Strong and Default Negation in Defeasible Logic Programming

Alejandro J. García Guillermo R. Simari
{ccgarcia, grs}@criba.edu.ar[1]

1 Introduction

Defeasible Logic Programming [8] (*DLP*) is an extension of Logic Programming capturing common-sense reasoning features, that are difficult to express in traditional Logic Programming. The presented language can manage defeasible reasoning, allowing the representation of defeasible and non-defeasible knowledge.

A defeasible logic program is defined in terms of two disjoint sets of rules: a set of *strong rules* for representing strict (sound) knowledge, and a set of *defeasible rules* for representing tentative information. In *DLP*, a query q will succeed when there is an *argument* $\mathcal{A}$ for q that is a *justification* for q. Building a justification involves looking for *counterarguments* that could be *defeaters* for $\mathcal{A}$. Since defeaters are arguments, there may exist defeaters for the defeaters, and so on, thus requiring a dialectical analysis.

DLP considers two different negations: *strong negation*, which is represented by the symbol "$\sim$" which is used for representing contradictory knowledge; and, *default negation*, represented by the symbol "*not*" used for representing incomplete information. In this work, we will show why two types of negations are needed, the differences between them, some properties, and how they could be combined to take advantage of the full expressiveness of the language.

2 Defeasible Logic Programs

In our language, a *literal* "L" is an atom "A" or a negated atom "$\sim A$" [13]. Literals will be allowed in the head and the body of a rule, and default negation just in the body, therefore we will follow the notions of *basic rule* and *extended rule* introduced by Lifschitz in [12].

Definition 2.1 (Strong Rule) *A Strong Rule is an ordered pair, conveniently denoted $Head \leftarrow Body$, whose first component $Head$ is a literal, and whose second component $Body$ is a finite set of literals possibly preceded by the symbol of default negation "not". A strong rule with the head L_0 and body $\{L_1, \ldots, L_n\}$ can also be written as: $L_0 \leftarrow L_1, \ldots, L_n$. As usual, if the body is empty, then a strong rule becomes "$L \leftarrow true$" (or simply "L") and it is called a* fact.

Strong rules correspond to rules used in Extended Logic Programming [9]. Defeasible rules, to be defined below, will add a new representational capability for expressing a weaker link between the head and the body in a rule. A defeasible rule "$Head \prec Body$" is understood as expressing that "*reasons to believe in the antecedent $Body$ provide reasons to believe in the consequent $Head$*" [30]. Defeasible rules are also extended here, allowing the use of default negation in the body of the rule with the obvious meaning.

Definition 2.2 (Defeasible Rule) *A Defeasible Rule is an ordered pair conveniently denoted $Head \prec Body$, whose first component $Head$ is a literal, and whose second component $Body$ is a finite set of literals possibly preceded by the symbol of default negation "not". A*

[1]Department of Computer Science, Universidad Nacional del Sur, Bahía Blanca, Argentina.

defeasible rule with head L_0 *and body* $\{L_1, \ldots, L_n\}$ *can also be written as:* $L_0 \prec L_1, \ldots, L_n$. *If the body is empty, we write "* $L \prec true$*" and we call it a* presumption.

Syntactically, the symbol "$\prec$" is all that distinguishes a defeasible rule from a strong rule. Pragmatically, a defeasible rule is used to represent defeasible knowledge, *i.e.*, tentative information that may be used if nothing could be posed against it. Thus, whereas a strong rule is used to represent non-defeasible information such as "$bird(X) \leftarrow penguin(X)$", which expresses that *"all penguins are birds."*, a defeasible rule is used to represent defeasible knowledge such as "$flies(X) \prec bird(X)$" which expresses that *"birds are presumed to fly"* or *"usually, a bird can fly."*

Besides the syntactic difference, each type of negation is used for a different purpose. We refer the interested reader to [2], where a detailed analysis of negation in Logic Programming and Non-Monotonic Reasoning is addressed. Strong negation is necessary for representing situations such as:

$\sim carnivorous(X) \leftarrow bull(X)$ ("*bulls are not carnivorous*")
$\sim guilty(X) \leftarrow innocent(X)$ ("*an innocent is not guilty*")
$\sim dangerous(X) \prec \sim carnivorous(X)$
("*usually, if an animal is not carnivorous then it is not dangerous.*")

However, there are other situations in which default negation behaves in a better way. For instance, to express that *"if someone cannot be proven guilty, then s/he is innocent"*, and *"usually, search for someone whenever s/he is missing and it has not been proven that s/he is dead"*:

$innocent(X) \leftarrow not\ guilty(X)$
$search\_for(X) \prec missing(X), not\ dead(X)$

Lets observe that both types of negation are needed, for expressing the rule[2]: "*in general, do not cross railway tracks if it cannot be proven that no train is coming*"

$\sim cross\_railway\_tracks \prec not\ \sim train\_is\_coming.$

As we will show below, the semantics of default negation will be different from the approach in conventional logic programming.

A *Defeasible Logic Program* (DLP) is a finite set of strong and defeasible rules. If $\mathcal{P}$ is a DLP, we will distinguish the subset $\mathcal{S}$ of strong rules and the subset $\mathcal{D}$ of defeasible rules in $\mathcal{P}$. When required we will denote $\mathcal{P}$ as $(\mathcal{S},\mathcal{D})$. A *defeasible query* (or simply, a query) is a defeasible rule with empty consequent denoted "$\prec Q_1, \ldots, Q_n$", where each Q_i $(1 \leq i \leq n)$ is a literal. An example of a DLP follows.

Example 2.1

$\sim drink\_water(X) \prec lost\_at\_sea(X)$
$drink\_water(X) \prec lost\_at\_sea(X), collect\_rain(X)$
$lost\_at\_sea(popeye) \prec true$
$collect\_rain(popeye) \prec true$
$dead(X) \prec \sim drink\_water(X)$
$search\_for(X) \prec lost\_at\_sea(X), not\ dead(X)$

□

Given a DLP $\mathcal{P}$ and a defeasible query Q, an *SLD-Defeasible Refutation* of $\mathcal{P} \cup \{Q\}$ is a finite sequence $C_1, C_2, \ldots, C_n$ of variants of strong or defeasible rules of $\mathcal{P}$, provided

[2]This rule was adapted from an example given by John McCarthy as reported in [9]

there exists a sequence $Q{=}Q_0, Q_1, \ldots, Q_n$ of defeasible queries and a sequence $\theta_1, \theta_2, \ldots, \theta_n$ of mgu's such each Q_{i+1} is derived from Q_i and C_{i+1} using θ_{i+1}, and Q_n is the empty rule. If there exist a SLD-defeasible refutation for Q, then the finite set of rules used in the refutation constitutes the *defeasible derivation* for Q.

Two literals are contradictory if they are complementary with respect to strong negation. Thus, *"drink_water(popeye)"* and *"∼drinks_water(popeye)"* are contradictory literals. Observe for example that from the DLP of Example 2.1 it is possible to have defeasible derivations for these contradictory literals. Therefore, in order to allow only one of two contradictory goals to be accepted as a sensible possibility, we need a criterion for choosing between them. This will be developed in the next section, but first, we will introduce the idea of a contradictory set of clauses.

Definition 2.3 (Contradictory DLP) *A defeasible logic program $\mathcal{P}$ is* contradictory *if and only if, it is possible to defeasibly derive from $\mathcal{P}$ a pair of complementary literals.*

The use of strong negation in program rules, enriches language expressiveness, and also allows to represent contradictory knowledge. However, if $\mathcal{P} = (\mathcal{S}, \mathcal{D})$ is a DLP, the set $\mathcal{S}$ of strong rules is used to represent non-defeasible information, so it must express certain internal coherence. Therefore, from now on, we will assume that in every DLP $\mathcal{P}$ the set $\mathcal{S}$ is non-contradictory. If a contradictory set $\mathcal{S}$ is used in a DLP then the same problems of Extended Logic Programming [9] will appear and the corresponding analysis has been done elsewhere.

Although the set $\mathcal{S}$ must be non-contradictory, the set $\mathcal{D}$, and hence $\mathcal{P}$ itself (*i.e.*, $\mathcal{S} \cup \mathcal{D}$), may be contradictory. It is only in this form that a DLP may contain contradictory information. Observe that the DLP of example 2.1 is a contradictory program, but its set $\mathcal{S}$ is not. In the next section we will describe a *justification procedure* capable of deciding acceptance between contradictory goals.

3 Justification Procedure

In *DLP*, answers to queries must be supported by an *argument*. Given a DLP $\mathcal{P}$, an argument $\mathcal{A}$ for a query q, also denoted $\langle \mathcal{A}, q \rangle$, is a subset of *ground instances* of defeasible rules of $\mathcal{P}$, such that: (1) There exists a defeasible derivation for q from $\mathcal{S} \cup \mathcal{A}$, (2) $\mathcal{S} \cup \mathcal{A}$ is non-contradictory, and (3) $\mathcal{A}$ is minimal with respect to set inclusion (*i.e.*, there is no $\mathcal{A}' \subset \mathcal{A}$ such that $\mathcal{A}'$ satisfies the first condition).

When considering a query q, an argument $\mathcal{A}$ for q will be built, but arguments that contradict $\mathcal{A}$ (called rebuttals or counter-arguments) could exist. We say that $\langle \mathcal{A}_1, h_1 \rangle$ *counter-argues* $\langle \mathcal{A}_2, h_2 \rangle$ *at literal* h, if and only if there exists a sub-argument $\langle \mathcal{A}, h \rangle$ of $\langle \mathcal{A}_2, h_2 \rangle$ such that the set $\mathcal{S} \cup \{h_1, h\}$ is contradictory.

Informally, a query q will succeed if the supporting argument for it is not defeated; that argument then becomes a *justification*. In order to establish if $\mathcal{A}$ is a non-defeated argument, counter-arguments that could be *defeaters* for $\mathcal{A}$ are considered, *i.e.*, counter-arguments that for some criterion[3], are preferred to $\mathcal{A}$. Formally, an argument $\langle \mathcal{A}_1, h_1 \rangle$ *defeats* $\langle \mathcal{A}_2, h_2 \rangle$ *at literal* h, if and only if there exists a sub-argument $\langle \mathcal{A}, h \rangle$ of $\langle \mathcal{A}_2, h_2 \rangle$ such that $\langle \mathcal{A}_1, h_1 \rangle$

[3]We have formally defined a particular criterion called *specificity* (see [25, 30]) which allows discrimination between two conflicting arguments. Intuitively, this notion of specificity favors two aspects in an argument: it prefers an argument (1) with greater information content and/or (2) with less use of defeasible rules. In other words, an argument is deemed better than another if it is *more precise* and/or *more concise*. Although one is required, the notion of *defeating argument* can be formulated independently of the particular argument-preference criterion being used. The only restriction on that criterion is the requirement of being a partial order.

counter-argues $\langle \mathcal{A}_2, h_2 \rangle$ at h, and either: (1) $\langle \mathcal{A}_1, h_1 \rangle$ is "better" than $\langle \mathcal{A}, h \rangle$ (then $\langle \mathcal{A}_1, h_1 \rangle$ is a *proper* defeater of $\langle \mathcal{A}_2, h_2 \rangle$); or (2) $\langle \mathcal{A}_1, h_1 \rangle$ is unrelated by the preference order to $\langle \mathcal{A}, h \rangle$ (then $\langle \mathcal{A}_1, h_1 \rangle$ is a *blocking* defeater of $\langle \mathcal{A}_2, h_2 \rangle$).

This definition of defeater comprises the Type I (counter-argument point in the conclusion) and Type II (an inner counter-argument point) defeaters as introduced in Pollock's seminal work [20, 19, 21, 22, 23, 24]. Since defeaters are arguments, there may exist defeaters for the defeaters, and so on. That prompts for a complete dialectical analysis. The formal definitions [29] follow:

Definition 3.1 (Dialectical tree) *Let $\mathcal{A}$ be an argument for h. A* dialectical tree *for $\langle \mathcal{A}, h \rangle$, denoted $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$, is recursively defined as follows:*

1. *A single node labeled with an argument $\langle \mathcal{A}, h \rangle$ with no defeaters (proper or blocking) is by itself the dialectical tree for $\langle \mathcal{A}, h \rangle$. This node is also the root of the tree.*
2. *Let $\langle \mathcal{A}_1, h_1 \rangle$, $\langle \mathcal{A}_2, h_2 \rangle$, ..., $\langle \mathcal{A}_n, h_n \rangle$ be all the defeaters (proper or blocking) for $\langle \mathcal{A}, h \rangle$. We construct the dialectical tree for $\langle \mathcal{A}, h \rangle$, $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$, by labeling the root node with $\langle \mathcal{A}, h \rangle$ and by making this node the parent node of the roots of the dialectical trees for $\langle \mathcal{A}_1, h_1 \rangle$, $\langle \mathcal{A}_2, h_2 \rangle$, ..., $\langle \mathcal{A}_n, h_n \rangle$, i.e., $\mathcal{T}_{\langle \mathcal{A}_1, h_1 \rangle}$, $\mathcal{T}_{\langle \mathcal{A}_2, h_2 \rangle}$, ..., $\mathcal{T}_{\langle \mathcal{A}_n, h_n \rangle}$.*

Definition 3.2 (Marking of a dialectical tree) *Let $\langle \mathcal{A}, h \rangle$ be an argument and $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ its dialectical tree, then:*

1. *All the leaves in $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ are marked as U-nodes.*
2. *Let $\langle \mathcal{B}, q \rangle$ be an inner node of $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$. Then $\langle \mathcal{B}, q \rangle$ will be a U-node iff every child of $\langle \mathcal{B}, q \rangle$ is a D-node. The node $\langle \mathcal{B}, q \rangle$ will be marked as a D-node iff it has at least a child marked as a U-node.*

Certain conditions are required in order to avoid the occurrence of cycles in the dialectical tree. It has been shown elswhere[4] that circular argumentation is a particular case of *fallacious argumentation*. An *acceptable dialectical tree* is a dialectical tree where those undesirable situations are averted. Thus, the notion of *justification* can be properly defined as follows.

Definition 3.3 (Justification) *Let $\mathcal{A}$ be an argument for a literal h, and let $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ be its associated acceptable dialectical tree. The argument $\mathcal{A}$ for a literal h will be a justification iff the root of $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ is a U-node.*

4 Negation in *DLP*

As discussed ealier, a defeasible derivation for a negated literal "$\sim p$" is carried out just as if the "$\sim$" symbol were part, syntactically, of the predicate name, thereby treating "$\sim p$" as an atomic predicate name. Although there can be defeasible derivations for a pair of complementary literals, the inference procedure of *DLP* justifies only one of them. Thus, a query "$\sim q$" succeeds when there exists a justification for "$\sim q$.", and fails when there is no justification for it.

The traditional approach considers that a goal "not p" succeeds when the derivation of "p" fails; and fails when the derivation of "p" succeeds. However, as shown above, in *DLP* the inference procedure is *justification*. Therefore, default negation in *DLP* will be defined in terms of *justification* rather than *defeasible derivation*. Thus, the semantics for default negation will be the following:

[4]Due to space restrictions we refer the interested reader to [29] where a detailed analysis of ill-formed reasoning dialogues are analyzed and definitions are introduced in order to avoid them.

- "*not* q" will succeed when there is no justification for "q", and
- "*not* q" will fail when there is a justification for "q".

The following example shows how negated literals behave in a DLP.

Example 4.1 For the DLP of Example 2.1 there exists an argument $\mathcal{A}$ for "$\sim drink\_water(popeye)$" and an argument $\mathcal{B}$ for "$drink\_water(popeye)$":

$$\mathcal{A} = \left\{ \begin{array}{l} \sim drink\_water(popeye) \prec lost\_at\_sea(popeye) \\ lost\_at\_sea(popeye) \prec true \end{array} \right\}$$

$$\mathcal{B} = \left\{ \begin{array}{l} drink\_water(popeye) \prec lost\_at\_sea(popeye), collect\_rain(popeye) \\ lost\_at\_sea(popeye) \prec true \\ collect\_rain(popeye) \prec true \end{array} \right\}$$

Since $\mathcal{B}$ is "better than" (*i.e.*, *more specific than*) $\mathcal{A}$, results that $\mathcal{B}$ is a justification. Therefore, the query "$drink\_water(popeye)$" succeeds. On the other hand, $\mathcal{A}$ is not a justification and the query "$\sim drink\_water(popeye)$" fails. Using the same DLP, the query "$dead(popeye)$" fails, because the argument

$$\mathcal{C} = \left\{ \begin{array}{l} dead(popeye) \prec \sim drink\_water(popeye) \\ \sim drink\_water(popeye) \prec lost\_at\_sea(popeye) \\ lost\_at\_sea(popeye) \prec true \end{array} \right\}$$

is not a justification. The argument $\mathcal{B}$ is a proper defeater for $\mathcal{C}$ in the counter-argument point "$\sim drink\_water(popeye)$". Since the query "$dead(popeye)$" fails, the query "*not* $dead(popeye)$" succeeds and therefore argument $\mathcal{E}$ is a justification for "$search\_for(popeye)$".

$$\mathcal{E} = \left\{ \begin{array}{l} search\_for(popeye) \prec lost\_at\_sea(popeye), not\ dead(popeye) \\ lost\_at\_sea(popeye) \prec true \end{array} \right\}$$

□

Lets suppose that we regard Example 2.1 above as an Extended Logic Program, excluding for a moment its last rule:

Example 4.2

$$\begin{array}{l} \sim drink\_water(X) \leftarrow lost\_at\_sea(X) \\ drink\_water(X) \leftarrow lost\_at\_sea(X), collect\_rain(X) \\ lost\_at\_sea(popeye) \\ collect\_rain(popeye) \\ dead(X) \leftarrow \sim drink\_water(X) \end{array}$$

□

Now, this program is an Extended Logic Program without "not". For those programs Gelfond and Lifchitz[9] established:

> Let $\mathcal{P}$ be an extended logic program without variables that doesn't contain 'not', and let Lit be the set of ground literals in the language of $\mathcal{P}$. The answer set of $\mathcal{P}$ is the smallest subset A of Lit such that
>
> 1. for any rule $L_0 \leftarrow L_1, \ldots, L_m$ from $\mathcal{P}$ if $L_1, \ldots, L_m \in A$ then $L_0 \in A$
> 2. if A contains a pair of complementary literals, then $A = Lit$.

Thus, the answer set for Example 4.2 calculated according to the definition above will be *Lit*. Clearly, from this program the pair of complementary literals ($\sim drink\_water(popeye)$ and $\sim drink\_water(popeye)$) could be derived.

Now, we will reintroduce the excluded rule, as an Extended Logic Program rule. Observe that the rule in question contains default negation,

Example 4.3

$$\sim drink\_water(X) \leftarrow lost\_at\_sea(X)$$
$$drink\_water(X) \leftarrow lost\_at\_sea(X), collect\_rain(X)$$
$$lost\_at\_sea(popeye)$$
$$collect\_rain(popeye)$$
$$dead(X) \leftarrow \sim drink\_water(X)$$
$$search\_for(X) \leftarrow lost\_at\_sea(X), not\ dead(X)$$

□

For those programs Gelfond and Lifchitz [9] define:

> Let $\mathcal{P}$ be an extended logic program without variables. For any set $A \subset Lit$, let $\mathcal{P}^A$ be the extended logic program obtained from $\mathcal{P}$ by deleting:
>
> 1. each rule that has a formula "not L" in its body with $L \in A$, and
> 2. all formulas of the form "not L" in the bodies of the remaining rules."
>
> Clearly $\mathcal{P}^A$ doesn't contain not, so that its answer set is already defined. If this answer set coincides with A, then we say that A is an answer set for $\mathcal{P}$.

For our example[5], when represented as an Extended Logic Program, the answer set of $\mathcal{P}^A$ will be *Lit* again. In the DLP framework that situation does not occur.

Thus, the main differences of *DLP* with other extensions of logic programming are:

- our approach provides a way of choosing between contradictory literals, and
- "*not p*" succeeds when the justification of "*p*" fails and not just when a derivation of "*p*" fails.

The following results show that in *DLP* negation satisfies the *Coherence Principle* formulated in [1]:

Proposition 4.1 *If a query "∼p" succeeds then the query "not p" also succeeds.*
*Proof:*if "∼*p*" succeeds then there is a justification for "∼*p*", therefore, there could be no justification for "*p*", and hence "*not p*" also succeeds.

It is easy to see that the converse proposition does not hold. If there is no justification for a literal "*p*" it could also be the case that there is no justification for "∼*p*". In Example 2.1 the query "*not dead*(*popeye*)" succeeds whereas the query "∼*dead*(*popeye*)" fails .

The definition of argument states that an argument must be free of contradictions. However, it says nothing regarding default negation. Example 4.4 shows that it is possible to construct an argument where "*p*" and "*not p*" are defeasible derived.

Example 4.4 Consider the following DLP:

[5]Clearly, variables could be consistently replaced throughout by the only individual constant *popeye* without altering the situation.

$a \prec b, c$	$p \prec e$
$b \prec p$	$\sim p \prec d$
$c \prec not\ p$	d
	e

From this DLP the literal "a" has the argument

$$\mathcal{A} = \{(a \prec b, c), (b \prec p), (p \prec e), (c \prec not\ p)\}$$

Note that in this argument the sub-queries "p" and "*not* p" have been both defeasibly derived. This is possible because there is a defeasible derivation for "p", but there is no justification for p because it has the blocking defeater "$\{\sim p \prec d\ \}$". □

Proposition 4.2 establishes an important "coherence" property, showing that an argument like the one of Example 4.4 could never be a justification.

Proposition 4.2 *If the sub-queries "l" and " not l" are used in the construction of an argument $\mathcal{A}$, then $\mathcal{A}$ cannot be a justification.*
Proof: if "*not* l" has been derived then there is no justification for "l". Hence, every argument for "l" has a non-defeated defeater $\mathcal{B}$ (at least one must exist). Therefore, $\mathcal{B}$ is a non-defeated defeater for $\mathcal{A}$, so $\mathcal{A}$ cannot be a justification.

In order to analyze some differences between the two types of negations in *DLP* consider the following DLPs .

$\mathcal{P}_1$	$\mathcal{P}_2$
$p \leftarrow not\ q$	$p \leftarrow \sim q$
	$\sim q \prec true$

Using program $\mathcal{P}_2$ queries "p" and "$\sim q$" succeed, whereas, using $\mathcal{P}_1$ only query "p" succeeds. When the presumption "$q \prec true$" is added to both programs (obtaining $\mathcal{P}_1'$ and $\mathcal{P}_2'$), then from $\mathcal{P}_2'$ there is no possible justification, since one presumption acts as a blocking defeater of the other. However, from $\mathcal{P}_1'$ there is a justification for "q".

$\mathcal{P}_1'$	$\mathcal{P}_2'$
$p \leftarrow not\ q$	$p \leftarrow \sim q$
$q \prec true$	$\sim q \prec true$
	$q \prec true$

Finally, if the fact "$\sim p$" is added to $\mathcal{P}_1$ the set of strong rules becomes contradictory. However, the fact "$\sim p$" can be added to $\mathcal{P}_2$, and in this case only "$\sim p$" and "$\sim q$" have a justification (see the program below).

$\mathcal{P}_1''$	$\mathcal{P}_2''$
$p \leftarrow not\ q$	$p \leftarrow \sim q$
$\sim p$	$\sim q \prec true$
	$\sim p$

The previous analysis shows that these two types of negation cannot be defined in terms of each other.

5 Related Work

An early attempt to introduce defeasible reasoning programming with specificity was Nute's d-Prolog [16, 17]. The inference engine of d-Prolog is based on the theoretical developments reported in [18, 15] and recently described in [6].

The language of d-Prolog provides facilities to define *absolute rules*, like *"every bat is a mammal"*, *defeasible rules* such as *"birds fly"*, and *defeater rules* like *"sick birds do not fly"*. The purpose of defeater rules is to account for the exceptions to defeasible rules. For instance, given the defeasible rule *"birds fly"*, the defeater rule *"sick birds do not fly"* will stop us from concluding that *"Tweety flies"*, in the presence of the fact *"Tweety is a sick bird"*.

Unlike Nute's d-Prolog, our system does not need to be supplied with *defeater rules*. The system will find the counterarguments and defeaters among the arguments it is able to construct, and using specificity constraints will take decisions about their relevance. The system will find those specificity constraints by comparing the way arguments are *constructed*. In other words, for the same example above, we will have the same absolute rule (fact) *"Tweety is a sick bird"* and two defeasible rules: *"birds fly"* and *"sick birds do not fly"*. When consulted about the status of the fact *"Tweety flies"* knowing that *"Tweety is a sick bird"*, our system will decide that believing *"Tweety flies"* is not justified. The reason is that the argument *"Tweety is a sick bird"* and *"sick birds do not fly"* is more specific than the argument *"Tweety is a sick bird"* and *"birds fly"*.

This characteristic of our language is very important because it maintains the declarative nature of the knowledge represented in it; *i.e.* the interaction among the *pieces* of knowledge is not expressed *in* the language in any way but as a result of the totality of the knowledge. For that reason, the burden of the defeasible inference falls upon the *language processor*, *i.e.* our system, which figures out the interactions, instead on the *knowledge encoder*, *i.e.* the programmer. The programmer does not have to evoke the *behavior* of the representation in order to add procedural control to the defeasible rules, in case he wants to modify it.

We claim that adding special kinds of defeasible rules amounts to a retreat from the goal of having a declarative language. It is true that reaching that goal is a difficult task, but the implementation of experimental systems, such as our system, should strive to maintain that ideal rather than start by giving it up. This constraint in the design of a language translates the burden of computing to the process of searching the solution space because there is no *procedural* guidance. That search space in turn can be quite large for non trivial situations. Much of the effort expended in the implementation was put to this task of doing an efficient search.

Other formalisms for defeasible argumentation have been separately developed. In [7] P. Dung has proposed a very abstract and general argument-based framework, where he completely abstracts from the notions of argument and defeat. Inspired by legal reasoning, H. Prakken and G. Sartor [27, 28] have developed an argumentation system that, like ours, uses the language of extended logic programming. They introduce a *dialectical proof theory* for an argumentation framework fitting the abstract format developed by Dung, Kowalski *et al.* [7, 4]. R. Kowalski and F. Toni [11] have outlined a formal theory of argumentation, in which defeasibility is stated in terms of non-provability claims. They argue that defeasible reasoning with rules of the form P if Q can be understood as "exact" reasoning with rules of the form P if Q and S cannot be shown, where S stands for one or more defeasible "non-provability claims". Other related works are by Vreeswijk [31], Bondarenko [3], and Loui [14]. The interested reader is referred to the following surveys in defeasible argumentation: Prakken & Vreesvijk [26], and Chesñevar, Maguitman & Loui [5].

In the area of Logic Programming, Gelfond and Lifschitz in *Logic Programming with Classical Negation* [9] introduced extended logic programs where classical negation as well

as negation as failure (default negation) can be used. However, there is no criterion for deciding between contradictory goals. Katsumi Inoue in *Extended Logic Programming with Default Assumptions* [10] expands Gelfond and Lifschitz's work in order to avoid the problems with contradictory programs. His approach resembles a defeasible argumentation system, but differs in that it has no preference criterion defined for deciding between contradictory explanations.

6 Conclusion

Defeasible Logic Programming captures common-sense reasoning features, that are difficult to express in traditional Logic Programming, and other extensions. Since strong and default negation are both available in the language, contradictory and incomplete information can be represented.

The defeasible argumentation formalism provides a criterion for deciding whether a negated literal is accepted: a query "$\sim q$" succeeds when there exists a justification for "$\sim q$", and fails when there is no justification for it. On the other hand, default negation has different semantics: "*not* q" will succeed when there is no justification for "q", and "*not* q" will fail when there is a justification for "q". We have shown that two kinds of negation are needed and that they can be combined for improving the expressiveness of the language.

References

[1] José J. Alferes and Luis M. Pereira. Contradiction: when avoidance equals removal (part i and ii). In *Proc. of Extensions of Logic Programming, 4th International Workshop ELP'93*. St. Andrews U.K., March 1993.

[2] José J. Alferes, Luis M. Pereira, and Teodor Przymusinski. Strong and explicit negation in non-monotonic reasoning and logic programming. In *Logics in Artificial Intelligence (JELIA'96)*, pages 143–163. Springer, 1996.

[3] A. Bondarenko, P.M. Dung, R.A. Kowalski, and F. Toni. An abstract, argumentation-theoretic approach to default reasoning. *Artificial Intelligence*, 93:63–101, 1997.

[4] A. Bondarenko, F. Toni, and R.A. Kowalski. An assumption-based framework for non-monotonic reasoning. *Proc. 2nd. International Workshop on Logic Programming and Non-monotonic Reasoning*, pages 171–189, 1993.

[5] C. I. Chesñevar, A. Maguitman, and R.P.Loui. Logical models of arguments. *submitted to ACM Computing Surveys*, 1998.

[6] Michael A. Covington, Donald Nute, and Andre Vellino. *Prolog Programming in Depth.* Prentice-Hall, 1997.

[7] Phan M. Dung. On the acceptability of arguments and its fundamental role in non-monotonic reasoning and logic programming and n-person games. *Artificial Intelligence*, 77:321–357, 1995.

[8] Alejandro J. García and Guillermo R. Simari. Defeasible logic programming. Technical report, Computer Science Department, Universidad Nacional del Sur, October 1998. Technical Report GIIA-1998-20.

[9] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In D. Warren and P. Szeredi, editors, *Proc. ICLP*, pages 579–597. MIT Press, 1990.

[10] K. Inoue. Extended logic programming with default assumptions. In *Proc of 8th. ICLP*, 1991.

[11] Robert A. Kowalski and Francesca Toni. Abstract argumentation. *Artificial Intelligence and Law*, 4(3-4):275–296, 1996.

[12] Vladimir Lifschitz. Foundations of logic programs. In Gerhard Brewka, editor, *Principles of Knowledge Representation*. CSLI Publications, 1996.

[13] John W. Lloyd. *Foundations of Logic Programmming*. Springer-Verlag, 1987.

[14] Ronald P. Loui, Jeff Norman, Joe Altepeter, Dan Pinkard, Dan Craven, Jessica Lindsay, and Mark Foltz. Progress on room 5: A testbed for public interactive semi-formal legal argumentation. In *Proc. of the 6th. International Conference on Artifcial Intelligence and Law*, July 1997.

[15] D. Nute. Defeasible logic. In C.J. Hogger D.M. Gabbay and J.A.Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol 3, Nonmonotonic Reasoning and Uncertain Reasoning*, pages 355–395. Oxford University Press, 1994.

[16] Donald Nute. Ldr: a logic for defeasible reasoning. research report 01-0013. Technical report, Advanced Computational Methods Center (ACMC). University of Georgia., 1986.

[17] Donald Nute. Defeasible reasoning: a philosophical analysis in PROLOG. In James H. Fetzer, editor, *Aspects of Artificial Intelligence*, pages 251–288. Kluwer Academic Publishers, 1988.

[18] Donald Nute. Basic defeasible logic. In Luis Fariñas del Cerro, editor, *Intensional Logics for Programming*. Claredon Press, Oxford, 1992.

[19] John Pollock. A theory of defeasible reasoning. In Inc. John Wiley & Sons, editor, *International Journal of Intelligent Systems*, volume 6, pages 33–54, 1991.

[20] John L. Pollock. Defeasible Reasoning. *Cognitive Science*, 11:481–518, 1987.

[21] John L. Pollock. Self-defeating arguments. *Minds and Machines (Special issue: defeasible reasoning)*, 1(4), November 1991.

[22] John L. Pollock. New foundations for practical reasoning. *Minds and Machines*, 2:113–144, 1992.

[23] John L. Pollock. *Cognitive Carpentry: A Blueprint for How to Build a Person*. Massachusetts Institute of Technology, 1995.

[24] John L. Pollock. Oscar - A general purpose defeasible reasoner. *Journal of Applied Non-Classical Logics*, 6:89–113, 1996.

[25] David L. Poole. On the Comparison of Theories: Preferring the Most Specific Explanation. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 144–147. IJCAI, 1985.

[26] H. Prakken and G. Vreeswijk. Logical systems for defeasible argumentation (to appear). In Gabbay, editor, *Handbook of Philosophical Logic, second edition*. 1998.

[27] Henry Prakken and Giovanni Sartor. A system for defeasible argumentation, with defeasible priorities. In *Proc. of the International Conference on Formal Aspects of Practical Reasoning, Bonn, Germany*. Springer Verlag, 1996.

[28] Henry Prakken and Giovanni Sartor. Argument-based logic programming with defeasible priorities. *Journal of Applied Non-classical Logics*, 7(25-75), 1997.

[29] Guillermo R. Simari, Carlos I. Chesñevar, and Alejandro J. García. The role of dialectics in defeasible argumentation. In *Anales de la XIV Conferencia Internacional de la Sociedad Chilena para Ciencias de la Computación*. Universidad de Concepción, Concepción (Chile), November 1994.

[30] Guillermo R. Simari and Ronald P. Loui. A Mathematical Treatment of Defeasible Reasoning and its Implementation. *Artificial Intelligence*, 53:125–157, 1992.

[31] Gerard A.W. Vreeswijk. Abstract argumentation systems. *Artificial Intelligence*, 90:225–279, 1997.