Forgetting and Conflict Resolving in Disjunctive Logic Programming*

Thomas Eiter
Technische Universität Wien, Austria
eiter@kr.tuwien.ac.at

Kewen Wang†
Griffith University, Australia
k.wang@griffith.edu.au

Abstract

We establish a declarative theory of forgetting for disjunctive logic programs. The suitability of this theory is justified by a number of desirable properties. In particular, one of our results shows that our notion of forgetting is completely captured by the classical forgetting. A transformation-based algorithm is also developed for computing the result of forgetting. We also provide an analysis of computational complexity. As an application of our approach, a fairly general framework for resolving conflicts in inconsistent knowledge bases represented by disjunctive logic programs is defined. The basic idea of our framework is to weaken the preferences of each agent by forgetting certain knowledge that causes inconsistency. In particular, we show how to use the notion of forgetting to provide an elegant solution for preference elicitation in disjunctive logic programming.

Introduction

Forgetting (Lin & Reiter 1994; Lang, Liberatore, & Marquis 2003) is a key issue for adequately handle a range of classical tasks such as query answering, planning, decision-making, reasoning about actions, or knowledge update and revision. It is, moreover, also important in recently emerging issues such as design and engineering of Web-based ontology languages. Suppose we start to design an ontology of *Pets*, which is a knowledge base of various pets (like cats, dogs but not lions or tigers). Currently, there are numerous ontologies on the Web. We navigated the Web and found an ontology *Animals* which is a large ontology on various animals including cats, dogs, tigers and lions. It is not a good idea to download the whole ontology Animals. The approach in the current Web ontology language standard OWL[1] is to discard those terminologies that are not desired (although this function is still very limited in OWL). For example, we may discard (or forget) tigers and lions from the ontology *Animals*. If our ontology is only a list of relations, we can handle the forgetting (or discarding) easily. However, an ontology is often represented as a logical theory, and the removal of one term may influence other terms in the ontology. Thus, more advanced methods are needed.

Disjunctive logic programming (DLP) under the answer set semantics (Gelfond & Lifschitz 1990) is now widely accepted as a major tool for knowledge representation and commonsense reasoning (Baral 2002). DLP is expressive in that it allows disjunction in rule heads, negation as failure in rule bodies and strong negation in both heads and bodies. Studying forgetting within DLP is thus a natural issue, and we make in this paper the following contributions:

- We establish a declarative, semantically defined notion of forgetting for disjunctive logic programs, which is a generalization of the corresponding notion for nondisjunctive programs proposed in (Wang, Sattar, & Su 2005). The suitability of this theory is justified by a number of desirable properties.
- We present a transformation-based algorithm for computing the result of forgetting. This method allows to obtain the result of forgetting a literal l in a logic program via a series of program transformations and other rewritings. In particular, for any disjunctive program P and any literal l, a syntactic representation $\mathsf{forget}(P, l)$ for forgetting l in P always exists. The transformation is novel and does not extend a previous one in (Wang, Sattar, & Su 2005), which as we show is incomplete.
- Connected with the transformation algorithm, we settle some complexity issues for reasoning under forgetting. They provide useful insight into feasible representations of forgetting.
- As an application of our approach, we present a fairly general framework for resolving conflicts in inconsistent knowledge bases. The basic idea of this framework is to weaken the preferences of each agent by forgetting certain knowledge that causes inconsistency. In particular, we show how to use the notion of forgetting to provide an elegant solution for preference elicitation in DLP.

Preliminaries

We briefly review some basic definitions and notation used throughout this paper.

A *disjunctive program* is a finite set of rules of the form

$$a_1 \vee \cdots \vee a_s \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n, \qquad (1)$$

*This work was partially supported by the Austrian Science Funds (FWF) projects P17212 and 18019, the European Commission project REWERSE (IST-2003-506779) and the Australia Research Council (ARC) Discovery Project 0666107.

†Part of the work was done while this author was working at Technische Universität Wien.

[1] http://www.w3.org/2004/OWL/

$s, m, n \geq 0$, where a, b's and c's are classical literals in a propositional language. A *literal* is a *positive literal* p or a *negative literal* $\neg p$ for some atom p. An *NAF*-literal is of the form $not\ l$ where not is for the negation as failure and l is a (ordinary) literal. For an atom p, p and $\neg p$ are called *complementary*. For any literal l, its complementary literal is denoted $\tilde{l}$.

To guarantee the termination of some program transformations, the body of a rule is a set of literals rather than a multiset.

Given a rule r of form (1), $head(r) = a_1 \vee \cdots \vee a_s$ and $body(r) = body^+(r) \cup not\ \ body^-(r)$ where $body^+(r) = \{b_1, \ldots, b_m\}$, $body^-(r) = \{c_1, \ldots, c_n\}$, and $not\ body^-(r) = \{not\ q \mid q \in body^-(r)\}$.

A rule r of the form (1) is *normal* or *non-disjunctive*, if $s \leq 1$; *positive*, if $n = 0$; *negative*, if $m = 0$; *constraint*, if $s = 0$; *fact*, if $m = 0$ and $n = 0$, in particular, a rule with $s = n = m = 0$ is the constant *false*.

A disjunctive program P is called *normal program* (resp. *positive program*, *negative program*), if every rule in P is normal (resp. positive, negative).

Let P be a disjunctive program and let X be a set of literals. A disjunction $a_1 \vee \cdots \vee a_s$ is satisfied by X, denoted $X \models a_1 \vee \cdots \vee a_s$ if $a_i \in X$ for some i with $1 \leq i \leq s$. A rule r in P is satisfied by X, denoted $X \models r$, iff "$body^+(r) \subseteq X$ and $body^-(r) \cap X = \emptyset$ imply $X \models head(r)$". X is a model of P, denoted $X \models P$ if every rule of P is satisfied by X.

An *interpretation* X is a set of literals that contains no pair of complementary literals.

The answer set semantics The *reduct* of P on X is defined as $P^X = \{head(r) \leftarrow body^+(r) \mid r \in P, body^-(r) \cap X = \emptyset\}$. An interpretation X is an *answer set* of P if X is a minimal model of P^X (by treating each literal as a new atom). $\mathcal{AS}(P)$ denotes the collection of all answer sets of P. P is *consistent* if it has at least one answer set.

Two disjunctive programs P and P' are *equivalent*, denoted $P \equiv P'$, if $\mathcal{AS}(P) = \mathcal{AS}(P')$.

As usual, B_P is the *Herbrand base* of logic program P, that is, the set of all (ground) literals in P.

Forgetting in Logic Programming

In this section, we want to define what it means to forget about a literal l in a disjunctive program P. The idea is to obtain a logic program which is equivalent to the original disjunctive program, if we ignore the existence of the literal l. We believe that forgetting should go beyond syntactic removal of rules/literals and be close to classical forgetting and answer set semantics (keeping its spirit) at the same time. Thus, the definition of forgetting in this section is given in semantics terms, i.e., based on answer sets, and naturally generalizes the corresponding one in (Wang, Sattar, & Su 2005).

In propositional logic, the result of forgetting $\mathsf{forget}(T, p)$ about a proposition p in a theory T is conveniently defined as $T(p/true) \vee T(p/false)$. This way cannot be directly generalized to logic programming since there is no notion of the "disjunction" of two logic programs. However, if we examine the classical forgetting in model-theoretic point of view, we can obtain the models of $\mathsf{forget}(T, p)$ in this way: first compute all models of T and remove p from each model if it contains p. The resulting collection of sets $\{M \setminus \{p\} \mid M \models T\}$ is exactly the set of all models of $\mathsf{forget}(T, p)$.

Similarly, given a consistent disjunctive program P and a literal l, we naively could define the result of forgetting about l in P as an extended disjunctive program P' whose answer sets are exactly $\mathcal{AS}(P) \setminus l = \{X \setminus \{l\} \mid X \in \mathcal{AS}(P)\}$. However, this notion of forgetting cannot guarantee the existence of P' for even simple programs. For example, consider $P = \{a \leftarrow .\ \ p \vee q \leftarrow\}$, then $\mathcal{AS}(P) = \{\{a, p\}, \{a, q\}\}$ and thus $\mathcal{AS}(P) \setminus p = \{\{a\}, \{a, q\}\}$. Since $\{a\} \subset \{a, q\}$ and, as well-known, answer sets are incomparable under set inclusion, $\mathcal{AS}(P) \setminus p$ cannot be the set of answer sets of any disjunctive program.

A solution to this problem is *a suitable notion of minimal answer set* such that the definition of answer sets, minimality, and forgetting can be fruitfully combined. To this end, we call a set X' an *l-subset of a set X*, denoted $X' \subseteq_l X$, if $X' \setminus \{l\} \subseteq X \setminus \{l\}$. Similarly, a set X' is a strict l-subset of X, denoted $X' \subset_l X$, if $X' \setminus \{l\} \subset X \setminus \{l\}$. Two sets X and X' of literals are l-equivalent, denoted $X \sim_l X'$, if $(X \setminus X') \cup (X' \setminus X) \subseteq \{l\}$.

Definition 1 *Let P be a consistent disjunctive program, let l be a literal in P and let X be a set of literals.*

1. *For a collection $\mathcal{S}$ of sets of literals, $X \in \mathcal{S}$ is* l-minimal *if there is no $X' \in \mathcal{S}$ such that $X' \subset_l X$. $\min_l(\mathcal{S})$ denotes the collection of all l-minimal elements in $\mathcal{S}$.*
2. *An answer set X of disjunctive program P is an* l-answer set *if X is l-minimal in $\mathcal{AS}(P)$. $\mathcal{AS}_l(P)$ consists of all l-answer sets of P.*

To make $\mathcal{AS}(P) \setminus l$ incomparable, we could take either minimal elements or maximal elements from $\mathcal{AS}(P) \setminus l$. However, selecting minimal answer sets is in line with semantic principles to minimize positive information.

For example, $P = \{a \leftarrow .\ p \vee q \leftarrow\}$, has two answer sets $X = \{a, p\}$ and $X' = \{a, q\}$. X is a p-answer set of P, but X' is not. This example shows that, for a disjunctive program P and a literal l, not every answer set is an l-answer set.

In the rest of this paper, we assume that P is a consistent program. The following proposition collects some easy properties of l-answer sets.

Proposition 1 *For any consistent program P and a literal l in P, the following four items are true:*

1. *An l-answer set X of P must be an answer set of P.*
2. *For any answer set X of P, there is an l-answer set X' of P such that $X' \subseteq_l X$.*
3. *Any answer set X of P with $l \in X$ is an l-answer set of P.*
4. *If an answer set X of P is not an l-answer set, then (1) $l \notin X$; (2) there exists an l-answer set Y of P such that $l \in Y \subset_l X$.*

Having the notion of minimality about forgetting a literal, we are now in a position to define the result of forgetting about a literal in a disjunctive program.

Definition 2 *Let P be a consistent disjunctive program and l be a literal. A disjunctive program P' is a result of* forgetting about l in P, *if P' represents l-answer sets of P, i.e., the following conditions are satisfied:*

1. *$B_{P'} \subseteq B_P \setminus \{l\}$ and*
2. *For any set X' of literals with $l \notin X'$, X' is an answer set of P' iff there is an l-answer set X of P such that $X' \sim_l X$.*

Notice that the first condition implies that l does not appear in P'. An important difference of the notion of forgetting here from existing approaches to updating and merging logic programs is that only l and possibly some other literals are removed. In particular, no new symbol is introduced in P'.

For a consistent extended program P and a literal l, some program P' as in the above definition always exists (cf. Algorithm 1 for details). However, different such programs P' might exist. It follows from the above definition that they are all equivalent under the answer set semantics.

Proposition 2 *Let P be a disjunctive program and l a literal in P. If P' and P'' are two results of forgetting about l in P, then P' and P'' are equivalent.*

We use $\mathsf{forget}(P, l)$ to denote a possible result of forgetting about l in P.

Example 1 *1. If $P_1 = \{q \leftarrow not\ p\}$, then $\mathsf{forget}(P_1, q) = \emptyset$ and $\mathsf{forget}(P_1, p) = \{q \leftarrow\}$.*

2. *If $P_2 = \{p \vee q \leftarrow\}$, then $\mathsf{forget}(P_2, p) = \emptyset$.*
3. *$P_3 = \{p \vee q \leftarrow not\ p.\ c \leftarrow q\}$ has the unique answer set $\{q, c\}$ and $\mathsf{forget}(P_3, p) = \{q \leftarrow .\ c \leftarrow\}$.*
4. *$P_4 = \{a \vee p \leftarrow not\ b.\ c \leftarrow not\ p.\ b \leftarrow\}$. Then $\mathsf{forget}(P_4, p) = \{c \leftarrow .\ b \leftarrow\}$.*

We will explain how to obtain $\mathsf{forget}(P, l)$ in the next section. The following proposition generalizes Proposition 2.

Proposition 3 *Let P and P' be two equivalent disjunctive programs and l a literal in P. Then $\mathsf{forget}(P, l)$ and $\mathsf{forget}(P', l)$ are also equivalent.*

However, forgetting here does not preserve some special equivalences of logic programs stronger than ordinary equivalence like strong equivalence (Lifschitz, Tang, & Turner 1999) or uniform equivalence (Eiter & Fink 2003). A notion of forgetting which preserves strong equivalence is interesting for some applications, but beyond the scope of this paper. In addition, our approach may be easily refined to preserve equivalences stronger than ordinary equivalences by a canonical form for the result of forgetting (e.g., the output of Algorithm 1).

Proposition 4 *For any consistent program P and a literal l in P, the following items are true:*

1. *$\mathcal{AS}(\mathsf{forget}(P, l)) = \{X \setminus \{l\} \mid X \in \mathcal{AS}_l(X)\}$.*
2. *If $X \in \mathcal{AS}_l(X)$ with $l \notin X$, then $X \in \mathcal{AS}(\mathsf{forget}(P, l))$.*
3. *For any $X \in \mathcal{AS}(P)$ such that $l \in X$, $X \setminus \{l\} \in \mathcal{AS}(\mathsf{forget}(P, l))$.*
4. *For any $X' \in \mathcal{AS}(\mathsf{forget}(P, l))$, either X' or $X' \cup \{l\}$ is in $\mathcal{AS}(P)$.*
5. *For any $X \in \mathcal{AS}(P)$, there exists $X' \in \mathcal{AS}(\mathsf{forget}(P, l))$ such that $X' \subseteq X$.*
6. *If l does not appear in P, then $\mathsf{forget}(P, l) = P$.*

Let $\models_s$ and $\models_c$ be the skeptical and credulous reasoning defined by the answer sets of a disjunctive program P, respectively: for any literal l,

$P \models_s l$ iff $l \in S$ for every $S \in \mathcal{AS}(P)$.
$P \models_c l$ iff $l \in S$ for some $S \in \mathcal{AS}(P)$.

Proposition 5 *Let l be a specified literal in disjunctive program P. For any literal $l' \neq l$,*

1. *$P \models_s l'$ iff $\mathsf{forget}(P, l) \models_s l'$.*
2. *$P \models_c l'$ if $\mathsf{forget}(P, l) \models_c l'$.*

This proposition says that, if l is ignored, $\mathsf{forget}(P, l)$ is equivalent to P under skeptical reasoning, but weaker under credulous reasoning (i.e., inferences are lost).

Similar to the case of normal programs, the above definitions of forgetting about a literal l can be extended to forgetting about a set F of literals. Specifically, we can similarly define $X_1 \subseteq_F X_2$, $X_1 \sim_F X_2$ and F-answer sets of a disjunctive program. The properties of forgetting about a single literal can also be generalized to the case of forgetting about *a set*. Moreover, the result of forgetting about a set F can be obtained one by one forgetting each literal in F.

Proposition 6 *Let P be a consistent disjunctive program and $F = \{l_1, \ldots, l_m\}$ be a set of literals. Then*

$$\mathsf{forget}(P, F) \equiv \mathsf{forget}(\mathsf{forget}(\mathsf{forget}(P, l_1), l_2), \ldots), l_m).$$

We remark that for removing a proposition p entirely from a program P, it is suggestive to remove both the literals p and $\neg p$ in P (i.e., all positive and negative information about p). This can be easily accomplished by $\mathsf{forget}(P, \{p, \neg p\})$.

Let $\mathsf{lcomp}(P)$ be Clark's completion plus the loop formulas for an ordinary disjunctive program P (Lee & Lifschitz 2003; Lin & Zhao 2004). Then X is an answer set of P iff X is a model of $\mathsf{lcomp}(P)$.

Now we have two kinds of operators $\mathsf{forget}(,)$ and $\mathsf{lcomp}()$. Thus for a disjunctive program and an atom p, we have two classical logical theories $\mathsf{lcomp}(\mathsf{forget}(P, p))$ and $\mathsf{forget}(\mathsf{lcomp}(P), p)$ on the signature $B_P \setminus \{p\}$. It is natural to ask what the relationship between these two theories is. Intuitively, the models of the first theory are all minimal models while the models of the second theory may not be minimal [2]. Let $P = \{p \leftarrow not\ q.\ q \leftarrow not\ p\}$. Then $\mathsf{lcomp}(\mathsf{forget}(P, p)) = \{\neg q\}$ and $\mathsf{forget}(\mathsf{lcomp}(P), p) = \{\mathsf{T} \leftrightarrow \neg q \vee \mathsf{F} \leftrightarrow \neg q\} \equiv \mathsf{T}$, which has two models $\{q\}$ and $\emptyset$.

However, we have the following result. However, notice that for this program P, the *minimal models* of $\mathsf{forget}(\mathsf{lcomp}(P), p)$ are the same as the *models* of $\mathsf{lcomp}(\mathsf{forget}(P, p))$. In fact, this result is true for ordinary disjunctive programs in general.

Theorem 1 *Let P be a logic program without strong negation and p an atom in P. Then X is an answer set of*

[2]Thanks to Esra Erdem and Paolo Ferraris for pointing this out to us.

$\mathsf{forget}(P, p)$ *if and only if* X *is a minimal model of the result of classical forgetting* $\mathsf{forget}(\mathsf{lcomp}(P), p)$. *That is,*

$$\mathcal{AS}(\mathsf{forget}(P, p)) = \mathsf{MMod}(\mathsf{forget}(\mathsf{lcomp}(P), p))$$

Here $\mathsf{MMod}(T)$ *denotes the set of all minimal models of a theory* T *in classical logic.*

This result means that the answer sets of $\mathsf{forget}(P, p)$ are exactly the minimal models of the result of forgetting about p in the classical theory $\mathsf{lcomp}(P)$. Thus $\mathsf{forget}(P, p)$ can be characterized by forgetting in classical logic. Notice that it would not make much sense if we replace $\mathsf{lcomp}(P)$ with a classical theory which is not equivalent to $\mathsf{lcomp}(P)$ in Theorem 1. In this sense, the notion of forgetting for answer set programming is unique.

We use $\mathsf{forget}_{min}(T, p)$ to denote a set of classical formulas whose models are the minimal models of the classical forgetting $\mathsf{forget}(T, p)$. Then the conclusion of Theorem 1 is reformulated as $\mathsf{lcomp}(\mathsf{forget}(P, p)) \equiv \mathsf{forget}_{min}(\mathsf{lcomp}(P), p)$.

The result is a nice property, since it means that one can "bypass" the use of an LP engine entirely, and represent also the answer sets of $\mathsf{forget}(P, p)$ in terms of a circumscription of classical forgetting, applied to $\mathsf{lcomp}(P)$. In fact, we can express combined forgetting and minimal model reasoning by a circumscription of $\mathsf{lcomp}(P)$.

Theorem 2 *Let* P *be a logic program without strong negation and* p *an atom in* P. *Then* S' *is an answer set of* $\mathsf{forget}(P, p)$ *if and only if either* $S = S'$ *or* $S = S' \cup \{p\}$ *is a model of* $\mathsf{Circ}(B_P \setminus \{p\}, \{p\}, \mathsf{lcomp}(P))$.

Computation of Forgetting

As we have noted, $\mathsf{forget}(P, l)$ exists for any consistent disjunctive program P and literal l. In this section, we discuss some issues on computing the result of forgetting.

Naive Algorithm

By Definition 2, we can easily obtain a naive algorithm for computing $\mathsf{forget}(P, l)$ using some ASP solvers for DLP, like DLV (Leone *et al.* 2004) or GnT (Janhunen *et al.* 2000).

Algorithm 1 (Computing a result of forgetting)
Input: disjunctive program P and a literal l in P.
Procedure:

Step 1. Using DLV compute $\mathcal{AS}(P)$;

Step 2. Remove the literal l from every element of $\mathcal{AS}(P)$ and denote the resulting collection as A'

Step 3. Obtain A'' by removing non-minimal elements from A'.

Step 4. Construct P' whose answer sets are exactly A'': Let $A'' = \{A_1, ..., A_m\}$ and for each A_i, $P_i = \{l' \leftarrow not\ \bar{A}_i \mid l' \in A_i\}$. $P' = \cup_{1 \le i \le n} P_i$. Here $\bar{A}_i = B_P \setminus A_i$.

Step 5. Output P' as $\mathsf{forget}(P, l)$.

This algorithm is complete w.r.t. the semantic forgetting defined in Definition 2.

Theorem 3 *For any consistent disjunctive program* P *and a literal* l, *Algorithm 1 always outputs* $\mathsf{forget}(P, l)$.

Basic Program Transformations

The above algorithm is semantic, and does not describe how to syntactically compute the result of forgetting in DLP. In this subsection, we develop an algorithm for computing the result of forgetting in P using program transformations and other modifications. Here we use the set $\mathbf{T}^*_{\mathsf{WFS}}$ of program transformations investigated in (Brass & Dix 1999; Wang & Zhou 2005). In our algorithm, an input program P is first translated into a negative program and the result of forgetting is represented as a nested program (under the minimal answer sets defined by Lifschitz et al. (1999)).

Elimination of Tautologies: P' is obtained from P by the elimination of tautologies if there is a rule r: $head(r) \leftarrow body^+(r), not\ \ body^-(r)$ in P such that $head(r) \cap body^+(r) \neq \emptyset$ and $P' = P \setminus \{r\}$.

Elimination of Head Redundancy P' is obtained from P by the elimination of head redundancy if there is a rule r in P such that an atom a is in both $head(r)$ and $body^-(r)$ and $P' = P \setminus \{r\} \cup \{head(r) - a \leftarrow body(r)\}$.

The above two transformations guarantee that those rules whose head and body have common literals are removed.

Positive Reduction: P' is obtained from P by positive reduction if there is a rule r: $head(r) \leftarrow body^+(r), not\ \ body^-(r)$ in P and $c \in body^-(r)$ such that $c \notin head(P)$ and P' is obtained from P by removing $not\ \ c$ from r. That is, $P' = P \setminus \{r\} \cup \{head(r) \leftarrow body^+(r), not\ (body^-(r) \setminus \{c\})\}$.

Negative Reduction: P' is obtained from P by negative reduction if there are two rules r: $head(r) \leftarrow body^+(r), not\ body^-(r)$ and r': $head(r') \leftarrow$ in P such that $head(r') \subseteq body^-(r)$ and $P' = P \setminus \{r\}$.

To define our next program transformation, we need the notion of *s-implication* of rules. This is a strengthened version of the notion of *implications* in (Brass & Dix 1999).

Definition 3 *Let* r *and* r' *be two rules. We say that* r' *is an s-implication of* r *if* $r' \neq r$ *and at least one of the following two conditions is satisfied:*

1. r' *is an implication of* r: $head(r) \subseteq head(r')$, $body(r) \subseteq body(r')$ *and at least one inclusion is proper; or*
2. r *can be obtained by changing some negative body literals of* r' *into head atoms and removing some head atoms and body literals from* r' *if necessary.*

Elimination of s-Implications: P_2 is obtained from P_1 by elimination of s-implications if there are two distinct rules r and r' of P_1 such that r' is an s-implication of r and $P_2 = P_1 \setminus \{r'\}$.

Unfolding: P' is obtained from P by unfolding if there is a rule r such that

$$\begin{aligned} P' \quad = \quad & P \setminus \{r\} \cup \{head(r) \vee (head(r') - b) \leftarrow \\ & (body^+(r) \setminus \{b\}), not\ body^-(r), body(r')) \mid \\ & b \in body^+(r), \exists r' \in P \text{ s.t. } b \in head(r')\}. \end{aligned}$$

Here $head(r') - b$ is the disjunction obtained from $head(r')$ by removing b.

Since an implication is always an s-implication, the following result is a direct corollary of Theorem 4.1 in (Brass & Dix 1999).

Lemma 1 *Each disjunctive program P can be equivalently transformed into a negative program N via the program transformations in $\mathbf{T}^*_{\mathsf{WFS}}$, such that on no rule r in N, a literal appears in both the head and the body of r.*

Transformation-Based Algorithm

We are now in a position to present our syntax-based algorithm for computing forgetting in a disjunctive program.

Algorithm 2 (Computing a result of forgetting)
Input: disjunctive program P and a literal l in P.
Procedure:

Step 1. Fully apply the program transformations in $\mathbf{T}^*_{\mathsf{WFS}}$ on program P and then obtain a negative program N_0.

Step 2. Separate l from head disjunction via semi-shifting: For each (negative) rule $r \in N_0$ such that $head(r) = l \vee A$ and A is a non-empty disjunction, it is replaced by two rules: $l \leftarrow not\ A, body(r)$ and $A \leftarrow not\ l, body(r)$. Here $not\ A$ is the conjunction of all $not\ l'$ with l' in A. The resulting disjunctive program is denoted N.

Step 3. Suppose that N has n rules with head l:

$r_j : l \leftarrow not\ l_{j1}, ..., not\ l_{jm_j}$ where $n \geq 0, j = 1, \ldots, n$ and $m_j \geq 0$ for all j.

If $n = 0$, then let Q denote the program obtained from N by removing all appearances of $not\ l$.

If $n = 1$ and $m_1 = 0$, then $l \leftarrow$ is the only rule in N having head l. In this case, remove every rule in N whose body contains $not\ l$. Let Q be the resulting program.

For $n \geq 1$ and $m_1 > 0$, let $D_1, \ldots, D_s$ be all possible conjunctions $(not\ not\ l_{1k_1}, \cdots, not\ not\ l_{nk_n})$ where $0 \leq k_1 \leq m_1$, ..., $0 \leq k_n \leq m_n$. Replace in N each occurrence of $not\ l$ in N by all possible D_i. Let Q be the result.

Step 4. Remove all rules with head l from Q and output the resulting program N'.

Some remarks: (1) This is only a general algorithm. Some program transformations could be omitted for some special programs and various heuristics could also be employed to make the algorithm more efficient; (2) In this process, a result of forgetting is represented by a logic program allowing nested negation as failure. This form seems more intuitive than using ordinary logic programs; (3) In the construction of D_i, $not\ not\ l_{ij}$ cannot be replaced with l_{ij} (even for a normal logic program). As one can see, if they are replaced, the resulting program represents only a subset of $\mathcal{AS}_l(P)$ (see Example 2). This also implies that Algorithm 1 in (Wang, Sattar, & Su 2005) is incomplete in general. (4) Algorithm 2 above essentially improves the corresponding algorithm (Algorithm 1) in (Wang, Sattar, & Su 2005) at least in two ways: (i) our algorithm works for a more expressive class of programs (i.e. disjunctive programs) and (ii) the next result shows that our algorithm is complete under the minimal answer set semantics of nested logic programs.

Theorem 4 *Let P be a consistent disjunctive program and l a literal. Then X is an answer set of* $\mathsf{forget}(P, l)$ *iff X is a minimal answer set of N'.*

Example 2 *Consider $P_4 = \{c \leftarrow not\ q.\ p \leftarrow not\ q.\ q \leftarrow not\ p\}$. Then, by Algorithm 2,* $\mathsf{forget}(P_4, p)$ *is the nested program $\{c \leftarrow not\ q.\ q \leftarrow not\ not\ q\}$, whose minimal answer sets are exactly the same as the answer sets of* $\mathsf{forget}(P_4, p)$*. Note that Algorithm 1 in (Wang, Sattar, & Su 2005) outputs a program $N' = \{c \leftarrow not\ q.\ q \leftarrow q\}$ which has a unique answer set $\{c\}$. However,* $\mathsf{forget}(P_4, p)$ *has two answer sets $\{c\}$ and $\{q\}$. This implies that the algorithm there is incomplete.*

The above algorithm is worst case exponential, and might also output an exponentially large program. As follows from complexity considerations, there is no program P' that represents the result of forgetting which can be constructed in polynomial time, even if auxiliary literals might be used which are projected from the answer sets of P'. This is a consequence of the complexity results below.

However, the number of rules containing l may not be very large and some conjunctions D_i may be omitted because of redundancy. Moreover, the splitting technique of logic programs (**?**) can be used to localize the computation of forgetting. That is, an input program P is split into two parts so that the part irrelevant to forgetting is separated from the process of forgetting.

Resolving Conflicts in Multi-Agent Systems

In this section, we present a general framework for resolving conflicts in multi-agents systems, which is inspired from the *preference recovery* problem (Lang & Marquis 2002). Suppose that there are n agents who may have different preferences on the same issue. In many cases, these preferences (or constraints) have conflicts and thus cannot be satisfied at the same time. It is an important issue in constraint reasoning to find an intuitive criteria so that preferences with higher priorities are satisfied. Consider the following example.

Example 3 *(Lang & Marquis 2002) Suppose that a group of four residents in a complex tries to reach an agreement on building a* swimming pool *and/or a* tennis court*. The preferences and constraints are as follows.*

1. *Building a tennis court or a swimming pool costs each one unit of money.*
2. *A swimming pool can be either* red *or* blue.
3. *The first resident would not like to spend more than one money unit, and prefers a red swimming pool.*
4. *The second resident would like to build at least one of tennis court and swimming pool. If a swimming pool is built, he would prefer a blue one.*
5. *The third resident would prefer a swimming pool but either colour is fine with him.*
6. *The fourth resident would like both tennis court and swimming pool to be built. He does not care about the colour of the pool.*

Obviously, the preferences of the group are jointly inconsistent and thus it is impossible to satisfy them at the same time.

In the following, we will show how to resolve this kind of preference conflicts using the theory of forgetting.

An *n-agent system* $\mathcal{S}$ is an n-tuple $(P_1, P_2, \ldots, P_n)$ of disjunctive programs, $n > 0$, where P_i represents agent i's knowledge (including preferences, constraints).

As shown in Example 3, $P_1 \cup P_2 \cup \cdots \cup P_n$ may be inconsistent. The basic idea in our approach is to forget some literals for each agent so that conflicts can be resolved.

Definition 4 *Let $\mathcal{S} = (P_1, P_2, \ldots, P_n)$ be an n-agent system. A* compromise *of $\mathcal{S}$ is a sequence $C = (F_1, F_2, \ldots, F_n)$ where each F_i is a set of literals. An* agreement *of $\mathcal{S}$ on C is an answer set of the disjunctive program* $\mathsf{forget}(\mathcal{S}, C)$ *where* $\mathsf{forget}(\mathcal{S}, C) = \mathsf{forget}(P_1, F_1) \cup \mathsf{forget}(P_2, F_2) \cup \cdots \cup \mathsf{forget}(P_n, F_n)$.

For a specific application, we may need to impose certain conditions on each F_i.

Example 4 *(Example 3 continued) The scenario can be encoded as a collection of five disjunctive programs (P_0 stands for general constraints): $\mathcal{S} = (P_0, P_1, P_2, P_3, P_4)$ where*

$$\begin{aligned}
P_0 = \{ & \mathit{red} \vee \mathit{blue} \leftarrow s. \quad \leftarrow \mathit{red}, \mathit{blue}. \\
& u_1 \leftarrow \mathit{not}\ s, t. \quad u_1 \leftarrow s, \mathit{not}\ t. \\
& u_2 \leftarrow s, t. \quad u_0 \leftarrow \mathit{not}\ s, \mathit{not}\ t\}; \\
P_1 = \{ & u_0 \vee u_1 \leftarrow . \quad \mathit{red} \leftarrow s\}; \\
P_2 = \{ & s \vee t \leftarrow . \quad \mathit{blue} \leftarrow s\}; \\
P_3 = \{ & s \leftarrow\}; \textit{ and } P_4 = \{s \leftarrow . \ t \leftarrow\}.
\end{aligned}$$

Since this knowledge base is jointly inconsistent, each resident may have to weaken some of her preferences so that an agreement is reached. Some possible compromises are:

1. *$C_1 = (\emptyset, F, F, F, F)$ where $F = \{s, \mathit{blue}, \mathit{red}\}$: Every resident would be willing to weaken her preferences on the swimming pool and its colour. Since* $\mathsf{forget}(\mathcal{S}, C_1) = P_0 \cup \{u_0 \vee u_1 \leftarrow . \ t \leftarrow\}$, *$\mathcal{S}$ has a unique agreement $\{t, u_1\}$ on C_1. That is, only a tennis court is built.*
2. *$C_2 = (\emptyset, F, F, F, F)$ where $F = \{u_0, u_1, u_2, \mathit{blue}, \mathit{red}\}$: Every resident can weaken her preferences on the price and the pool colour. Since* $\mathsf{forget}(\mathcal{S}, C_2) = P_0 \cup \{s \vee t \leftarrow . \ s \leftarrow . \ t \leftarrow\}$, *$\mathcal{S}$ has two possible agreements $\{s, t, \mathit{red}\}$ and $\{s, t, \mathit{blue}\}$ on C_2. That is, both a tennis court and a swimming pool will be built but the pool colour can be either red or blue.*
3. *$C_3 = (\emptyset, \{\mathit{blue}, \mathit{red}\}, \emptyset, \emptyset, \{t\})$: The first resident can weaken her preference on pool colour and the fourth resident can weaken her preference on tennis court. Since* $\mathsf{forget}(\mathcal{S}, C_3) = P_0 \cup P_2 \cup P_3 \cup \{u_0 \vee u_1 \leftarrow . \ s \vee t \leftarrow . \ s \leftarrow\}$, *$\mathcal{S}$ has a unique agreement $\{s, \mathit{blue}, u_1\}$ on C_3. That is, only a swimming pool will be built and its colour is blue.*

As shown in the example, different compromises lead to different results. We do not consider the issue of how to reach compromises here, which is left for future work.

Computational Complexity

In this section we address the computational complexity of forgetting for different classes of logic programs. Our results show that for general disjunctive programs, (1) the model checking of forgetting is Π_2^p-complete; (2) the credulous reasoning of forgetting is Σ_3^p-complete. However, for normal programs or negative disjunctive programs, the complexity levels are lower: (1) the model checking of forgetting is co-NP-complete; (2) the credulous reasoning of forgetting is Σ_2^p-complete. The design of Algorithm 2 in Section is heavily based on the complexity analysis here. Our complexity results for forgetting are summarized in the following table and formally stated after the table.

	disjunctive	negative	normal
model checking	Π_2^p	co-NP	co-NP
$\models_c$	Σ_3^p	Σ_2^p	Σ_2^p

Theorem 5 *Given a disjunctive program P, a literal l, and a set of literals X, deciding whether X is an l-answer set of P is Π_2^p-complete.*

Intuitively, in order to show that X is an l-answer set, we have to witness that X is an answer set (which is coNP-complete to test), and that there is no answer set X' of P such that $X' \subset_l X$. Any X' disproving this can be guessed and checked using an NP-oracle in polynomial time. Thus, l-answer set checking is in Π_2^p, as stated in Theorem 5.

Proof (Sketch) Π_2^p membership holds since checking whether a set of literals X' is an answer set of a disjunctive program P is in co-NP. The hardness result is shown by a reduction from deciding whether a given disjunctive program P (without strong negations) has no answer set, which is Π_2^p-complete (Eiter & Gottlob 1995). Construct a logic program $P' = \{\mathit{head}(r) \leftarrow p, \mathit{body}(r) \mid r \in P\} \cup \{q \leftarrow \mathit{not}\ p. \ p \leftarrow \mathit{not}\ q\} \cup \{a \leftarrow \ \mid a$ appears in $P\}$, where p and q are two fresh atoms. This program P' has one answer set X_0 in which p is false and all other atoms are true; all other answer sets are of the form $X \cup \{p\}$, where $X \in \mathcal{AS}(P)$. It holds that $X_0 \in \mathcal{AS}_p(P')$ iff P has no answer set. ■

The construction in the above proof can be extended to show Σ_3^p-hardness of credulous inference.

Theorem 6 *Given a disjunctive program P and literals l and l', deciding whether* $\mathsf{forget}(P, l) \models_c l'$ *is Σ_3^p-complete.*

In Theorem 6 a suitable l-answer set containing l' can be guessed and checked, by Theorem 5 using Σ_2^p-oracle. Hence, credulous inference $\mathsf{forget}(P, l) \models_c l'$ is in Σ_3^p. The matching lower bounds, Π_2^p- resp. Σ_3^p-hardness can be shown by encodings of suitable quantified Boolean Formulas (QBFs).

In Theorems 5 and 6, the complexity is coNP- and Σ_2^p-complete, respectively, if P is either negative or normal.

Theorem 7 *Given a* negative *program N, a literal l, and a set of literals X, deciding $X \in \mathcal{AS}_l(N)$ is co-NP-complete.*

Proof (Sketch) The co-NP membership holds since checking whether a set of literals X' is an answer set of a negative program P is polynomial. As for co-NP-hardness, let $C = C_1 \wedge \cdots \wedge C_k$ be a CNF over atoms $y_1, \ldots, y_m$, where each C_j is non-empty. For $1 \le i \le m$, let $N_i = \{y_i \leftarrow \mathit{not}\ y'_i. \ y'_i \leftarrow \mathit{not}\ y_i. \ y_i \leftarrow \mathit{not}\ l. \ y'_i \leftarrow \mathit{not}\ l\}$, and for $1 \le j \le k$, let $Z_j = \{y_i \mid y_i \in C_j\} \cup \{y'_i \mid \neg y_i \in C_j\}$. Define $N = \cup_{i=1}^m (N_i \cup \{\ \leftarrow \mathit{not}\ Z_i\}) \cup \{l \leftarrow \mathit{not}\ y_1. \ l \leftarrow \mathit{not}\ y'_1\}$. Then, $X = \{y_i, y'_i \mid 1 \le i \le m\}$ is an answer set of N. Moreover, X is an l-answer set, iff C is unsatisfiable. The satisfiable assignments correspond to the answer sets of N containing l. ■

This construction can be lifted to show that credulous inference $\models_c$ of a literal from the l-answer sets of N is Σ_2^p-hard.

Theorem 8 *Given a negative program N and literals l and l', deciding whether* $\mathsf{forget}(N, l) \models_c l'$ *is Σ_2^p-complete.*

Proof (Sketch). By Theorem 7, Σ_2^p membership is easy. As for Σ_2^p-hardness, take a QBF $\exists X \forall Z E$, where E is a DNF on X and Z and contains some variable from Z in each clause. Construct the same program as above in Theorem 7 for $C = \neg E$ and where $Y = X \cup Z$ and y_1 is from Z, but (1) omit the clauses $x_i \leftarrow not\ l$ and $x_i' \leftarrow not\ l$. (2) add a clause $l' \leftarrow not\ l$. For each subset $S \subseteq X$, the set

$$S \cup \{x_i' \mid x_i \in X \setminus S\} \cup Z \cup \{z_j' \mid z_j \in Z\} \cup \{l'\}$$

is an answer set of N. These are also all answer sets of N that contain l' (and do not contain l). Furthermore, this set is an l-answer set, iff there is no satisfying assignment for C ($=\neg E$) which corresponds on X to S. Overall, this means that there is some l-answer set of the program in which l' is true, iff the formula $\exists X \forall Z E$ is true. ■

In the proof of Theorem 7, a CNF is actually reduced to a normal program. It is thus easy to see the following result.

Theorem 9 *Given a* normal *program P, a literal l, and a set of literals X, deciding whether X is an l-answer set of P is co-NP-complete.*

Similarly, we can show the credulous reasoning with forgetting for normal program is Σ_2^p-complete.

Theorem 10 *Given a* normal *program P, a literal l, and a literal l', deciding whether* $\mathsf{forget}(P, l) \models_c l'$ *is Σ_2^p-complete.*

By applying techniques that build on non-uniform complexity classes similar as in (Cadoli *et al.* 2000), we conjecture that there is no program $\mathsf{forget}(P, l)$ of polynomial size unless the polynomial hierarchy collapses, even if auxiliary literals might be used (which are projected off). Thus, the exponential blow up of $\mathsf{forget}(P, l)$ is, to some extent, unavoidable in general.

Related Work and Conclusion

We have proposed a theory of forgetting literals in disjunctive programs. Although our approach is purely declarative, we have proved that it is coupled by a syntactic counterpart based on program transformations. The properties of forgetting show that our approach captures the classical notion of forgetting. As we have explained before, the approach in this paper naturally generalizes the forgetting for normal programs investigated in (Wang, Sattar, & Su 2005).

Another approach to forgetting for normal programs is proposed in (Zhang, Foo, & Wang 2005), which is purely procedural since the result of forgetting is obtained by removing some rules and/or literals. A shortcoming of that approach is that there is, intuitively, no semantic justification for the removal.

As an application of forgetting, we have also presented a fairly general framework for resolving conflicts in disjunctive logic programming. In particular, this framework provides an elegant solution to the preference recovery problem. There are some interesting issues to be pursued. First, we are currently improving and implementing the algorithm for computing the result of forgetting. Second, we will explore the application of forgetting in various scenarios of conflict resolving, such as belief merging, update of disjunctive programs, inheritance in disjunctive programs.

References

Baral, C. 2002. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press.

Ben-Eliyahu, R., and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Ann. Math. and AI* 12(1-2):53–87.

Brass, S., and Dix, J. 1999. Semantics of disjunctive logic programs based on partial evaluation. *J. Logic Programming* 38(3):167–312.

Cadoli, M.; Donini, F.; Liberatore, P.; and Schaerf, M. 2000. Space Efficiency of Propositional Knowledge Representation Formalisms. *J. Artif. Intell. Res.* 13:1–31.

Eiter, T., and Fink, M. 2003. Uniform equivalence of logic programs under the stable model semantics. In *Proc. 19th ICLP*, 224–238.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys* 33(3):374–425.

Eiter, T., and Gottlob, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. and AI* 15(3-4):289–323.

Gelfond, M., and Lifschitz, V. 1990. Logic programs with classical negation. In *Proc. ICLP*, 579–597.

Janhunen, T.; Niemelä, I.; Simons, P.; and You, J.-H. 2000. Partiality and Disjunctions in Stable Model Semantics. In *Proc. KR 2000*, 411–419.

Lang, J., and Marquis, P. 2002. Resolving inconsistencies by variable forgetting. In *Proc. 8th KR*, 239–250.

Lang, J.; Liberatore, P.; and Marquis, P. 2003. Propositional independence: Formula-variable independence and forgetting. *J. Artif. Intell. Res.* 18:391–443.

Lee, J., and Lifschitz, V. 2003. Loop formulas for disjunctive logic programs. In *Proc. ICLP*, 451–465.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2004. The DLV System for Knowledge Representation and Reasoning. *ACM TOCL* (to appear).

Lifschitz, V.; Tang, L.; and Turner, H. 1999. Nested expressions in logic programs. *Ann. Math. and AI* 25:369–389.

Lifschitz, V., and Turner, H. 1994. Splitting a logic program. In *Proc. ICLP*, 23-37.

Lin, F., and Reiter, R. 1994. Forget it. In *Proc. AAAI Symp. on Relevance*, 154–159.

Lin, F., and Zhao, Y. 2004. ASSAT: Computing answer set of a logic program by sat solvers. *Artif. Intell.* 157(1-2): 115–137.

Wang, K., and Zhou, L. 2005. Comparisons and computation of well-founded semantics for disjunctive logic programs. *ACM TOCL* 6(2):295–327.

Wang, K.; Sattar, A.; and Su, K. 2005. A theory of forgetting in logic programming. In *Proc. 20th AAAI*, 682–687. AAAI Press.

Zhang, Y.; Foo, N.; and Wang, K. 2005. Solving logic program conflicts through strong and weak forgettings. In *Proc. IJCAI*, 627–632.