A Theory of Forgetting in Logic Programming*

Kewen Wang[1,2] and **Abdul Sattar**[1,2] and **Kaile Su**[1]
[1]Institute for Integrated Intelligent Systems
[2]School of Information and Computation Technology
Griffith University, Australia
{k.wang,a.sattar,k.su}@griffith.edu.au

Abstract

The study of *forgetting* for reasoning has attracted considerable attention in AI. However, much of the work on forgetting, and other related approaches such as independence, irrelevance and novelty, has been restricted to the classical logics. This paper describes a detailed theoretical investigation of the notion of forgetting in the context of logic programming. We first provide a semantic definition of forgetting under the answer sets for extended logic programs. We then discuss the desirable properties and some motivating examples. An important result of this study is an algorithm for computing the result of forgetting in a logic program. Furthermore, we present a modified version of the algorithm and show that the time complexity of the new algorithm is polynomial with respect to the size of the given logic program if the size of certain rules is fixed. We show how the proposed theory of forgetting can be used to characterize the logic program updates.

Introduction

The ability of discarding irrelevant information is a key feature that an intelligent agent must possess to adequately handle reasoning tasks such as query answering, planning, decision-making, reasoning about actions, knowledge update and revision. This ability is referred to as forgetting (Lin and Reiter 1994) or elimination (Brown 1990), and, often, studied under different names such as irrelevance, independence, irredundancy, novelty and separability (see (Subhramanian *et al.* 1997; Lang *et al.* 2003) for more details). For example, we have a knowledge base K and a query Q. It may be hard to determine if Q is true or false directly from K. However, if we discard or forget some part of K that is independent of Q, the querying task may become much easier.

According to (Lin and Reiter 1994), if T is a theory in propositional language and p is a ground atom, then the result of forgetting p in T is denoted $\mathsf{forget}(T, p)$ which can be characterized as $T(p/true) \vee T(p/false)$, i.e. the disjunction of two theories obtained from T by replacing p by $true$ and $false$, respectively. For example, if $T = \{(cooking \vee cleaning) \wedge singing\}$, then $\mathsf{forget}(T, singing)$ is the theory $\{cooking \vee cleaning\}$.

The notion of forgetting has found its applications in artificial intelligence. However, the existing theories of forgetting are mainly investigated in the context of classical logics. It would be interesting to establish a theory of forgetting in logic programming and nonmonotonic reasoning. This issue is first considered in (Zhang *et al.* 2005) and consequently two kinds of forgetting are defined (the strong and weak forgettings) by first transforming a logic program P into a reduced form and then deleting some rules (and literals). While they have been used to resolve conflicts in logic programming, these approaches suffer from some shortcomings: (1) There is no semantic justification for the strong or weak forgetting. Specifically, the relationship between the semantics of a logic program and the result of the strong or weak forgetting is unclear. (2) It is not addressed in (Zhang *et al.* 2005) that what are the desirable properties for a reasonable notion of forgetting in logic programming. In particular, one may ask what is the difference of these notions of forgetting from traditional approaches to deletion of rules/literals in logic programming and databases. (3) More importantly, both of the strong and weak forgettings are syntax-sensitive. That is, equivalent programs may have different results of forgetting about the same literal. For example, $P = \{p \leftarrow .\ \ q \leftarrow not\ p\}$ and $P' = \{p \leftarrow\}$ are equivalent programs under the answer sets. However, $\mathsf{WForgetLP}(P, p) = \{q \leftarrow\}$ and $\mathsf{WForgetLP}(P', p) = \{\}$ are not equivalent. Here $\mathsf{WForgetLP}(P, p)$ denotes the result of the weak forgetting about p in P.

Thus, a more reasonable notion of forgetting is highly desirable for nonmonotonic reasoning (and logic programming). In this paper, we choose answer set programming (ASP) (Lifschitz 2002) as the underlying nonmonotonic logic. ASP is a paradigm of logic programming under the answer sets (Gelfond and Lifschitz 1990) and it is becoming one of the major tools for knowledge representation due to its simplicity, expressive power, connection to major nonmonotonic logics and efficient implementations. First of all, we believe that a reasonable semantic notion of forgetting in ASP should satisfy the following criteria. Let P be a logic program and P' be the result of forgetting about a literal l in P.

(F1) The proposed forgetting is a natural generalization of

*The third author was partially supported by the NSFC under grants 60496327, 10410638 and 60473004
Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

classical one: $T(p/true) \vee T(p/false)$.

(F2) No new symbol is introduced in P'.

(F3) The reasoning under P' is equivalent to the reasoning under P *if* l *is ignored.*

(F4) The definition of forgetting is not syntax-sensitive. That is, the results of forgetting about l in equivalent programs are also equivalent.

(F5) The semantic notion of forgetting should be coupled with a syntactic counterpart.

(F1) specifies the major intuition behind forgetting and clarifies the difference of forgetting from deletion; (F2) is necessary because the forgetting is to eliminate redundant symbols. This is a difference of forgetting from some approaches to revision, update and merging (it is another issue to combine forgetting with other approaches to adding new information); (F3) provides a semantic justification for the forgetting. Note that P' and P may have different answer sets in general (see Proposition 3); (F4) guarantees the notion of forgetting is well-defined. (F5) is useful for applications of forgetting in knowledge representation.

To our best knowledge, *there is no theory of forgetting in nonmonotonic reasoning or logic programming which is based on the above criteria*. However, the definition of forgetting in classical logic cannot be directly translated to logic programming. For example, if P is a logic program, $P(p/true) \vee P(p/false)$ is not even a logic program in general. Moreover, as we will see later, it is not straight forward to replace $P(p/true) \vee P(p/false)$ by an appropriate logic program.

In this paper, we first introduce a notion of forgetting in answer set programming and then show that this notion of forgetting satisfies the above criteria (F1)-F(4) as well as some other attracting properties (e.g. Theorem 1). Thus our notion of forgetting captures the classical notion of forgetting. To justify (F5), we then develop an algorithm for computing the result of forgetting in a given logic program; and a variant of the algorithm that is polynomial time if the size of certain rules is fixed. These results illustrate that our notion of forgetting possesses all major properties of classical forgetting. The proposed theory of forgetting provides a general framework for reasoning tasks such as merging, update and revision of logic programs. As an example, we show how to capture the update answer sets (Eiter *et al.* 2002) by using our theory of forgetting. Our study also shows that developing a semantic theory of forgetting for logic programs is a non-trivial task.

Preliminaries

We deal with extended logic programs (Gelfond and Lifschitz 1990) whose rules are built from some atoms where default negation not and strong negation $\neg$ are allowed. A literal is either an atom a or its strong negation $\neg a$. For any atom a, we say a and $\neg a$ are complementary literals. For any set X of literals, $not\ X = \{not\ l \mid l \in X\}$.

An *extended logic program* is a finite set of rules of the following form

$$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n \qquad (1)$$

where l_0 is either a literal or empty, each l_i is a literal for $i = 1, \ldots, n$, and $0 \leq m \leq n$. If l_0 is empty, then the rule is a *constraint*.

Without loss of generality, we consider propositional programs. For technical reasons, it should be stressed that the body of a rule is a set of literals rather than a multiset. For instance, $a \leftarrow not\ b, c, c$ is not a rule in our sense while $a \leftarrow not\ b, c$ is a rule. That is, we assume that any rule of a logic program has been simplified by eliminating repeated literals in its body.

If a rule of form (1) contains no default negation, it is called *positive*; P is a positive program if every rule of P is positive.

If a rule of form (1) contains no body atoms, it is called *negative*; P is a negative program if every rule of P is negative.

Given a rule r of form (1), $head(r) = l_0$ and $body(r) = body^+(r) \cup not\ body^-(r)$ where $body^+(r) = \{l_1, \ldots, l_m\}$, $body^-(r) = \{l_{m+1}, \ldots, l_n\}$. The set $head(P)$ consists of all literals appearing in rule heads of P.

In the rest of this section we assume that P is an extended logic program and X is a set of literals. A rule r in P is satisfied by X, denoted $X \models r$, iff "if $body^+(r) \subseteq X$ and $body^-(r) \cap X = \emptyset$, then $head(r) \in X$". X is a model of P, denoted $X \models P$ if every rule of P is satisfied by X.

The answer set semantics The *reduct* of logic program P on a set X of literals, written P^X, is obtained as follows:

- Delete every r from P such that there is a $not\ \ q \in body^-(r)$ with $q \in X$.
- Delete all negative literals from the remaining rules.

Notice that P^X is a set of rules without any negative literals. Thus P^X may have no model or have a unique minimal model, which coincides with the set of literals that can be derived by resolution.

X is a *answer set* of P if X is the minimal model of P^X.

A logic program may have zero, one or more answer sets. We use $\parallel P \parallel$ to denote the collection of answer sets of P.

A program is *consistent* if it has at least one answer set.

Two logic programs P and P' are *equivalent*, denoted $P \equiv P'$, if they have the same answer sets.

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 introduce a semantic definition of forgetting for extended logic programs. That is, we want to define what it means to forget about a literal l in a logic program P. The intuition behind the forgetting theory is to obtain a logic program which is equivalent to the original logic program if we ignore the existence of the literal l.

It is direct to forget a literal l in a set X of literals, that is, just remove l from X if $l \in X$. This notion of forgetting can be easily extended to subsets. A set X' is an l-subset of X if $X' - \{l\} \subseteq X - \{l\}$. Similarly, a set X' is a true l-subset of X if $X' - \{l\} \subset X - \{l\}$.

Two sets X and X' of literals are l-equivalent, denoted $X \sim_l X'$, iff $(X - X') \cup (X' - X) \subseteq \{l\}$.

Given a consistent logic program P and a literal l, we could define a result of forgetting about l in P as an extended logic program P' whose answer sets are exactly $\| P \| - l = \{X - \{l\} \mid X \in \| P \|\}$. However, such a notion of forgetting cannot even guarantee the existence for some simple programs. For example, consider $P = \{a \leftarrow . \quad p \leftarrow not\ q. \quad q \leftarrow not\ p\}$, then $\| P \| = \{\{a, p\}, \{a, q\}\}$ and thus $\| P \| - p = \{\{a\}, \{a, q\}\}$. Since $\{a\} \subseteq \{a, q\}$ and each answer set must be minimal, $\| P \| - p$ cannot be the set of answer sets of any logic program. So *we need a notion of minimality of answer sets which can naturally combine the definition of answer sets, minimality and forgetting together.*

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

1. *We say X is* l-minimal *in a collection $\mathcal{S}$ of sets of literals if $X \in \mathcal{S}$ and there is no $X' \in \mathcal{S}$ such that X' is a true l-subset of X . In particular, if $\mathcal{S}_P$ is the set of models of P, then we say X is an* l-minimal model *of a logic program P if X is a model of P and it is l-minimal in $\mathcal{S}_P$.*
2. *X is an* answer set of P by forgetting l *(briefly,* l-answer set*) if X is an l-minimal model of the reduct P^X.*

For $P = \{a \leftarrow . \quad q \leftarrow not\ p. \quad p \leftarrow not\ q\}$, it 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 logic program P and a literal l, an answer set may not be an l-answer set.

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 logic program.

Definition 2 *Let P be a consistent logic program and l be a literal. A logic program P' is a result of* forgetting about l in P *if the following conditions are satisfied:*

1. $B_{P'} \subseteq B_P - \{l\}$.
2. *For any set X' of literals, 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'. In particular, no new symbol is introduced in P'.

A logic program P may have different logic programs as results of forgetting about the same literal l. However, it follows from the above definition that any two results of forgetting about the same literal in P are equivalent under the answer set semantics.

Proposition 1 *Let P be an extended logic 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 (i.e. they have the same answer sets).*

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

Examples

1. If $P_1 = \{q \leftarrow not\ p\}$, then $\mathsf{forget}(P_1, p) = \{q \leftarrow\}$ and $\mathsf{forget}(P_1, q) = \{\}$.
2. If $P_2 = \{q \leftarrow not\ p. \quad p \leftarrow not\ q\}$, then $\mathsf{forget}(P_2, p) = \{q \leftarrow q\}$. The reason is that P_2 has two answer sets $\{p\}$ and $\{q\}$ but only $\{p\}$ is a p-answer set of P_2. Thus $\mathsf{forget}(P_2, p)$ has a unique answer set $\{\}$.
3. Consider $P_3 = \{q \leftarrow not\ p. \quad p \leftarrow\}$, which has the unique answer $\{p\}$. Thus $\mathsf{forget}(P_3, p) = \{\}$ rather than $\{q \leftarrow\}$. This is intuitive because we are forgetting all impacts of p on P_3. In particular, "forgetting about p" is different from "assuming $not\ p$".
4. Let $P_4 = \{a \leftarrow not\ b. \quad b \leftarrow not\ a. \quad p \leftarrow not\ a. \quad c \leftarrow not\ p\}$. According to (Zhang *et al.* 2005), the weak forgetting $\mathsf{WForgetLP}(P_4, p) = \{a \leftarrow not\ b. \quad b \leftarrow not\ a. \quad c \leftarrow\}$; the strong forgetting of $\mathsf{SForgetLP}(P_4, p) = \{a \leftarrow not\ b. \quad b \leftarrow not\ a\}$. However, $\mathsf{forget}(P_4, p) = \{a \leftarrow not\ b. \quad b \leftarrow not\ a. \quad c \leftarrow a\}$.

In the next section we will explain how to obtain $\mathsf{forget}(P, l)$. The following proposition generalizes 1 and shows that the criteria (F4) is satisfied.

Proposition 2 *Let P and P' be two equivalent logic programs and l a literal in P. Then* $\mathsf{forget}(P, l)$ *and* $\mathsf{forget}(P', l)$ *are also equivalent under the answer sets.*

Now we show that our notion of forgetting does satisfy (F3).

Proposition 3 *For any consistent program P and a literal l in P, the following two 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 $X' \subseteq X$.*

This result implies that, if l is ignored, $\mathsf{forget}(P, l)$ is equivalent to P under both credulous and skeptical reasoning with respect to the answer set semantics.

Let $\mathsf{lcomp}(P)$ be Clark's completion plus the loop formulas for P. Then it is shown in (Lin and Zhao 2004) that X is an answer set of P iff X is a model of $\mathsf{lcomp}(P)$ (in the classical logic).

Theorem 1 *Let P be a logic program and l a literal in P. Then*

$$\mathsf{lcomp}(\mathsf{forget}(P, l)) \equiv \mathsf{forget}(\mathsf{lcomp}(P), l).$$

This result means that the answer sets of $\mathsf{forget}(P, l)$ are exactly the models of the result of forgetting about l in the classical theory $\mathsf{lcomp}(P)$. Thus $\mathsf{forget}(P, l)$ can be intuitively and completely characterized by the classical forgetting. 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.

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 $X_1 \subseteq_F X_2$, $X_1 =_F X_2$ and F-answer sets of a logic program. Those 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 4 *Let P be a consistent program and $F = F' \cup \{l\}$. Then*

$$\mathsf{forget}(P, F) \equiv \mathsf{forget}(\mathsf{forget}(P, l), F').$$

Computation of Forgetting

Since Definition 2 is a semantic one, it does not guarantee the existence of the result of forgetting about l in P. So one important issue is to study the problem of computing the result of forgetting. In the following we will justify the criterion (F5) by developing algorithms for computing forget(P, l) using program transformations. The basic idea is to *equivalently* transform the original program P into a standard form N first and then to obtain forget(P, l) directly from N.

Program transformations

The program transformations in this section are introduced in (Brass *et al.* 2001). P and P' are extended logic programs.

Elimination of Tautologies P' is obtained from P by the elimination of tautologies if there is a rule r in P such that $head(r) \in body^+(r)$ and $P' = P - \{r\}$.

Positive Reduction P' is obtained from P by the positive reduction if there is a rule 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.

Negative Reduction P' is obtained from P by negative reduction if there are two rules r and r' in P such that $body(r') = \emptyset$ and $head(r') \in body^-(r)$. and $P' = P - \{r\}$.

Let r and r' be two distinct rules in a logic program. We say r' is an implication of r if $head(r) = head(r')$ and $body(r) \subset body(r')$.

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

For two rules r and r' with $head(r') \in body^+(r)$, the unfolding of r with r', denoted $unfold(r, r')$, is the rule $head(r) \leftarrow (body(r) - \{head(r')\}), body(r')$.

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

$$P' = P - \{r\} \cup \{unfold(r, r') \mid r' \in P, head(r') \in body^+(r)\}.$$

$\mathcal{T}$ denotes the set of the program transformations introduced above.

Lemma 1 *(Brass* et al. *2001) Every logic program can be transformed into a canonical form by $\mathcal{T}$, which is a negative program.*

This lemma is also true for extended logic programs.

Algorithms for Computing forget(P, l)

We are now ready to present our basic algorithm for computing a result of forgetting about a given literal in a logic program.

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

Step 1. Transform P into its canonical form N.

Step 2. Suppose that N has n rules having head l (but body does not contain l due to the Elimination of tautology) where $n \geq 0$:

$$r_j : l \leftarrow not\ l_{j1}, ..., not\ l_{jm_j}$$

where $j = 1, \ldots, n$ and $m_j \geq 0$ for all j.

If $n = 0$, then N' is 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 D_1 is defined as false.

If $n \geq 1$ and $m_1 > 0$, then $m_i > 0$, for $i = 1, \ldots, s$, by the Elimination of implications. Let $D_1, \ldots, D_s$ be all possible conjunctions $(l_{1k_1}, \cdots, l_{nk_n})$ where $0 \leq k_1 \leq m_1$, ..., $0 \leq k_n \leq m_n$.

Replace every appearance of $not\ l$ in rule bodies of N by all possible D_i and the resulting program is denoted Q.

Step 3. Remove all rules with head l from Q and the resulting program is denoted N'.

Output N' as forget(P, l).

Theorem 2 *For any consistent program P and a literal l, Algorithm 1 always returns a result of forgetting about l in P.*

Before we present a proof sketch of Theorem 2, first consider two examples.

Example 1 *Let $P = \{a \leftarrow not\ b.\quad b \leftarrow not\ a.\quad c \leftarrow not\ b.\quad p \leftarrow not\ a, not\ c.\quad d \leftarrow not\ p\}$. Since P is a negative program, we can directly get* forget$(P, p) = \{a \leftarrow not\ b.\quad b \leftarrow not\ a.\quad c \leftarrow not\ b.\quad d \leftarrow a.\quad d \leftarrow c\}$

Consider another program for which program transformations are needed.

Example 2 *Let $P = \{p \leftarrow not\ p_1.\quad p \leftarrow p, not\ q_1.\quad p \leftarrow p_1, not\ q_1.\quad p_1 \leftarrow not\ p_2.\quad p_1 \leftarrow p_2, not\ q\}$. Then it can be equivalently transformed into the program $N = \{p \leftarrow .\quad p_1 \leftarrow\}$. So* forget$(P, p) = \{p_1 \leftarrow\}$, forget$(P, q) = \{p \leftarrow .\quad p_1 \leftarrow\}$, forget$(P, p_1) = \{p \leftarrow\}$.

For some special programs, it can be easier to compute forget(P, l).

Corollary 3 *Let P be a consistent program and N be the canonical program obtained from P by $\mathcal{T}$. Then*

1. *If $l \in X$ for every $X \in \| P \|$, then* forget(P, l) *is the program obtained from N by removing all rules containing l. In this case,* forget(P, l) *coincides with the strong forgetting defined in (Zhang* et al. *2005).*
2. *If $l \notin X$ for every $X \in \| P \|$, then* forget(P, l) *is the program obtained from N by removing all rules with head l and all appearances of negative literal $not\ l$. In this case,* forget(P, l) *coincides with the weak forgetting defined in (Zhang* et al. *2005).*

While Algorithm 1 provides a canonical form for the result of forgetting, it is exponential in the worst case. This may be from two sources: the Unfolding and the construction of D_i for $i = 1, \ldots, s$. However, we can replace the Unfolding by a restricted version, that is, only unfolding on l.

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

$$\begin{aligned} P' &= P - \{r\} \\ &\quad \cup \{unfold(r, r') \mid \text{there is a rule } r' \in P \\ &\qquad \text{such that } head(r') = l, l \in body(r)\}. \end{aligned}$$

Notice that we require $head(r') = l$ here. If we just want to obtain $\mathsf{forget}(P, l)$, some program transformations are not actually needed.

Algorithm 2 Input*: logic program P and a literal l.*
Procedure*:*

Step 1. Fully apply the Negative Reduction and the Unfolding on l to program P and the resulting program is denoted N.

Step 2. Suppose that N has n rules whose head contains l (but body does not contain l due to the Elimination of tautology) where $n \geq 0$:

$$r_j : l \leftarrow l'_{j1}, ..., l'_{jn_j}, not\ l_{j1}, ..., not\ l_{jm_j}$$

where $n_j, m_j \geq 0$ for all j ($1 \leq j \leq n$).

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

If $n = 1$, $n_1 = 0$ and $m_1 = 0$, then $l \leftarrow$ is the only rule in N whose head is l. In this case D_1 is false.

If $n \geq 1$ and $n_1 + m_1 > 0$, then $n_i + m_i > 0$, for $i = 1, \ldots, s$, by the Elimination of implications. Let $D_1, \ldots, D_s$ be all possible conjunctions $(L_{1k_1}, \cdots, L_{nk_n})$ where each L_{ik_i} is either $not\ l'_{ik_i}$ or l_{ik_i} for $i = 1, \ldots, n$.

Replace every appearance of $not\ l$ in rule bodies of N by all possible D_i and the resulting program is denoted Q_l.

Step 3. Remove all rules with head l and the resulting program is denoted N'.

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

Similar to Theorem 2, we can prove that Algorithm 2 is correct but the proof is more tedious since N is not negative.

Theorem 4 *For any consistent program P and a literal l, Algorithm 2 always returns a result of forgetting about l in P.*

Notice that the first step in Algorithm 4 can be finished in polynomial time. In most cases, the size of rules having head l is not very large compared to the size of the whole program. So we can get a polynomial algorithm for computing $\mathsf{forget}(P, l)$ if the size of rules having head l is bounded.

Corollary 5 *If the size of rules in P having head l is bounded, then Algorithm 2 is polynomial in the size of P.*

Proof Sketch of Theorem 2

Since P is finite, Algorithm 1 will finally terminate.

To prove that N' is a result of forgetting about l in P, we need only to show that N' is a result of forgetting about l in N. That is, for any set X' of literals (with $l \notin X'$), X' is an answer set of N' iff there exists an l-answer set X of N such that $X \sim_l X'$.

N can be split into three disjoint parts: $N = N_1 \cup N_2 \cup N_3$ where N_1 consists of rules in N in which l does not appear; $N_2 = \{r \in N \mid l \in head(r), l \notin body^-(r)\}$; $N_3 = \{r \in N \mid l \notin head(r), l \in body^-(r)\}$.

Let N'_3 be the program obtained from N_3 by performing transformations in Step 2. Then $N' = N_1 \cup N'_3$.

Let $D_1, D_2, \ldots, D_s$ denote the conjunctions constructed from l_{ij} in Step 2. So each rule r in N with $l \in body(r)$ corresponds to s rules in N': $head(r) \leftarrow D_i, (body(r) - \{not\ l\})$ for $i = 1 \ldots, s$.

$\Rightarrow$:) Suppose that X is an answer set of N'. We want to prove that there is an l-answer set X' of N such that $X \sim_l X'$.

If $n = 0$ or ($n = 1$ and $m_1 = 0$), the proof is direct. So we assume that $n \geq 1$ and $m_i \geq 1$ for $i = 1, \ldots, n$. Consider two possible cases:

Case 1. $X \not\models D_i$ for all $i = 1, \ldots, s$:

(1.1) We first show that $X' = X \cup \{l\}$ is a model of N.

Since $l \notin X$, we have

$N^{X'} = (N_1 \cup N_2 \cup N_3)^{X'} = (N_1)^X \cup (N_2)^X$.

It can be shown that X' is a model of both $(N_1)^X$ and $(N_2)^X$. Thus X' is a model of N^X.

(1.2) We then show that $X' = X \cup \{l\}$ is an l-minimal model of $N^{X'}$.

Suppose that X'' is the least model of $N^{X'}$. Notice that $l \in X''$ because the rule $l \leftarrow$ is in N^X. Thus we need only to show that X' is the least model of $N^{X'}$. (1.2) We then show that $X' = X \cup \{l\}$ is an l-minimal model of $N^{X'}$. By the assumption, $(X'' - \{l\}) \not\models D_i$ for all i with $1 \leq i \leq s$. Thus $(X'' - \{l\}) \models (N')^X$, which implies that $(X'' - \{l\}) = X$. So we have $X'' = X'$. Therefore X' is l-answer set of N.

Case 2. If $X \models D_{i_0}$ for some i with $1 \leq i_0 \leq s$: We need only to show that X is l-answer set of N.

(2.1) Since $N^X = (N_1)^X \cup (N_2)^X \cup (N_3)^X$, we can show that X is a model of N^X.

(2.2) If $X' \subseteq_l X$ such that $X' \models N^X$, then we can show that X is also l-answer set of N by distinguishing two possible subcases: $l \in X'$ and $l \notin X'$. $\Leftarrow$:) Conversely, suppose X is l-answer set of N. We want to prove that $X - \{l\}$ is an answer set of N'.

Case 1. $l \notin X$: We show that X is an answer set of N'. Note that $(N')^X = (N_1 \cup N'_3)^X = (N_1)^X \cup (N'_3)^X$..

It can be shown that X is a model of both Thus X is a model of $(N')^X$. On the other hand, suppose $X' \subseteq X$ and $X' \models (N')^X$. By $l \notin X$, we have $(N_2)^X = \emptyset$. For any rule $r' \in (N_3)^X$, it is of the form $h \leftarrow$. Then the rules of the form $h \leftarrow D_i$ for $1 \leq i \leq s$ are all in $(N'_3)^X$. Notice that $l \notin X'$ implies $X' \models D_{i_0}$ for some i_0 with $1 \leq i_0 \leq s$. It follows from $X' \models (l' \leftarrow D_{i_0})$ that $h \in X'$, which implies that $X' \models r'$. So $X' \models (N_3)^X$. We have $X' \models (N')^X$, a contradiction. Thus, X is an answer set of N'.

Case 2. $l \in X$: We show that $X' = X - \{l\}$ is an answer set of N'. In this case, $X \models (N_1)^X$ and $X \not\models D_i$ for all i, which imply $X' \models (N_1)^{X'}$ and $X' \not\models D_i$ for all i since l does not appear in D_i or N_1. Thus $X' \models (N')^{X'}$. On the other hand, suppose $X'' \subseteq X'$ and $X'' \models (N')^{X'}$. Then $X'' \cup \{l\} \models (N_1)^X$ and $X'' \cup \{l\} \models (N_2)^X$. Obviously, $X'' \cup \{l\} \models (N_3)^X$. By the l-minimality of X, we have $X'' \cup \{l\} =_l X$. Thus, $X'' = X'$. That is, X' is a minimal model of $(N')^{X'}$. Therefore, X' is an answer set of N'. ■

Logic Program Update by Forgetting

The theory of forgetting developed in previous sections is a very general framework for update, revision, merging, inheritance hierarchy and even preference handling in logic programming. In particular, the modified PMA (Doherty *et al*. 1998), the update answer sets (Eiter *et al*. 2002), inheritance answer sets (Buccafurri *et al*. 1999) and dynamic answer sets (Alferes *et al*. 1998) can all be captured by our forgetting operator. As an example, we show in this section how to characterize the update answer sets using forgetting.

An *update sequence* is an ordered sequence $\mathcal{P} = [P_1, P_2, \ldots, P_t]$ where each P_i is an extended logic program for $i = 1, \ldots, t$ and $t \geq 1$.

Informally, P_{i+1} is assumed to update the information represented by $[P_1, \ldots, P_i]$. So P_{i+1} represents more recent information than P_i and thus the rules in P_{i+1} are assigned higher priority in case of a conflict.

In Eiter *et al*'s approach, an update sequence $\mathcal{P}$ is first translated into a single logic program $P_\lhd$ and the answer set semantics of $\mathcal{P}$ is defined as the set of answer sets of $P_\lhd$. So this is basically a syntactic approach.

To define the translated program $P_\lhd$, for each literals l, we introduce new literals l_i and l_i^- where $i = 1, \ldots, t$. For each rule r, a new literal $rej(r)$ is also introduced. The extended Herbrand base is denoted $B^*_\mathcal{P}$.

Definition 3 *The update program $P_\lhd = P_1 \lhd \cdots \lhd P_t$ over B^*_P consists of the following rules:*

1. *All constraints in P_i ($1 \leq i \leq t$).*
2. *For each $r \in P_i$ ($1 \leq i \leq t$):*

$$\begin{array}{rcll} l_i & \leftarrow & body(r), not\ rej(r) & \text{if } head(r) = l \\ l_i^- & \leftarrow & body(r), not\ rej(r) & \text{if } head(r) = not\ l \end{array}$$

3. *For each $r \in P_i$ ($1 \leq i \leq t$):*

$$\begin{array}{rcll} rej(r) & \leftarrow & body(r), l_{i+1}^- & \text{if } head(r) = l \\ l_i^- & \leftarrow & body(r), l_{i+1} & \text{if } head(r) = not\ l \end{array}$$

4. *For each literal l in $\mathcal{P}$,*

$$l_i^- \leftarrow l_{i+1}^-; \quad l_i \leftarrow l_{i+1}; \quad l \leftarrow l_1; \quad \leftarrow l_1, l_1^-.$$

A set X of literals is an answer set of the update sequence $\mathcal{P}$ if $X = X' \cap B_\mathcal{P}$ for some answer set X' of $\mathcal{P}_\lhd$.

To characterize the update semantics, we define

$$F(X) = \cup_{1 \leq i \leq t} F_i(X)$$

where the sequence $F_n(X), \ldots, F_1(X)$ for $\mathcal{P}$ are recursively defined as $F_t(X) = \emptyset$, and for $i < t$,

$$\begin{array}{rcl} F_i(X) & = & \{head(r) \mid \text{ there exist } r \in P_i \text{ and } r' \in P_j \\ & & \text{s.t. } head(r) \text{ and } head(r') \text{ are complementary,} \\ & & X \models (body(r) \cap body(r')), \\ & & head(r') \notin F_{i+1}(X) \cup \cdots \cup F_t(X)\}. \end{array}$$

The intuition behind $F(X)$ is that a literal will be forgotten provided that its validity causes conflict with more recent information.

Theorem 6 *Let $\mathcal{P} = [P_1, \ldots, P_t]$ be an update sequence and X a set of literals. Denote $P = P_1 \cup \cdots \cup P_t$. Then X is an update answer set of $\mathcal{P}$ iff X is an answer set of* $\mathsf{forget}(P, F(X))$.

Conclusion

We have proposed a novel semantic approach to forgetting for reasoning with logic programs. The suitability of this notion of forgetting is justified against four criteria as well as illustrating examples. An important result is the algorithm for computing the result of forgetting in a logic program. Furthermore, we show that a variant of the algorithm is polynomial in the size of the given logic program if the size of certain rules is fixed. The proposed theory of forgetting is a very general framework for a variety of AI tasks including merging, update and revision of logic programs. In particular, we show that the update answer sets can be intuitively captured in our framework. Issues for future research include (1) The relation of our approach to relevance, independence and novelty. Note that the notion of relevance for reasoning can be naturally defined once the forgetting is defined. (2) Applications of our theory of forgetting in characterizing some other approaches to conflict resolving, for example, extended abduction introduced in (Sakama and Inoue 2003). (3) Determining classes of logic programs such that the computing of forgetting is computationally tractable.

References

Alferes, J., Leite, J, Pereira, L., Przymusinska, H., and Przymusinski, T. 1998. Dynamic logic programming. In *Proc. of KR'98*, pp.98–109.

S. Brass, J. Dix, B. Freitag, and U. Zukowski. Transformation-based bottom-up computation of the well-founded model. *Theory and Practice of Logic Programming*, 1(5):497–538, 2001.

F. Brown. *Boolean Reasoning: The Logic of Boolean Equations (2nd Edition)*. Dover Publications, 2003.

Buccafurri, F., Faber, W., and Leone, N. 1999. Disjunctive logic programs with inheritance. In *Proc. of ICLP'99*, pp.79–93.

Doherty, P., Lukaszewicz, W., and Madalinska-Bugaj, E. 1998. The pma and relativizing change for action update. In *Proc. of KR'08*, pp. 258–269.

Eiter, T., Fink, M., Sabbatini, G., and Tompits, H. 2002. On properties of update sequences based on causal rejection. *Theory and Practice of Logic Programming*, 2:711–767.

Gelfond, M., and Lifschitz, V. 1990. Logic programs with classical negation. In *Proc. of ICLP'90*, pp. 579–597.

J. Lang, P. Liberatore, and P. Marquis. Propositional independence: Formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.

V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.

F. Lin and Y. Zhao. Assat: Computing answer set of a logic program by sat solvers. *Artificial Intelligence*, 157:115–137, 2004.

F. Lin and R. Reiter. Forget it. In *Proc. of AAAI Symp. on Relevance*, pp. 154–159. New Orleans (LA), 1994.

C. Sakama and K. Inoue. An abductive framework for computing knowledge base updates. *Theory and Practice of Logic Programming*, 3(6):671–713, 2003.

Subhramanian, D., Greiner, R., and Pearl, J. (editors). 1997. *Artificial Intelligence: Special Issue on Relevance*.

Y. Zhang, N. Foo, and K. Wang. Solving logic program conflicts through strong and weak forgettings. In *Proc. of IJCAI*, 2005 (accepted).