Disjunctive Logic Programs with Inheritance

Francesco Buccafurri
DIMET – Universitá di Reggio Calabria
89100, Reggio Calabria, Italia
bucca@ns.ing.unirc.it

Wolfgang Faber, Nicola Leone
Institut für Informationssysteme E184/2
Technische Universität Wien
1040 Vienna, Austria
{faber,leone}@dbai.tuwien.ac.at

Abstract

The paper proposes a new knowledge representation language, called DLP$^<$, which extends disjunctive logic programming (with strong negation) by inheritance. The addition of inheritance enhances the knowledge modeling features of the language providing a natural representation of default reasoning with exceptions.

A declarative model-theoretic semantics of DLP$^<$ is provided, which is shown to generalize the answer set semantics of disjunctive logic programs.

The knowledge modeling features of the language are illustrated by encoding classical nonmonotonic problems in DLP$^<$.

The complexity of DLP$^<$ is analyzed, proving that inheritance does not cause any computational overhead, as reasoning in DLP$^<$ has exactly the same complexity as reasoning in disjunctive logic programming. This is confirmed by the existence of an efficient translation from DLP$^<$ to plain disjunctive logic programming. Using this translation, an advanced KR system supporting the DLP$^<$ language has been implemented on top of the `dlv` system.

1 Introduction

Disjunctive logic programs are logic programs where disjunction is allowed in the heads of the rules and (NAF) negation may occur in the bodies of the rules. Such programs are now widely recognized as a valuable tool for knowledge representation and commonsense reasoning [5, 18, 13]. One of the attractions of disjunctive logic programming is its ability to naturally model incomplete knowledge [5, 18]. The need to differentiate atoms, which are false because of the failure to prove them true (NAF, or CWA negation), from

atoms the falsity of which is explicitly provable, led to extend disjunctive logic programs by strong negation [13]. Strong negation, permitted also in the rules' heads, further enhances the knowledge modeling features of the language, and its usefulness is widely acknowledged in the literature [1, 5, 16, 2, 26, 3]. However, it does not allow to represent default reasoning with exceptions in a direct and natural way. Indeed, to render a default rule r *defeasible*, r must be equipped by an extra negative literal, which "blocks" inferences from r for abnormal instances [14]. For instance, to encode the famous NMR example stating that birds *normally* fly while penguins do not fly, one should write the rule $fly(X) \leftarrow bird(X),\ not\ \neg fly(X)$ along with the fact $\neg fly(penguin)$.[1]

This paper proposes an extension of disjunctive logic programming by inheritance, called DLP$^<$. The addition of inheritance enhances the knowledge modeling features of the language. Possible conflicts are solved in favour of the rules which are "more specific" according to the inheritance hierarchy. This way, a direct and natural representation of default reasoning with exceptions is achieved (e.g., defeasible rules do not need to be equipped with extra literals as above – see section 4).

The main contribution of the paper are the following.

- We formally define the DLP$^<$ language, providing a declarative model theoretic semantics of DLP$^<$, which is shown to generalize the answer set semantics of [13].

- We illustrate the knowledge modeling features of the language by encoding classical nonmonotonic problems in DLP$^<$. Interestingly, DLP$^<$ supplies a natural representation also of frame axioms.

- We analyze the computational complexity of reasoning over DLP$^<$ programs. Importantly, while inheritance enhances the knowledge modeling ability of disjunctive logic programming, it does not cause any computational overhead, as reasoning in DLP$^<$ has exactly the same complexity as reasoning in disjunctive logic programming.

- We compare DLP$^<$ to related works proposed in the literature. In particular, we evidentiate the differences between DLP$^<$ and Disjunctive Ordered Logic ($\mathcal{DOL}$) [7]; we point out the relation to the answer set semantics of [13]; we compare DLP$^<$ with prioritized disjunctive logic programs [26], and analyze its relationships to inheritance networks [28] and to updates of logic programs.

- We implement a DLP$^<$ system. To this end, we first design an efficient translation from DLP$^<$ to plain disjunctive logic programming. Then, using this translation, we implement a DLP$^<$ evaluator on top

[1] *not* and $\neg$ denote the weak negation symbol and the strong negation symbol, respectively.

of the `dlv` system [11]. The system prototype can be freely retrieved from [12]. To the best of our knowledge, this is the first fully operational system supporting a prioritized disjunctive logic programming language.

The long version of this paper, containing proofs of all theorems, further DLP$^<$ encoding, and details can be retrieved from the above mentioned web page.

2 Syntax of DLP$^<$

Let the following disjoint sets be given: a set $\mathcal{V}$ of *variables*, a set Π of *predicates*, a set Λ of *constants*, and a finite partially ordered set of symbols $(\mathcal{O}, <)$, where $\mathcal{O}$ is a set of strings, called *object identifiers*, and $<$ is a strict partial order (i.e., the relation $<$ is: (i) irreflexive – $c \not< c \ \ \forall c \in \mathcal{O}$, and (ii) transitive – $a < b \wedge b < c \Rightarrow a < c \ \ \forall a, b, c \in \mathcal{O}$).

A *term* is either a constant in Λ or a variable in $\mathcal{V}$ (note that function symbols are not considered in this paper).

An *atom* is a construct of the form $a(t_1, ..., t_n)$, where a is a *predicate* of arity n in Π and $t_1, ..., t_n$ are terms.

A *literal* is either a *positive literal* p or a *negative literal* $\neg p$, where p is an atom ($\neg$ is the *strong negation* symbol). Two literals are *complementary* if they are of the form p and $\neg p$, for some atom p.

Given a literal L, $\neg.L$ denotes its complementary literal. Accordingly, given a set A of literals, $\neg.A$ denotes the set $\{\neg.L \mid L \in A\}$.

A *rule* r is an expression of the form:

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \cdots, b_k, \mathit{not}\ b_{k+1}, \cdots, \mathit{not}\ b_m, \qquad n \geq 1,\ m \geq 0$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are literals, and *not* is the *negation as failure* symbol. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k, \mathit{not}\ b_{k+1}, ..., \mathit{not}\ b_m$ is the *body* of r. $b_1, ..., b_k$ is called the *positive part* of the body of r and $\mathit{not}\ b_{k+1}, ..., \mathit{not}\ b_m$ is called the *NAF (negation as failure) part* of the body of r. We often denote the sets of literals appearing in the head, in the positive part of the body, and in the NAF part of the body of a rule r by $Head(r)$, $Body^+(r)$, and $Body^-(r)$, respectively.

An *object* o is a pair $\langle oid(o), \Sigma(o)\rangle$, where $oid(o)$ is an object identifier in $\mathcal{O}$ and $\Sigma(o)$ is a (possibly empty) set of rules.

A *knowledge base* on $\mathcal{O}$ is a set of objects, one for each element of $\mathcal{O}$.

Given a knowledge base $\mathcal{K}$ and an object identifier $o \in \mathcal{O}$, the DLP$^<$ *program for* o (*on* $\mathcal{K}$) is the set of objects $\mathcal{P} = \{(o', \Sigma(o')) \in \mathcal{K} \mid o = o' \text{ or } o < o'\}$.

The relation $<$ induces a partial order on $\mathcal{P}$ in the obvious way, that is, given $o_i = (oid(o_i), \Sigma(o_i))$ and $o_j = (oid(o_j), \Sigma(o_j))$, $o_i < o_j$ iff $oid(o_i) < oid(o_j)$ (read "o_i is more specific than o_j").

A term, an atom, a literal, a rule, or program is *ground* if no variable appears in it.

Informally, a knowledge base can be viewed as a set of *objects* embedding the definition of their properties specified through disjunctive logic rules, organized in a IS-A (inheritance) hierarchy (induced by the relation $<$). A program $\mathcal{P}$ for an object o on a knowledge base $\mathcal{K}$ consists of the portion of $\mathcal{K}$ "seen" from o looking up in the IS-A hierarchy. Thanks to the inheritance mechanism, $\mathcal{P}$ incorporates the knowledge explicitely defined for o plus the knowledge inherited from the higher objects.

If a knowledge base admits a *bottom* element (i.e., an object less than all the other objects, by the relation $<$), we usually refer to the knowledge base as "program", since it is equal to the program for the bottom element.

Moreover, we represent the transitive-reduction of the relation $<$ on the objects.[2] In order to represent the membership of a pair of objects (resp., object identifiers) (o_2, o_1) to the transitive-reduction of $<$ we use the notation $o_2 : o_1$, and say that o_2 is a *sub-object* of o_1.

Example 1 Consider the following program $\mathcal{P}$:

$$\begin{array}{ll} o_1 & \{\ a \vee \neg b \leftarrow c, not\ d\ \ \} \\ o_2 : o_1 & \{\ b \leftarrow \quad \neg a \vee c \leftarrow \quad c \leftarrow b\ \} \end{array}$$

$\mathcal{P}$ consists of two objects o_1 and o_2. o_2 is a sub-object of o_1. According to the convention illustrated above, the knowledge base on which $\mathcal{P}$ is defined coincides with $\mathcal{P}$, and the object for which $\mathcal{P}$ is defined is o_2 (the bottom object). ∎

3 Semantics of DLP$^<$

In this section we assume that a knowledge base $\mathcal{K}$ is given and an object o has been fixed. Let $\mathcal{P}$ be the DLP$^<$ program for o on $\mathcal{K}$. The *Universe* $U_\mathcal{P}$ of $\mathcal{P}$ is the set of all constants appearing in the rules. The *Base* $B_\mathcal{P}$ of $\mathcal{P}$ is the set of all possible ground literals constructible from the predicates appearing in the rules of $\mathcal{P}$ and the constants occurring in $U_\mathcal{P}$. Note that, unlike in traditional logic programming the Base of a DLP$^<$ program contains both positive and negative literals. Given a rule r occurring in $\mathcal{P}$, a *ground instance* of r is a rule obtained from r by replacing every variable X in r by $\sigma(r)$, where σ is a mapping from the variables occurring in r to the constants in $U_\mathcal{P}$. We denote by $ground(\mathcal{P})$ the (finite) multiset of all instances of the rules occurring in $\mathcal{P}$. The reason why $ground(\mathcal{P})$ is a multiset is that a rule may appear in several different objects of $\mathcal{P}$, and we require the respective ground instances be distinct. Hence, we can define a function obj_of from ground instance of rules in $ground(\mathcal{P})$ onto the set $\mathcal{O}$ of the object identifiers, associating with a ground instance $\overline{r}$ of r the (unique) object of r.

A subset of ground literals in $B_\mathcal{P}$ is said to be *consistent* if no pair of complementary literals is in it. An *interpretation* I is a consistent subset

[2] (a, b) is in the transitive-reduction of $<$ iff $a < b$ and there is no c such that $a < c$ and $c < b$.

of $B_{\mathcal{P}}$. Given an interpretation $I \subseteq B_{\mathcal{P}}$, a ground literal (either positive or negative) L is *true* w.r.t. I if $L \in I$ holds. L is *false* w.r.t. I otherwise.

Given a rule $r \in ground(\mathcal{P})$, the head of r is *true* in I if at least one literal of the head is true w.r.t I. The body of r is *true* in I if: (i) every literal in $Body^+(r)$ is true w.r.t. I, and (ii) every literal in $Body^-(r)$ is false w.r.t. I. Rule r is *satisfied* in I if either the head of r is true in I or the body of r is not true in I.

Two ground rules r_1 and r_2 are *conflicting on* L if $L \in Head(r_1)$ and $\neg.L \in Head(r_2)$.

Next we introduce the concept of a *model* for a DLP$^<$-program. Unlike in traditional logic programming, the notion of satisfiability of rules is not sufficient for this goal, as it does not take into account the presence of explicit contradictions. Hence, we first present some preliminary definitions.

Definition 1 For an interpretation I, two conflicting ground rules $r_1, r_2 \in ground(\mathcal{P})$, such that $L \in Head(r_2)$, we say that r_1 *overrides* r_2 *on* L *in* I if: 1) $obj_of(r_1) < obj_of(r_2)$, 2) $\neg.L \in I$, and 3) the body of r_2 is true in I.

A rule $r \in ground(\mathcal{P})$ is *overridden in* I if for each $L \in Head(r)$ there exists $r_1 \in ground(\mathcal{P})$ such that r_1 overrides r on L in I. ∎

Intuitively, the notion of overriding allows us to solve conflicts arising between rules with complementary heads. For instance, suppose that both a and $\neg a$ are derivable in I from rules r and r', respectively. If r is more specific than r' in the inheritance hierarchy, then r' is overruled, meaning that a should be preferred to $\neg a$ because it is derivable from a more trustable rule.

Example 2 Consider the program $\mathcal{P}$ of Example 1. Let $I = \{\neg a, b, c\}$ be an interpretation. Rule $\neg a \vee c \leftarrow$ in the object o_2 overrides rule $a \vee \neg b \leftarrow c, not\ d$ in o_1 on the literal a in I. Moreover, rule $b \leftarrow$ in o_2 overrides rule $a \vee \neg b \leftarrow c, not\ d$ in o_1 on the literal $\neg b$ in I. Thus, the rule $a \vee \neg b \leftarrow c, not\ d$ in o_1 is overridden in I. ∎

Definition 2 Let I be an interpretation for $\mathcal{P}$. I is a *model for* $\mathcal{P}$ if every rule in $ground(\mathcal{P})$ is satisfied or overridden in I. I is a *minimal model for* $\mathcal{P}$ if no (proper) subset of I is a model for $\mathcal{P}$. ∎

Definition 3 Given an interpretation I for $\mathcal{P}$, the *reduction of* $\mathcal{P}$ *w.r.t.* I, denoted by $G_I(\mathcal{P})$, is the set of rules obtained from $ground(\mathcal{P})$ by 1) removing every rule overridden in I, 2) removing every rule r such that $Body^-(r) \cap I \neq \emptyset$, 3) removing the NAF part from the bodies of the remaining rules. ∎

Example 3 Consider the program $\mathcal{P}$ of Example 1. Let I be the interpretation $\{\neg a, b, c\}$. As shown in Example 2, rule $a \vee \neg b \leftarrow c, not\ d$ is overridden in I. Thus, $G_I(\mathcal{P})$ is the set of rules $\{\neg a \vee c \leftarrow, b \leftarrow, c \leftarrow b\}$. Consider now the interpretation $M = \{a, b, c\}$. It is easy to see that $G_M(\mathcal{P}) = \{a \vee \neg b \leftarrow c, \neg a \vee c \leftarrow, b \leftarrow, c \leftarrow b\}$. ∎

We observe that the reduction of a program is simply a set of ground rules. Given a set S of ground rules, we denote by $pos(S)$ the positive disjunctive program (called the *positive version of* S), obtained from S by considering each negative literal $\neg p(\bar{X})$ as a positive one with predicate symbol $\neg p$.

Definition 4 Let M be a model for $\mathcal{P}$. We say that M is a *(*DLP$^<$*-)answer set* for $\mathcal{P}$ if M is a minimal model of the positive version $pos(G_M(\mathcal{P}))$ of $G_M(\mathcal{P})$. ∎

Example 4 Consider the program $\mathcal{P}$ of Example 1:

It is easy to see that the interpretation I of Example 3 is not an answer set for $\mathcal{P}$. Indeed, although I is a model for $pos(G_I(\mathcal{P}))$ it is not minimal, since the interpretation $\{b, c\}$ is a model for $pos(G_I(\mathcal{P}))$, too. Note that the interpretation $I' = \{b, c\}$ is not an answer set for $\mathcal{P}$ either, because $G_{I'}(\mathcal{P}) = \{a \vee \neg b \leftarrow c, \neg a \vee c \leftarrow, b \leftarrow, c \leftarrow b\}$, and I' is not a model for $pos(G_{I'}(\mathcal{P}))$, since the rule $a \vee \neg b \leftarrow c$ is not satisfied in I'.

On the other hand, the interpretation M of Example 3 is an answer set for $\mathcal{P}$, since M is a minimal model for $pos(G_M(\mathcal{P}))$. Moreover, it can be easily realized that M is the only answer set for $\mathcal{P}$. ∎

4 Knowledge Representation with DLP$^<$

In this section, we present a number of examples which illustrate how knowledge can be represented using DLP$^<$. To start, we show the DLP$^<$ encoding of a classical example of nonmonotonic reasoning.

Example 5 Consider the following program $\mathcal{P}$ with $\mathcal{O}(\mathcal{P})$ consisting of three objects *bird*, *penguin* and *tweety*, such that *penguin* is a sub-object of *bird* and *tweety* is a sub-object of *penguin*:

$$\begin{array}{ll} bird & \{\ flies \leftarrow\ \} \\ penguin : bird & \{\ \neg flies \leftarrow\ \} \\ tweety : penguin & \{\ \} \end{array}$$

Unlike in traditional logic programming, our language supports two types of negation, that is *strong negation* and *negation as failure*. Strong negation is useful to express negative pieces of information under the complete information assumption. Hence, a negative fact (by strong negation) is true only if it is explicitely derived from the rules of the program. As a consequence, the head of rules may contain also such negative literals and rules can be conflicting on some literals. According to the inheritance principles, the ordering relationship between objects can help us to assign different levels of reliability to the rules, allowing us to solve possible conflicts. For instance, in our example, the contradicting conclusion *tweety both flies and*

does not fly seems to be entailed from the program (as *tweety* is a *penguin* and *penguins* are *birds*, both $flies$ and $\neg flies$ can be derived from the rules of the program). However, this is not the case. Indeed, the "lower" rule $\neg flies \leftarrow$ specified in the object *penguin* is considered as a sort of refinement to the first general rule, and thus the meaning of the program is rather clear: *tweety does not fly*, as tweety is a penguin. That is, $\neg flies \leftarrow$ is preferred to the default rule $flies \leftarrow$ as the hierarchy explicitely states the specificity of the former. Intuitively, there is no doubt that $M = \{\neg flies\}$ is the only reasonable conclusion. ∎

The next example, from the field of database authorizations, combines the use of both weak and strong negation.

Example 6 Consider the following knowledge base representing a set of security specification about a simple *part-of* hierarchy of objects.

o_1 $\{ r_1 : authorize(bob) \leftarrow not\ authorize(ann)$
$r_2 : authorize(ann) \vee authorize(tom) \leftarrow not\ \neg authorize(alice)\ \}$
$o_2 : o_1\ \{ r_3 : \neg authorize(alice) \leftarrow \}$
$o_3 : o_1\ \{ r_4 : \neg authorize(bob) \leftarrow \}$

Objects o_2 is part-of the object o_1 as well as o_3 is part-of o_1. Access authorizations to objects are specified by rules with head predicate *authorize* and subjects to which authorizations are granted appear as arguments. Strong negation is utilized to encode negative authorizations that represent explicit denials. Negation as failure is used to specify the absence of authorization (either positive or negative). Inheritance implements the automatic propagation of authorizations from an object to all its sub-objects. The overriding mechanism allows us to represent exceptions: for instance, if an object o inherits a positive authorization but a denial for the same subject is specified in o, then the negative authorization prevails on the positive one.

Consider the program $\mathcal{P}_{o_2} = \{(o_1, \{r_1, r_2\}), (o_2, \{r_3\})\}$ for the object o_2 on the above knowledge base. This program defines the access control for the object o_2. Thanks to the inheritance mechanism, authorizations specified for the object o_1, to which o_2 belongs, are propagated also to o_2. It consists of rules r_1, r_2 (inherited from o_1) and r_3. Rule r_1 states that *bob* is authorized to access object o_2 provided that no authorization for *ann* to access o_2 exists. Rule r_2 authorizes either *ann* or *tom* to access o_2 provided that no denial for *alice* to access o_2 is derived. Finally, rule r_3 defines a denial for *alice* to access object o_2. Due to the absence of the authorization for *ann*, the authorization to *bob* of accessing the object o_2 is derived (by rule r_1). Further, the explicit denial to access the object o_2 for *alice* (rule r_3) allows to derive neither authorization for *ann* nor for *tom* (by rule r_2). Hence, the only answer set of this program is $\{authorize(bob), \neg authorize(alice)\}$.

Consider now the program $\mathcal{P}_{o_3} = \{(o_1, \{r_1, r_2\}), (o_3, \{r_4\})\}$ for the object o_3. Rule r_4 defines a denial for *bob* to access object o_3. The authorization

for *bob* (defined by rule r_1) is no longer derived. Indeed, even if rule r_1 allows to derive such an authorization due to the absence of authorizations for *ann*, it is overridden by the explicit denial (rule r_4) defined in the object o_3 (i.e., at a more specific level). The body of rule r_2 inherited from o_1 is true for this program since no denial for alice can be derived, and it entails a mutual exclusive access to object o_3 for *ann* and *tom* (note that no other head contains $authorize(ann)$ or $authorize(bob)$). The program $\mathcal{P}_{o_3}$ has two answer sets, namely $\{authorize(ann), \neg authorize(bob)\}$ and $\{authorize(tom), \neg authorize(bob)\}$ representing two alternative authorization sets utilizable to grant the access to the object o_3. ∎

Solving the Frame Problem

The frame problem has first been addressed by McCarthy and Hayes [22], and in the meantime a lot of research has been conducted to overcome it (see e.g. [27] for a survey).

In short, the frame problem arises in planning, when actions and fluents are specified: An action affects some of the fluents, but all unrelated fluents should remain as they are. In most formulations using classical logic, one must specify for every pair of actions and unrelated fluents that the fluent remains unchanged. Clearly this is an undesirable overhead, since with n actions and m fluents, $n \times m$ clauses might have to be specified.

Instead, it would be nice to be able to specify for each fluent that it *"normally remains valid"* and that only actions which explicitly entail the contrary can change them.

Indeed, this goal can be achieved in a very elegant way using DLP$^<$: One object contains the rules which specify *inertia* (the fact that fluents normally do not change). Another object inherits from it and specifies the actions and the effects of actions — in this way a very natural, straightforward and effective representation is achieved, which avoids the frame problem.

Example 7 As an example we show how the famous Yale Shooting Problem, which is due to Hanks and McDermott [15], can be represented and solved with DLP$^<$:

The scenario involves an individual (or in a less violent version a turkey), who can be shot with a gun. There are two fluents, *alive* and *loaded*, which intuitively mean that the individual is alive and that the gun is loaded, respectively. There are three actions, *load*, *wait* and *shoot*. Loading has the effect that the gun is loaded afterwards, shooting with the loaded gun has the effect that the individual is no longer alive afterwards (and also that the gun is unloaded, but this not really important), and waiting has no effects. It is also known that initially the individual is alive, and that first the gun is loaded, and after waiting, the gun is shot with. The question is: Which fluents hold after these actions and between them?

In our encoding, the *inertia* object contains the defaults for the fluents,

the *domain* object additionally specifies the effects of actions, while the *yale* object encodes the problem instance including the time framework.[3]

$$
\begin{array}{rl}
\mathit{inertia}\ \{ & \mathit{alive}(T1) \leftarrow \mathit{alive}(T), \mathit{next}(T,T1) \\
& \neg\mathit{alive}(T1) \leftarrow \neg\mathit{alive}(T), \mathit{next}(T,T1) \\
& \mathit{loaded}(T1) \leftarrow \mathit{loaded}(T), \mathit{next}(T,T1) \\
& \neg\mathit{loaded}(T1) \leftarrow \neg\mathit{loaded}(T), \mathit{next}(T,T1)\} \\
\mathit{domain} : \mathit{inertia}\ \{ & \mathit{loaded}(T1) \leftarrow \mathit{load}(T), \mathit{next}(T,T1) \\
& \neg\mathit{loaded}(T1) \leftarrow \mathit{shoot}(T), \mathit{loaded}(T), \mathit{next}(T,T1) \\
& \neg\mathit{alive}(T1) \leftarrow \mathit{shoot}(T), \mathit{loaded}(T), \mathit{next}(T,T1)\} \\
\mathit{yale} : \mathit{domain}\ \{ & \mathit{load}(0) \leftarrow \quad \mathit{wait}(1) \leftarrow \quad \mathit{shoot}(2) \leftarrow \quad \mathit{alive}(0) \leftarrow \\
& \mathit{next}(0,1) \leftarrow \qquad \mathit{next}(1,2) \leftarrow \qquad \mathit{next}(2,3) \leftarrow\}
\end{array}
$$

The only answer set for this program contains, besides the facts of the *yale* object, $\mathit{loaded}(1)$, $\mathit{loaded}(2)$, $\mathit{alive}(0)$, $\mathit{alive}(1)$, $\mathit{alive}(2)$ and $\neg\mathit{loaded}(3)$, $\neg\mathit{alive}(3)$. That is, the individual is alive until the shoot action is taken, and no longer alive afterwards, and the gun is loaded between loading and shooting. ∎

We want to point out that this formalism is equally suited for solving problems which involve finding a plan (i.e. a sequence of actions), rather than determining the effects of a given plan as in the Yale Shooting Problem: You have to add a rule $\mathit{action}(T) \vee \neg\mathit{action}(T) \leftarrow \mathit{next}(T,T1)$ for every action, and you have to specify the goal state by a query, e.g. $\neg\mathit{alive}(3), \neg\mathit{loaded}(3)?$[4]. The complete DLP$^<$ encoding of a planning example is shown in the full version of this paper that can be retrieved from [12].

5 Computational Complexity

As for the classical nonmonotonic formalisms [20, 19, 25], two important decision problems, corresponding to two different reasoning tasks, arise in DLP$^<$:

(Brave Reasoning) Given a DLP$^<$ program $\mathcal{P}$ and a ground literal L, decide whether there exists an answer set M for $\mathcal{P}$ such that L is true w.r.t. M.

(Cautious Reasoning) Given a DLP$^<$ program $\mathcal{P}$ and a ground literal L, decide whether L is true in all answer sets for $\mathcal{P}$.

We next prove that the complexity of reasoning in DLP$^<$ is exactly the same as in traditional disjunctive logic programming. That is, inheritance comes for free, as the addition of inheritance does not cause any computational overhead. We consider the propositional case, i.e., we consider ground DLP$^<$ programs.

[3] With our prototype the more convenient builtins of `dlv` can be used instead.

[4] This is a dlv language feature which (for this example) is equivalent to the rules $h \leftarrow \neg\mathit{alive}(3), \neg\mathit{loaded}(3)$ and $i \leftarrow \mathit{not}\ h, \mathit{not}\ i$, meaning that only answer sets containing $\neg\mathit{alive}(3)$ and $\neg\mathit{loaded}(3)$ should be considered.

Lemma 5.1 *Given a ground* DLP$^<$ *program* $\mathcal{P}$ *and an interpretation* M *for* $\mathcal{P}$*, deciding whether* M *is an answer set for* $\mathcal{P}$ *is in* coNP*.*

Proof. We check in NP that M is not an answer set of $\mathcal{P}$ as follows. Guess a subset I of M, and verify that: (1) M is not a model for $pos(G_M(\mathcal{P}))$, or (2) I is a model for $pos(G_M(\mathcal{P}))$ and $I \subset M$. The construction of $pos(G_M(\mathcal{P}))$ (see Definition 3) is feasible in polynomial time, and the tasks (1) and (2) are clearly tractable. Thus, deciding whether M is not an answer set for $\mathcal{P}$ is in NP, and, consequently, deciding whether M is an answer set for $\mathcal{P}$ is in coNP. ∎

Theorem 5.2 *Brave Reasoning on* DLP$^<$ *programs is* Σ_2^P*-complete.*

Proof. Given a ground DLP$^<$ program $\mathcal{P}$ and a ground literal L, we verify that L is a brave consequence of $\mathcal{P}$ as follows. Guess a set $M \subseteq B_\mathcal{P}$ of ground literals, check that (1) M is an answer set for $\mathcal{P}$, and (2) L is true w.r.t. M. Task (2) is clearly polynomial; while (1) is in coNP, by virtue of Lemma 5.1. The problem therefore lies in Σ_2^P.

Σ_2^P-hardness follows from Theorem 6.1 and the results in [8, 9]. ∎

Theorem 5.3 *Cautious Reasoning on* DLP$^<$ *programs is* Π_2^P*-complete.*

Proof. Given a ground DLP$^<$ program $\mathcal{P}$ and a ground literal L, we verify that L is not a cautious consequence of $\mathcal{P}$ as follows. Guess a set $M \subseteq B_\mathcal{P}$ of ground literals, check that (1) M is an answer set for $\mathcal{P}$, and (2) L is not true w.r.t. M. Task (2) is clearly polynomial; while (1) is in coNP, by virtue of Lemma 5.1. Therefore, the complement of cautious reasoning is in Σ_2^P, and cautious reasoning is in Π_2^P.

Π_2^P-hardness follows from Theorem 6.1 and the results in [8, 9]. ∎

6 Related Work

6.1 Answer Set Semantics

Answer set semantics, proposed by Gelfond and Lifschitz in [13], is the most widely acknowledged semantics for disjunctive logic programs with strong negation. For this reason, while defining the semantics of our language, we took care of ensuring full agreement with answer set semantics (on inheritance-free programs).

Theorem 6.1 *Let* $\mathcal{P}$ *be a* DLP$^<$ *program consisting of a single object* $o = \langle oid(o), \Sigma(o) \rangle$*. Then,* M *is an answer set of* $\mathcal{P}$ *if and only if it is a consistent answer set of* $\Sigma(o)$ *(according with [13]).*

6.2 Disjunctive Ordered Logic

Disjunctive Ordered Logic ($\mathcal{DOL}$) is an extension of Disjunctive Logic Programming with strong negation and inheritance (without default negation) proposed in [7]. The DLP$^<$ language incorporates some ideas taken from $\mathcal{DOL}$. However, the two languages are very different in several respects. Most importantly, unlike with DLP$^<$, even if a program belongs to the common fragment of $\mathcal{DOL}$ and of the language of [13] (i.e., it contains neither inheritance nor default negation), $\mathcal{DOL}$ semantics is completely different from answer set semantics, because of a different way of handling contradictions.[5] In short, we observe the following differences between $\mathcal{DOL}$ and DLP$^<$:

- $\mathcal{DOL}$ does not include default negation *not*, while DLP$^<$ includes it.
- $\mathcal{DOL}$ and DLP$^<$ have different semantics on the common fragment. Consider a program $\mathcal{P}$ consisting of a single object $o = \langle oid(o), \Sigma(o)\rangle$, where $\Sigma(o) = \{p \leftarrow; \quad \neg p \leftarrow\}$. Then, according with $\mathcal{DOL}$, the semantics of $\mathcal{P}$ is given by two models, namely, $\{p\}$ and $\{\neg p\}$. On the contrary, $\mathcal{P}$ has no answer set according with DLP$^<$ semantics.
- DLP$^<$ generalizes (consistent) answer set semantics to disjunctive logic programs with inheritance; while $\mathcal{DOL}$ does not generalize it.

6.3 Prioritized Logic Programs

DLP$^<$ can be also seen as an attempt to handle priorities in disjunctive logic programs (the lower the object in the inheritance hierarchy, the higher the priority of its rules).

There are several works on preference handling in logic programming [6, 14, 23, 16, 24, 26]. However, we are aware of only one previous work on priorities in disjunctive programs, namely, the paper by Sakama and Inoue [26]. This interesting work can be seen as an extension of answer set semantics to deal with priorities. Comparing the two works, under the viewpoint of priority handling, we observe the following:

- On priority free programs, the two languages yield essentially the same semantics, as they generalize answer set semantics and consistent answer set semantics, respectively.
- In [26], priorities are defined among literals, while priorities concern program's rules in DLP$^<$.
- The different kind of priorities (on rules vs. literals) and the way how they are dealt with in the two approaches, imply different complexity in the respective reasonings. Indeed, from the simulation of abductive reasoning in the language of [26], and the complexity results on abduction derived in [10], it follows that brave reasoning is Σ_3^P-complete

[5] Actually, this was a main motivation for the authors to look for a different language.

for the language of [26]. On the contrary, brave reasoning is "only" Σ_2^P-complete in DLP$^<$[6].

A comparative analysis of the various approaches to the treatment of preferences in ($\vee$-free) logic programming has been carried out in [6].

6.4 Inheritance Networks

From a different perspective, the objects of a DLP$^<$ program can also be seen as the nodes of an inheritance network.

We next show that DLP$^<$ satisfies the basic semantic principles which are required for inheritance networks in [28].

[28] constitutes a fundamental attempt to present a formal mathematical theory of multiple inheritance with exceptions. The starting point of this work is the consideration that an intuitively acceptable semantics for inheritance must satisfy two basic requirements:

1. Being able to reason with redundant statements, and
2. not making unjustified choices in ambiguous situations.

Touretzky illustrates this intuition by means of two basic examples.

The former requirement is presented by means of the *Royal Elephant* example, in which we have the following knowledge: "Elephants are gray.", "Royal elephants are elephants.", "Royal elephants are not gray.", "Clyde is a royal elephant.", "Clyde is an elephant."

The last statement is clearly redundant; however, since it is consistent with the others there is no reason to rule it out. Touretzky shows that an intuitive semantics should be able to recognize that Clyde is not gray, while many systems fail in this task.

Touretzky's second principle is shown by the *Nixon diamond* example, , in which the following is known: "Republicans are not pacifists.", "Quakers are pacifists.", "Nixon is both a Republican and a quaker."

According to our approach, he claims that a good semantics should draw no conclusion about the question whether Nixon is a $pacifist$.

The proposed solution for the problems above is based on a topological relation, called *inferential distance ordering*, stating that an individual A is "nearer" to B than to C iff A has an inference path *through* B to C. If A is "nearer" to B than to C, then as far as A is concerned, information coming from B must be preferred w.r.t. information coming from C. Therefore, since Clyde is "nearer" to royal elephant than to elephant, he states that Clyde is not gray. On the contrary no conclusion is taken on Nixon, as there is not any relationship between *quaker* and *republican*.

The semantics of DLP$^<$ fully agrees with the intuition underlying the *inferential distance ordering*.

[6] We refer to the complexity in the propositional case here.

Example 8 Let us represent the Royal Elephant example in our framework:

$elephant$	$\{gray \leftarrow\}$
$royal_elephant : elephant$	$\{\neg gray \leftarrow\}$
$clyde : elephant,\ royal_elephant$	$\{\ \}$

The only answer set of the above DLP$^<$ program is $\{\neg gray\}$.

The Nixon Diamond example can be expressed in our language as follows:

$republican$	$\{pacifist \leftarrow\}$
$quaker$	$\{\neg pacifist \leftarrow\}$
$nixon : republican, quaker$	$\{\ \}$

This DLP$^<$ program has no answer set, and therefore no conclusion is drawn.

6.5 Updates in Logic Programs

The definition of the semantics of updates in logic programs is another topic where DLP$^<$ could potentially be applied. Roughly, a simple formulation of the problem is: given a ($\vee$-free) logic program P and a sequence $U_1, \cdots, U_n$ of successive updates (insertion/deletion of ground atoms), determine what is or is not true in the end. Expressing the insertion (deletion) of an atom A by the rule $A \leftarrow (\neg A \leftarrow)$, we can represent this problem by a DLP$^<$ knowledge base $\{\langle t_0, P\rangle, \langle t_1, \{U_1\}\rangle, \cdots, \langle t_n, \{U_n\}\rangle\}$ (t_i intuitively represents the instant of time when the update U_i has been executed), where $t_n < \cdots < t_0$. The answer sets of the program for t_k can be taken as the semantics of the execution of $U_1, \cdots, U_k$ on P. For instance, given the logic program $P = \{a \leftarrow b, not\ c\}$ and the updates $U_1 = \{b \leftarrow\}$, $U_2 = \{c \leftarrow\}$, $U_3 = \{\neg b \leftarrow\}$, we build the DLP$^<$ program

t_0	$\{\ a \leftarrow b, c, not\ d\ \}$
$t_1 : t_0$	$\{\ b \leftarrow\ \}$
$t_2 : t_1$	$\{\ c \leftarrow\ \}$
$t_3 : t_2$	$\{\ \neg b \leftarrow\ \}$.

The answer set $\{a, b, c\}$ of the program for t_2 gives the semantics of the execution of U_1 and U_2 on P; while the answer set $\{c\}$ of the program for t_3 expresses the semantics of the execution of U_1, U_2 and U_3 on P in the given order.

The semantics of updates obtained in this way is very similar to the approach adopted for the ULL language in [17]. Further investigations are needed on this topic to see whether DLP$^<$ can represent update problems in more general settings like those treated in [21] and in [4].

7 Implementation Issues

We have implemented a fully operational prototype supporting the DLP$^<$ language on top of the disjunctive logic programming system `dlv` [11]. To this end, we have defined a rewriting function π which translates a DLP$^<$

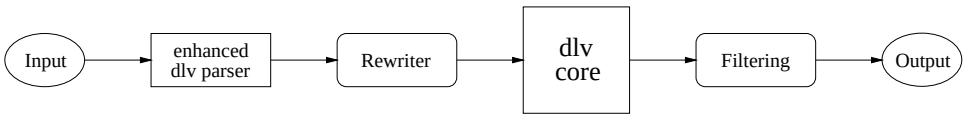

Figure 1: Flow diagram of the prototype.

program $\mathcal{P}$ in an equivalent plain disjunctive logic program $\pi(\mathcal{P})$ (with strong negation, but without inheritance), the (consistent) answer sets of $\pi(\mathcal{P})$ correspond to the answer sets of $\mathcal{P}$ and vice versa. For space limitation, we cannot show the rewriting technique here; full details on the translation algorithm along with the formal proof of its correctness are provided in the long version of the paper which can be retrieved from the web [12].

A schematic visualisation of its architecture is shown in Figure 1. First of all, we have extended the `dlv` parser to incorporate the DLP$^<$ syntax. In this way, all the advanced features of `dlv` (e.g. bounded integer arithmetics, comparison builtins, etc.) are also available with DLP$^<$. The *Rewriter* module implements the translation function π mentioned above. Once the rewritten version $\pi(\mathcal{P})$ of $\mathcal{P}$ is generated, its answer sets are then computed using the dlv core. Before the output is shown to the user, a suitable filtering function is applied in order to strip the internal predicates (which are added by the rewriting function π) from the output.

On the webpage [12] the prototype is described in detail, and binary versions are available for different platforms.

Acknowledgements

The idea of representing inertia in planning problems in a higher object is due to Axel Polleres. This work was partially supported by FWF (Austrian Science Funds) under the projects P11580-MAT and Z29-INF.

References

[1] Alferes, J., Pereira, L.M., On Logic Program Semantics with Two Kinds of Negation *Proc. of JICSLP 1992*, pp. 574-588

[2] Alferes, J., Pereira, L.M., Przymusinski, T.C., Strong and Explicit Negation in Non-Monotonic Reasoning and Logic Programming, *Proc. of JELIA 1996*, pp. 143-163

[3] Alferes, J., Pereira, L.M., Przymusinski, T.C., 'Classical' Negation in Nonmonotonic Reasoning and Logic Programming, *Journal of Automated Reasoning* 20(1), 1998, pp. 107-142.

[4] Alferes, J., Leite, J.A., Pereira, L.M., Przymusinska, H., Przymusinski, T.C., Dynamic Logic Programming, *Proc. KR'98*, Morgan Kaufmann, 1998.

[5] Baral, C., Gelfond, M. Logic Programming and Knowledge Representation *Journal of Logic Programming*, Vol. 19/20, May/July 1994, pp. 73–148.

[6] Brewka, G., Eiter, T., Preferred Answer Sets for Extended Logic Programs, *Proc. KR'98*, Morgan Kaufmann, 1998.

[7] Buccafurri, F., Leone, N., Rullo, P., Disjunctive Ordered Logic: Semantics and Expressiveness, *Proc. KR'98*, Morgan Kaufmann, 1998.

[8] Eiter, T., Gottlob, G., On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 1995.

[9] Eiter, T., Gottlob, G., and Mannila, H., Disjunctive Datalog, *ACM Transactions on Database Systems*, 22(3), September, 1997.

[10] Eiter, T., Gottlob, G., Leone, N., Abduction From Logic Programs: Semantics and Complexity. *Theoretical Computer Science* 189(1-2), pp. 129–177, 1997.

[11] Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F., The Knowledge Representation System `dlv`: Progress Report, Comparisons, and Benchmarks, *Proc. KR'98*, Morgan Kaufmann, 1998.

[12] Faber, W., The DLP$^<$ Homepage: `www.dbai.tuwien.ac.at/proj/dlv/inheritance/`

[13] Gelfond, M., and Lifschitz, V., Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365-385, 1991.

[14] Gelfond, M. and Son, T.C., Reasoning with Prioritized Defaults, *Proc. of LPKR'97*, Port Jefferson, New York, October 1997.

[15] Hanks, S., McDermott, D., Nonmonotonic Logic and Temporal Projection, *Artificial Intelligence*, 33:379–412, 1987.

[16] Kowalski, R.A., Sadri, F., Logic Programs with Exceptions, *Proc. of 7th ICLP*, Jerusalem, 1990, pp. 598-616.

[17] Leone, N., Palopoli, L., Romeo, M., A Language for Updating Logic Programs, *Journal of Logic Programming*, 23(1), 1995, pp. 1–61.

[18] Lobo, J., Minker, J., and Rajasekar, A., *Foundations of Disjunctive Logic Programming*. MIT Press, Cambridge, MA, 1992.

[19] Marek, W., Truszczyński, M., Modal logic for default reasoning, *Annals of Mathematics and Artificial Intelligence*, 1, 1990, pp. 275-302.

[20] Marek, W., Truszczyński, M., Autoepistemic Logic, *Journal of the ACM*, 38, 3, 1991, pp. 518-619.

[21] Marek, W., Truszczyński, M., Revisions Specifications by means of Programs, *Proc. JELIA'94*, Sringer, LNAI 838, 1994, pp. 122–136.

[22] McCarthy, J., Hayes, P. J., Some Philosophical Problems from the Standpoint of Artificial Intelligence, in *Machine Intelligence 4*, pp. 463–502, 1969.

[23] Nute, D., Defeasible Logic. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume III, pages 353–395. Clarendon Press, Oxford, 1994.

[24] Pradhan, S., and Minker, J., Combining Datalog Databases Using Priorities. *International Journal of Cooperative Intelligent Systems*, 1996.

[25] Reiter, R., A Logic for Default Reasoning, *Artificial Intelligence*, vol.13, pp. 81-132, 1980.

[26] Sakama, C., Inoue, K., Representing priorities in logic programs, *Proc. JICSLP'96* MIT Press, 1996, pp. 82–96.

[27] Shanahan, M., *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*, MIT Press, 1997.

[28] Touretzky, D.S., *The Mathematics of Inheritance Systems*, Pitman London, 1986.