow the approach of H. At-Kaci [1]. First, let us consider the compilation of terms in the rule body, and next the compilation of terms in the rule head.

The code in the rule body is responsible for constructing the representation of its terms in the **HEAP**, in such a way that the code of any selected rule head would find those terms, in order to perform the expected unification. As discussed before, a structure f/n is represented with at least $n+2$ cells (note that some subterms may also be structures). In Example 4.1, the subterms are built in the **HEAP** before the term f/n itself. In other words, the representation of every subterm must already be in the **HEAP** *before* considering the term.

Let us describe how term registers are used during the construction of the representation of terms. Each component of the term being compiled is associated with one term register. The assignments, for example, in the term $f(X, g(X, Y), Y)$ are $X_1 = f(X_2, X_3, X_4)$, $X_2 = X$, $X_3 = g(X_2, X_4)$, and $X_4 = Y$. Accordingly, any term can be understood as a sequence of equations of the form $X_i = id$, for constants and

location	in struct.	**WAM** & **JAM** opcodes
body	yes	`put_constant c Ai`
body	no	`set_constant c`
head	yes	`get_constant c Ai`
head	no	`unify_constant c`

Table 2: **WAM** and **JAM** instructions for constants.

variables, or $X_i = f(X_{i_1}, \ldots, X_{i_n})$ for structures. This is usually called *flattened form of terms* [1].

The flattened form of terms is closer to what we need, but not every term in its flattened form is suitable. We only accept sequences of equations such that no term register is referred prior to its definition (*i.e.*, every term register appears on the right side of an equation if and only if it has appeared on the left side of a previous equation). This is what we call **flattened form of a term in the rule body** (FFB for short). For the previous term, its FFB is $X_2 = X$, $X_4 = Y$, $X_3 = g(X_2, X_4)$, and $X_1 = f(X_2, X_3, X_4)$. Since every rule in a defeasible logic program is universally closured, the particular name of each variable is irrelevant. Thus, we can denote each variable by its associated term register. In the last example, X_2 denotes X, and X_4 denotes Y. The resulting sequence of equations (ignoring particular names) is $X_3 = g(X_2, X_4)$, and $X_1 = f(X_2, X_3, X_4)$.

Having obtained this sequence of equations, the desired set of instructions that builds the term representation in the **HEAP** is exactly the set of instructions obtained from the translation of each subterm found in a simple left to right scan of the FFB. The general idea for compiling a term is that each subterm is associated with a new entry in the temporary symbol table, and thereafter a linear scan of the table generates the required translation. There still is a pitfall to be avoided: the flattened form obtained in this way may not be valid.

It is easy to see why we cannot blindly apply this method in the context of a predictive recursive compiler: the flattened form obtained in the compiler's parse through the term need not be acceptable. For example, consider the term $f(g(X))$; its corresponding sequence of equations (once again, ignoring particular names) is $X_1 = f(X_2)$, and $X_2 = g(X_3)$. This sequence is not an FFB: X_2 is referred prior to its definition. Other compilers, such as those entirely written in **Prolog**, solve this problem considering the whole term before constructing its flattened form (*i.e.*, considering terms as terminal symbols in their grammar). Our solution to this inconvenience is natural: rearrange the entries in the temporary symbol table in such a way that the resulting flattened form becomes acceptable (*i.e.*, the flattened form is always an FFB).

Figures 1 and 2 depicted the grammar accepted by the preprocessor. In order to describe the term compilation technique, we need to introduce the translator's grammar (compiler's second phase). Figure 4 shows the initial grammar. Recall that we need to factorize, and to eliminate the left recursion from the initial grammar. The grammar obtained is portrayed in Figure 5.

In the first place, we address the compilation of terms that are subterms in another term (*i.e.*, they appear inside some structure). In the grammar, it is clear that any subterm can inheritate the position in the table of the outer functor, as well as its current arity (*i.e.*, the position, inside the term, of the subterm being considered). We denote with *parent* the position in the table of the outer functor, *current_arity* the current arity, and *arity* the actual arity. The term translation will be stored between the table positions *parent* and $parent + arity$.

In this setting, the translation of a constant is simple. The compiler initially places that translation at the end of table. In order to put it in the right position, we need to move the last entry to the position $parent + current\_arity$. The translation of variables can be done in a like manner: the compiler initially places the translation of the variable according to Table 1 at the end of the table. Then, this last entry is moved to its appropriate position ($parent + current\_arity$). Recall that different instructions are generated depending on whether it is the first occurrence of a variable or not. Considering this, we

```
          program  →  clause program | EOF
           clause  →  head neck body consistency-check . | head .
             head  →  literal
          literal  →  atom | ~atom
             atom  →  functor parameter-list
          functor  →  lower-id
   parameter-list  →  ( term-list ) | ε
        term-list  →  term , term-list | term
             term  →  number | atom | list
         variable  →  upper-id | anon-id
             list  →  variable | [ term | list ] | []
             neck  →  :- | <- | -< | @-
             body  →  query-list
       query-list  →  query , query-list | query
            query  →  !  | literal | not literal
consistency-check  →  # literal | ε
```

Figure 4: Translator's initial grammar.

have to be careful when moving that entry: an occurrence that is not the first may become the first after the rearrangement (rendering invalid the translation of both occurrences). With the help of a little trick, we avoid altogether the problematic situation imposing that the first appearance of any variable cannot occur inside a structure. We achieve this by creating a ghost occurrence of any variable such that its first appearance occurs inside a structure just before the term in which it is contained. The ghost occurrence should be placed in the position $parent - 1$ with the opcode `set_variable Xi` (*i.e.*, the opcode for a variable in the rule body appearing for the first time). After this, the occurrence inside the structure is compiled as always (it is no longer the first occurrence).

The compilation of a structure inside another structure is more subtle. Note that the translation of the inner structure may span over more than one entry, and therefore the outer structure will span over more than what we expected. We avoid this problem recoursing once again to term registers. For example, the term $f(g(X))$ need be understood

```
          program  →  clause program | EOF
           clause  →  literal clause'
          clause'  →  neck query-list consistency-check . | .
          literal  →  atom | ~atom
             atom  →  lower-id parameter-list
   parameter-list  →  ( term-list ) | ε
        term-list  →  term term-list'
       term-list'  →  , term-list | ε
             term  →  number | atom | list
             list  →  upper-id | anon-id | [ term | list ] | []
             neck  →  :- | <- | -< | @-
       query-list  →  query query-list'
      query-list'  →  , query-list | ε
            query  →  !  | literal | not literal
consistency-check  →  # literal | ε
```

Figure 5: Translator's predictive recursive grammar.

as $X_i = g(X)$ plus $f(X_i)$. This can be easily accomplished by compiling any structure inside another structure *ignoring* the outer structure. The temporary symbol table will end up with a partially compiled structure in the middle (the outer structure), and a completely compiled structure in the bottom (the inner structure). At this point, we complete the compilation of the outer structure by adding a reference to the completely compiled structure. This seems somewhat messy, but in fact it is quite straightforward: just place the reference to the completely compiled structure (*i.e.*, the term register holding that structure) in the position $parent + current\_arity$.

At the end of the term translation, some housekeeping is required: after compiling every subterm of a structure, the translation of its subterms are located below the translation of the term itself (violating the FFB). Moving the range $(parent, parent + arity)$ to the bottom of the temporary symbol table restores the acceptability of the flattened form.

Briefly stated, the whole process of compiling a subterm occurring inside a structure encompasses the following tasks:

- If the subterm is a constant, its translation should be placed in the position $parent + current\_arity$ of the temporal symbol table.
- If the subterm is a variable, its translation should be placed in the position $parent + current\_arity$ of the temporal symbol table, taking care of the problematic situation where this occurrence is the first and appears inside another structure.
- If the subterm is a structure, its translation should be placed at the bottom of the temporal symbol table. Thereafter, a reference to that translation should be placed —before the housekeeping— in the position $parent + current\_arity$.

We have discussed how to compile terms inside a structure. The compilation of terms that appears as predicate parameters (*i.e.*, they are not subterms of any term) is remarkably easy: constants and variables are simply compiled according to their tables, and for any structure, we consider the predicate in which it is contained as an outer structure, and simply follow the same technique as before (the similarity is prominent, see Figure 5).

The compilation of terms in the rule head resembles the previous case. Note, however, that some query code is always executed before the rule head code. Considering this, the execution of the query code may leave in the **HEAP** either a term, or an unbinded variable. For this reason, the code associated with the rule head should admit two working modes: a *read mode* that is used to unify the rule head with the terms in the **HEAP**, and a *write mode* that is used to construct the term that the unbinded variable will refer to. In any case, the code associated with the rule head has to perform the expected unification.

The code associated with a term in the rule body needs to construct its representation in the **HEAP**. On the contrary, the code associated with a term in the rule head does not. (In a certain way, a tentative representation of that term is already present in the **HEAP**.) We define the **flattened form of term in the rule head** (FFH for short) in a way analogous to the FFB. A flattened form of a term is an FFH if and only if its corresponding sequence of equations is such that no term register is referred after its definition (*i.e.*, a term register appears on the left side of an equation if and only if that term register does not appears on the right side of any previous equation). For the term

$f(X, g(X, Y), Y)$, its FFH is $X_1 = f(X_2, X_3, X_4)$, $X_3 = g(X_2, X_4)$, $X_2 = X$, and $X_4 = Y$. Applying the same ideas, from this particular kind of flattened form we can obtain the corresponding translation.

4.2 List compilation

The *list* is a special kind of term. We have reviewed how to compile a general term, therefore, in some way, we already know how to compile lists. The main difficulty is that, unlike the regular terms, the same *semantic object* can be represented by several *syntactic objects*.

Consider the following rule that captures the usual syntax for lists:

```
list → [ term-list | list ] | [ term-list ] | []
```

Even though defeasible logic programming is type free, the compiler performs certain type checking on lists. For instance, according to the previous grammar, any sound compiler should reject the term `[a|a]` on the basis that the object appearing after the vertical slash is neither a list, nor the constant `[]`.

Given that the compiler already has a preprocessor, we can get rid of these ambiguities in the preprocessing phase, and only deal with the *canonical form* of lists throughout the translation phase. We say that a list is in its canonical form if it is described by the following rule:

```
list → [ term | list ] | []
```

For example, the canonical form for `[1,2,3]` is `[1|[2|[3|[]]]]`. Let us discuss our approach for converting any list to its canonical form. Recall the following rules from the grammar in Figure 2:

```
 list   →  upper-id | anon-id | [ term list' ] | []
list'   →  , term list' | | list | ε
```

This grammar is not the usual way of expressing general lists. Despite of this, the language accepted by this grammar is the same language accepted by the previous (and more familiar) grammar. Assume that `emit()` is a procedure that outputs its parameter to the temporary storage, later used as input for the translator. The conversion is made according to the following syntax-directed definition:

Syntactic Rules	**Semantic Rules**
`list → upper-id`	`emit(upper-id.lexeme)`
`list → anon-id`	`emit('_')`
`list → [ term list' ]`	`emit('[');parse(term);emit('|');` `parse(list');emit(']')`
`list → []`	`emit('[]')`
`list' → , term list'`	`emit('[');parse(term);emit('|');` `parse(list');emit(']')`
`list' → | list`	`parse(list)`
`list' → ε`	`emit('[]')`

Those alternative representations will not reach the translator; it will deal only with canonical lists. If we understand brackets as parenthesis and vertical slashes as commas, canonical lists have the form of a structure with an empty-name functor. In the literature, the implicit functor is usually denoted with a dot. For instance, the canonical list `[1|[2|[3|[]]]]` can be understood as the term `.(1,.(2,.(3,[])))`. Under this interpretation, we know how to compile lists, given that the name of the implicit functor is always the same (to allow the correct unifications among lists), and that the implicit functor is not a valid functor (to avoid the incorrect unification among lists and structures).

Nevertheless, it is possible to compile lists in a more efficient way. There are special instructions defined for lists in **JAM** that avoid wasting **HEAP** space storing that (useless) implicit functor. The lists compilation is done following the procedure described for general terms, but using list instructions instead of term instructions. A list is represented in the **HEAP** by the cell $(list, address)$, where $address$ is the pair $(region, offset)$ that denotes the location of the two cells that contain the translation of the subterms of the list. Contrast this representation against the representation defined for structures. Note that this representation is compact and no cell is wasted with the empty-name functor.

Example 4.2. Assuming the offset n as a starting address in the **HEAP**, we can represent the list `[a|[b|[c|[]]]]` —actually located at $n+4$— in the following way:

n	constant	$c/0$
$n+1$	constant	`[]`/0
$n+2$	constant	$b/0$
$n+3$	list	(**HEAP**, n)
$n+4$	constant	$a/0$
$n+5$	list	(**HEAP**, $n+2$)

5 Conclusions

Defeasible logic programming has reached the stage of development where compiler efficiency begins to weight. However, maintainability is also a desirable property for **DLP** compilers. An interesting trade-off between these factors is achieved with predictive recursive compilers, but, in these compilers, term compilation becomes challenging. In this paper, we have proposed a technique for compiling terms that solves the difficulties encountered in this setting.

We have also defined a simple way of dealing with the ambiguities in list representation, applicable to any compiler with some sort of preprocessing phase. These techniques could be adapted to almost any recently develop logic programming extension (as long as it does not involve dynamic operators).

References

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

[2] **Aho, A., Sethi, R., and Ullman, J.** *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[3] **Carlsson, M.** A Prolog Compiler and its Extension for OR-Parallelism. Technical Report R90-06, Swedish Institute of Computer Science, 1990.

[4] **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.

[5] **Gelfond, M., and Lifschitz, V.** Logic Programs with Classical Negation. In *Proceedings of the 7th International Conference on Logic Programming* (1990), D. H. D. Warren and P. Szeredi, Eds., pp. 579–597.

[6] **García, A. J., and Simari, G. R.** Un compilador para la programación en lógica rebatible. In *Proceedings of the III Congreso Argentino de Ciencias de la Computación* (La Plata, Argentina, October 1997), Universidad Nacional de La Plata, pp. 1279–1297.

[7] **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, Argentina, October 1997), Universidad Nacional de La Plata, pp. 1201–1220.

[8] **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, United Kingdom, August 1998), 13th European Conference on Artificial Intelligence, pp. 13–19.

[9] **Meier, M., and Schimpf, J.** An architecture for Prolog extensions. *Lecture Notes in Computer Science 660* (1993), pp. 319–338.

[10] **Pereira, L. M., Aparício, J. N., and Alféres, J. J.** Nonmonotonic Reasoning with Well Founded Semantic. In *Proceedings of the 8th International Conference on Logic Programming* (1991), K. Furokawa, Ed., pp. 475–489.

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

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

[13] **Stankevicius, A. G.** Visualización e Interpretación de Programas Lógicos Rebatibles. BSc. thesis, Departamento de Ciencias de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina, February 1999.

[14] **Warren, D. H. D.** An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, United States, October 1983.