Semantic Forgetting in Answer Set Programming ⋆

Thomas Eiter [a] Kewen Wang [b,⋆⋆]

[a] *Institut für Informationssysteme, Technische Universität Wien*
Favoritenstraße 9-11, A-1040 Vienna, Austria

[b] *School of Information and Communication Technology, Griffith University*
Brisbane QLD 4111, Australia

Abstract

The notion of forgetting, also known as variable elimination, has been investigated extensively in the context of classical logic, but less so in (nonmonotonic) logic programming and nonmonotonic reasoning. The few approaches that exist are based on syntactic modifications of a program at hand. In this paper, we establish a declarative theory of forgetting for disjunctive logic programs under answer set semantics that is fully based on semantic grounds. 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 can be entirely captured by classical forgetting. We present several algorithms for computing a representation of the result of forgetting, and provide a characterization of the computational complexity of reasoning from a logic program under forgetting. As applications of our approach, we present a fairly general framework for resolving conflicts in inconsistent knowledge bases that are represented by disjunctive logic programs, and we show how the semantics of inheritance logic programs and update logic programs from the literature can be characterized through forgetting. The basic idea of the conflict resolution 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.

Key words: answer set programming, nonmonotonic logic programs, knowledge representation, forgetting, computational complexity

⋆ Preliminary versions of this paper with some of the results have been presented at AAAI 2006 and at NMR 2006.

⋆⋆Part of the work was done while this author was visiting Technische Universität Wien.

Email addresses: `eiter@kr.tuwien.ac.at` (Thomas Eiter), `k.wang@griffith.edu.au` (Kewen Wang).

1 Introduction

For intelligent agents, the ability to discard irrelevant information has been recognized as an important feature (that is mastered well by humans) and received broad attention in artificial intelligence, both from a cognitive and a computational perspective. In the area of knowledge representation, this ability is often referred to as *forgetting* [49] or *variable elimination* [9], but has been studied under many different names including irrelevance, independence, irredundancy, novelty, or separability (see [63; 34] for more details).

Forgetting has its root in Boolean Algebra [6] where it is a fundamental reasoning process. C. I. Lewis [41] has pointed out that, for purposes of application of Boolean logic to commonsense reasoning, the elimination/forgetting is a process more important than solution [1] since most processes of reasoning take place through the elimination of "middle" variables. Boole writes of such middle variables that it "usually happens in commonsense reasoning, and especially when we have more than one premises, that some of the elements [in the premises] are not required to appear in the conclusion."

Forgetting and its applications have been investigated extensively in the context of classical logic, for example, [5; 34; 36; 37; 49; 55; 56; 67], but less so in nonmonotonic logic programming and reasoning. In this context, it was first considered in [70; 71], where two types of forgetting—strong and weak forgetting—have been defined by first transforming a logic program P into a reduced form and then deleting some rules (and literals) from it. While this approach works well in a number of cases, it has two major drawbacks. First, its semantic underpinning is not fully clear. Specifically, the relationship between the intended semantics of a logic program, in terms of its answer sets, and the result of the syntactic transformations that are carried out by strong and weak forgetting is unclear. Second, this approach does not address desirable properties for a reasonable notion of forgetting in nonmonotonic 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.

A further aspect is that both strong and weak forgetting are syntax-sensitive, i.e., programs that are semantically equivalent may have different results after forgetting about the same literal. For example, the programs $P = \{p \leftarrow .\ q \leftarrow not\ p\}$ and $Q = \{p \leftarrow\}$ are equivalent under the answer set semantics. Weak forgetting about p from P yields the program $\mathsf{WForgetLP}(P, p) = \{q \leftarrow\}$ and from Q the program $\mathsf{WForgetLP}(Q, p) = \{\}$; clearly, these programs are not equivalent.

[1] In [41] a problem is formulated as a Boolean equation such that a solution of the Boolean equation corresponds to a solution of the given problem. In particular, solving a Boolean equation is treated as a process of eliminating/forgetting variables that represent unknowns.

While the role of syntax in logic programming is, with respect to reading of rules as in classical logic well-acknowledged, one might argue that relative to the semantics of this syntax, equivalent programs should behave in the same way. In particular, in this example the result of forgetting about p in P and Q should yield semantically the same result (note that, under answer set semantics, the second rule in P is redundant).

A similar phenomenon can be observed for strong forgetting. Consider $P = \{q \leftarrow not\ p.\ \ q \leftarrow not\ q\}$ and $Q = \{q \leftarrow\}$. Then these two programs are equivalent under the answer set semantics. However, the results of strong forgetting about p from P and Q are $\mathsf{SForgetLP}(P, p) = \{q \leftarrow not\ q\}$ and $\mathsf{SForgetLP}(Q, p) = \{q \leftarrow\}$, respectively, which are obviously not equivalent. The discrepancy is here even more noticeable: the result of strong forgetting about an atom from a consistent program can be inconsistent.

Thus, an alternative notion of forgetting for nonmonotonic logic programming is highly desirable. In this paper, we choose answer set programming (ASP) [44] as the underlying nonmonotonic logic. ASP is a new paradigm of logic programming under the answer set semantics [29], which is becoming a major tool for knowledge representation and reasoning due to its simplicity, expressive power, and connection to major nonmonotonic logics. A number of efficient ASP solvers, such as DLV, Smodels, ASSAT, Cmodels, or Clasp are available (see [3]), which can handle large problem instances.

Prior to defining a notion of forgetting for nonmonotonic logic programming, we may pose the question what desirable properties a reasonable theory of forgetting should have. The following ones appear to be natural candidates for us. Let P be a logic program and let P' be the result of forgetting about a literal l in P.

(F1) The proposed notion of forgetting should be a "natural" generalization of, and relate to, forgetting in classical logic.
(F2) No new symbols are introduced in P', i.e., the vocabulary stays the same.
(F3) The reasoning under P' is equivalent to the reasoning under P *if l is ignored.*
(F4) The result of forgetting is not sensitive to syntax in that the results of forgetting about l in semantically equivalent programs should also be semantically equivalent.
(F5) The semantic notion of forgetting is coupled with a syntactic counterpart, i.e., there is effective constructible syntax for representing the result of forgetting.

Property (F1) specifies the major intuition behind forgetting and clarifies the difference of forgetting from deletion. (F2) is necessary because the forgetting is to eliminate the symbols to be forgotten. This is a difference between forgetting and some approaches to revision, update, and deletion, such as [1; 10; 19; 31; 16]; note that to combine forgetting with other approaches to adding new information is a different issue. Property (F3) provides a semantic justification for the

forgetting. Note that P' and P may have different answer sets in general (see Proposition 1); (F4) guarantees that the notion of forgetting is semantically well-defined. Finally, property (F5) is useful for applications of forgetting in knowledge representation.

To the best of our knowledge, there is no theory of forgetting in nonmonotonic reasoning or logic programming which is based on the above criteria; notice that properties (F3) and (F4) do not hold for weak and strong forgetting from [70; 71] (see Section 7 for more discussion). However, the definition of forgetting in classical logic cannot be directly adapted to logic programming (cf. Section 3.1). The main contributions of the present paper are as follows.

- We establish a declarative, semantically defined notion of forgetting for disjunctive logic programs under answer set semantics called *semantic forgetting*. The suitability of semantic forgetting is justified by a number of desirable properties, including the ones given above.
- As one of them, we show that our notion of forgetting naturally captures classical forgetting. As we show, this can be exploited for reasoning under forgetting about a literal from a logic program by resorting to representations of a nonmonotonic logic program in terms of classical logic [50; 51; 39].
- As another such property, for every consistent disjunctive program P and literal l, a syntactic representation $\mathsf{forget}(P, l)$ for forgetting about l in P in terms of a nonmonotonic logic program always exists. Besides two semantics-based algorithms for computing such a representation, we also present a transformation-based algorithm. This algorithm allows to obtain the result of forgetting about a literal l in P via a series of program transformations and other rewritings.
- In connection with these algorithms, we characterize the computational complexity of the major reasoning tasks from logic programs under forgetting about a literal. As it turns out, model checking and credulous reasoning under forgetting about a literal from a logic program are more complex than in the standard setting (by one level in the polynomial hierarchy), while skeptical reasoning has the same complexity. These results provide useful insights into feasible representations of forgetting, and suggest that a polynomial-size result of forgetting may not always be feasible. This (and stronger results) can be established by applying the theory of Cadoli et al. [12; 13].
- 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 some subgoals that may cause inconsistency. In particular, we show how to use the notion of forgetting to provide an elegant solution for preference elicitation in ASP.
- Furthermore, we show that inheritance programs [10], update programs [18; 19] and fragments of dynamic programs [1; 2] can be characterized in terms of the semantic forgetting.

While in this paper, we focus on nonmonotonic logic programs, the basic ideas un-

derlying our approach to semantic forgetting may be applied to other well-known formalisms of nonmonotonic reasoning, such as default logic [60] or autoepistemic logic [57], as well, of which nonmonotonic logic programs under answer set semantics can be seen as particular fragments. In fact, these formalisms extend classical logic, and a notion of forgetting that complies with classical forgetting (which is based on semantics) seems needed there. Our results thus also provide a benchmark for approaches to forgetting in other formalisms of nonmonotonic reasoning, which remain to be developed.

The rest of the paper is organized as follows. Section 2 briefly recalls some basics of disjunctive logic programs and the answer sets. Section 3 defines the notion of forgetting in ASP, shows some important properties, and relates it to classical forgetting and independence [34]. Thereafter, Section 4 presents algorithms for computing the result of forgetting in ASP, while Section 5 studies some complexity issues. Section 6 then presents some applications, namely to conflict resolution in multi-agent systems, to inheritance logic programs, and to logic program updates. The final Section 7 concludes the work.

2 Preliminaries

We briefly review some basic definitions and notation in answer set programming that will be used throughout this paper.

A *disjunctive logic program* (simply, *logic program*) is a finite set of rules of the form

$$a_1 \vee \cdots \vee a_s \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n, \tag{1}$$

where $s, m, n \geq 0$, and all a_i, b_j, and c_k are from a set Lit of classical literals in a propositional language We assume here that all a_i are pairwise distinct, and similarly all b_j and all c_k. A *literal* is a *positive literal* p or a *negative literal* $\neg p$ for some atom p. For an atom p, p and $\neg p$ are called *complementary*. For any literal l, its complementary literal is denoted by $\neg.l$.

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 $not\ body^-(r) = \{not\ q \mid q \in body^-(r)\}$. Occasionally, in abuse of notation we view $head(r)$ also as set $\{a_1, \ldots, a_s\}$.

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 $n = 0$. The rule with $s = n = m = 0$ is the constant *false*. A logic program P is called *normal* (resp. *positive*, *negative*), if every rule in P is normal (resp. positive, negative).

We denote by $Lit_P \subseteq Lit$ the *literal base* of logic program P, that is, the set of all literals occurring in P. Unless stated otherwise or clear from the context, Lit will be implicitly given by Lit_P. An *interpretation* is a set of literals $X \subseteq Lit$ that contains no pair of complementary 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 is satisfied by X, denoted $X \models r$, if $X \models head(r)$ whenever $body^+(r) \subseteq X$ and $body^-(r) \cap X = \emptyset$ hold. Furthermore, X is a model of P, denoted $X \models P$, if $X \models r$ for every rule $r \in P$. A model X of P is a *minimal model* of P if for any model X' of P, $X' \subseteq X$ implies $X' = X$.

The semantics of a logic program P is defined in terms of its *answer sets* [30] as follows. Given an interpretation X, the *reduct* of P on X is defined as $P^X = \{head(r) \leftarrow body^+(r) \mid r \in P, body^-(r) \cap X = \emptyset\}$. Then X is an *answer set* of P, if X is a minimal model of P^X. By $\mathcal{AS}(P)$ we denote the collection of all answer sets of P.

A logic program P may have zero, one or multiple answer sets. P is *consistent*, if it has at least one answer set. It is well-known that the answer sets of a logic program P are incomparable: for any X and X' in $\mathcal{AS}(P)$, $X \subseteq X'$ implies $X = X'$.

Example 1 *Let P be the logic program consisting of the following rules:*

$$a \vee b \leftarrow not\ c$$

$$d \leftarrow a$$

$$d \leftarrow b$$

Then P has two answer sets $X_1 = \{a, d\}$ and $X_2 = \{b, d\}$. Obviously, X_1 and X_2 are incomparable.

Two logic programs P and P' are *equivalent*, denoted $P \equiv P'$, if $\mathcal{AS}(P) = \mathcal{AS}(P')$, i.e., P and P' have the same answer sets.

By $P \models_s l$ and $P \models_c l$ we denote skeptical and credulous consequence of a literal l from a logic program P, respectively; that is, $P \models_s l$ iff $l \in S$ for every $S \in \mathcal{AS}(P)$ and $P \models_c l$ iff $l \in S$ for some $S \in \mathcal{AS}(P)$.

3 Forgetting in Logic Programming

In this section, we define what it means to forget about a literal l in a logic program P. The idea is to obtain a logic program which does not contain l and is equivalent to the original logic 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.

3.1 Definition of Semantic Forgetting

In classical propositional logic, the result of forgetting $\mathsf{forget}(T, p)$ about a proposition p in a finite theory T is conveniently defined as $T(p/true) \vee T(p/false)$, where $T(p/true)$ and $T(p/false)$ are obtained by taking the conjunction of all sentences in T and replacing all occurrences of p with $true$ and $false$, respectively. This method cannot be directly generalized to logic programming, since there is no notion of the "disjunction" of two logic programs. However, if we look at forgetting from a model-theoretic perspective, then we can obtain the models of $\mathsf{forget}(T, p)$ as follows: compute first all (2-valued) models of T and then 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 logic program P and a literal l, we could naively define the result of forgetting about l in P as a logic 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 $P = \{a \leftarrow .\ p \vee q \leftarrow\}$. Here $\mathcal{AS}(P) = \{\{a, p\}, \{a, q\}\}$ 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 logic 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 $X' \subseteq_l X$ and $X \subseteq_l X'$.

Definition 1 (l-Answer Set) *Let P be a consistent logic program, let l be a literal in Lit_P, and let $X \subseteq Lit_P$ 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 $X' \subset_l X$. By $\min_l(\mathcal{S})$ we denote the collection of all l-minimal elements in $\mathcal{S}$.*

(2) An answer set X of logic program P is an l-answer set *if X is l-minimal in $\mathcal{AS}(P)$. By $\mathcal{AS}_l(P)$ we denote the set of all l-answer sets of P.*

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 logic

program P and a literal l, not every answer set is an l-answer set.

The sets in $\mathcal{AS}_l(P) \setminus l = \{X \setminus \{l\} \mid X \in \mathcal{AS}_l(P)\}$ are incomparable, and so we can find a logic program which has this collection as its answer sets. Note that to achieve incomparability, one could select other answer sets than those which are minimal in $\mathcal{AS}(P) \setminus l$ (e.g., the maximal ones). However, selecting minimal answer sets is in line with the guiding principle of logic programming and nonmonotonic reasoning to minimize positive information.

Note that in the above definition, P was assumed to be consistent (i.e., $\mathcal{AS}(P) \neq \emptyset$). If a logic program is inconsistent, the result of forgetting seems not to be clear, since the possibility of removing inconsistency from the logic program might have to be considered. For example, a logic program P may have partial stable models [61] while it is inconsistent under the answer set semantics (i.e. the stable model semantics). Forgetting from inconsistent programs is an interesting issue, but we do not consider it here. In the rest of this paper, we always assume that P is a consistent logic program.

The following proposition collects some easy properties of l-answer sets.

Proposition 1 *For every consistent program P and every literal l in Lit_P, the following holds:*

(1) Every l-answer set X of P is an answer set of P.
(2) For every answer set X of P, there exists an l-answer set X' of P such that $X' \subseteq_l X$.
(3) Every 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 of P, then there exists an l-answer set Y of P such that $l \in Y$ and $Y \subset_l X$.
(5) If $l \notin Lit_P$, then X is an l-answer set of P iff X is an answer set of P.

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 (Semantic Forgetting) *Let P be a consistent logic program and l be a literal. A logic program P' represents the result of* forgetting about l in P, *if*

(1) $Lit_{P'} \subseteq Lit_P \setminus \{l\}$, i.e., l does not occur in P', and
(2) $\mathcal{AS}(P') = \mathcal{AS}_l(P) \setminus l$, i.e., for every set X' of literals such that $l \notin X'$, X' is an answer set of P' iff there exists an l-answer set X of P such that $X' \sim_l X$.

We use $\mathsf{forget}(P, l)$ *as a generic notation for a logic program representing the result of forgetting about l in P.*

An important difference of the notion of forgetting here from existing approaches to updating and merging logic programs, cf. [1; 10; 19; 31; 16], is that merely l and

possibly some other literals are removed. However, no new symbols are introduced in P'.

For a consistent logic program P, some program P' as in the above definition always exists (cf. Algorithm 2 for details). However, P' may not be unique. It follows from the definition that they are all equivalent under answer set semantics.

Proposition 2 *Let P be a consistent logic program and let $l \in Lit_P$ be a literal. If P' and P'' are two results for forgetting about l in P, then P' and P'' are equivalent.*

Before further properties of forgetting are explored in Section 3.2, let us look at some example programs.

Example 2 *(1) If $P_1 = \{q \leftarrow not\ p\}$, then $\mathsf{forget}(P_1, q)$ has the empty answer set and $\mathsf{forget}(P_1, p)$ the answer set $\{q\}$. A possible representation for $\mathsf{forget}(P_1, p)$ is obtained from P_1 by removing $not\ p$ in the rule $q \leftarrow not\ p$, and for $\mathsf{forget}(P_1, q)$ by removing the whole rule $q \leftarrow not\ p$.*

(2) If $P_2 = \{q \leftarrow not\ p.\ p \leftarrow not\ q\}$, then $\mathsf{forget}(P_2, p)$ has the empty answer set (which is represented by the empty program). Indeed, P_2 has two answer sets $\{p\}$ and $\{q\}$ but only $\{p\}$ is a p-answer set of P_2. Similarly, $\mathsf{forget}(P_2, q)$ has the empty answer set.

(3) Consider $P_3 = \{q \leftarrow not\ p.\ p \leftarrow\}$, which has the single answer $\{p\}$. Thus $\mathsf{forget}(P_3, p)$ has the empty answer set, which is represented by the empty program, rather than by $\{q \leftarrow\}$. This is intuitive, because we are forgetting all impacts of p on P_3. In particular, "forgetting about p" is different from "assuming $not\ p$".

From the above examples, one might guess that some program $\mathsf{forget}(P, p)$ can be obtained by simply removing rules and/or literals with head p and/or positive body literal p, and by removing $not\ p$ in the remaining rules. However, as the next example shows, this is not true in general.

(4) Let $P_4 = \{a \leftarrow not\ b.\ b \leftarrow not\ a.\ p \leftarrow not\ a.\ c \leftarrow not\ p\}$. According to [71], the result of weak forgetting about p in P_4 is the program $\mathsf{WForgetLP}(P_4, p) = \{a \leftarrow not\ b.\ b \leftarrow not\ a.\ c \leftarrow\}$, while the result of strong forgetting about p in P_4 is the program $\mathsf{SForgetLP}(P_4, p) = \{a \leftarrow not\ b.\ b \leftarrow not\ a\}$. Neither is fully intuitive: c depends on a (by means of double negation), but in both $\mathsf{WForgetLP}(P_4, p)$ and $\mathsf{SForgetLP}(P_4, p)$ any connection between c and a is lost. In contrast, $\mathsf{forget}(P_4, p)$ has the two answer sets $\{b\}$ and $\{a, c\}$ in which the connection between a and c (equivalence via "double negation" of p) is maintained. Syntactically, $\mathsf{forget}(P_4, p)$ may be represented by the program $\{a \leftarrow not\ b.\ b \leftarrow not\ a.\ c \leftarrow a\}$, for instance, where this connection is maintained via the rule $c \leftarrow a$.

(5) $P_5 = \{p \vee q \leftarrow not\ p.\ c \leftarrow q\}$ has the single answer set $\{q, c\}$. Here, p is involved in unstratified negation and must be false in every answer set. Thus $\mathsf{forget}(P_5, p)$ has the same answer set as P_5; a syntactic representation of the program is $= \{q \leftarrow .\ c \leftarrow\}$, which intuitively results by pushing through the

only possible value for p and simplifying the program.

(6) *Let $P_6 = \{a \vee p \leftarrow not\ b.\ c \leftarrow not\ p.\ b \leftarrow\}$. This program has the single answer set $\{b, c\}$, and no atom p is involved in cyclic negation. Forgetting about p in P_6 does not change the answer set; a possible program for* $\mathsf{forget}(P_6, p)$ *is $\{c \leftarrow .\ b \leftarrow\}$, which again corresponds to a simplified version of the program P_6 the results after pushing through the value of p (note that the first rule in P_6 is never applicable).*

We will discuss how to obtain a concrete program $\mathsf{forget}(P, l)$ in the next section.

3.2 Basic Properties of Forgetting

In this subsection, we present some further properties of forgetting. First, the number of answer sets can never increase.

Proposition 3 *Let P be a consistent logic program. Then, for every literal l in Lit_P, it holds that $|\mathcal{AS}(\mathsf{forget}(P, l))| \leq |\mathcal{AS}(P)|$.*

This is a simple consequence of the fact that only some, but not all answer sets of P are l-answer sets. Note that this property is compliant with the principle of closing the world, and eliminating possibilities in favor of a default case.

The following proposition generalizes Proposition 2.

Proposition 4 *Let P and P' be two consistent logic programs and l a literal in P. If P and P' are equivalent, 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 [45] or uniform equivalence [17; 20]. We recall that two programs P and Q are *strongly equivalent*, if for each (finite) set of rules R the programs $P \cup R$ and $Q \cup R$ have the same answer sets; they are *uniformly equivalent*, if this holds for each set of facts R.

We say that forgetting *preserves* an equivalence $\equiv_X$ on logic programs, if for every logic programs Q and Q' and for every literal l, $Q \equiv_X Q'$ implies $\mathsf{forget}(Q, l) \equiv_X \mathsf{forget}(Q', l)$. Here X can be strong equivalence, uniform equivalence, or any other equivalence relation on the collection of (disjunctive) logic programs.

An equivalence relation $\equiv_X$ for logic programs on Lit is *invariant* under literal extensions, if the following holds: whenever P and P' are programs such that $Lit_P, Lit_{P'} \subseteq Lit$ and $l \notin Lit$ is a new literal, then $P \equiv_X P'$ w.r.t. Lit iff $P \equiv_X P'$ w.r.t. $Lit \cup \{l\}$.

An equivalence $\equiv_X$ is *stronger* than ordinary equivalence $\equiv$ if the following conditions are satisfied:

(1) For any programs P and P', $P \equiv_X P'$ implies $P \equiv P'$.
(2) There exist two programs P and P' such that $P \equiv P'$ and $P \not\equiv_X P'$.

Proposition 5 *Let $\equiv_X$ be an equivalence relation on a collection of logic programs on Lit that is stronger than ordinary equivalence and invariant under literal extensions. Then forgetting does not preserve $\equiv_X$.*

Both strong and uniform equivalence [45; 17] are clearly stronger than ordinary equivalence, and clearly also invariant under literal extensions. Hence, none of them is preserved by the definition of forgetting introduced. This, however, is a consequence of the freedom to arbitrarily instantiate the generic program $\mathsf{forget}(P, l)$. For specific realizations of $\mathsf{forget}(P, l)$, both strong and uniform equivalence may be preserved under forgetting; for example, the realizations $\mathsf{forget}_1(P, l)$ and $\mathsf{forget}_2(P, l)$ in Section 4 have this property. A suitable notion of forgetting which preserves strong equivalence is interesting for some applications, but beyond the scope of this paper.

An issue related to this is a possible definition of forgetting in terms of a consequence relation between logic programs, in analogy to defining forgetting in the classical context [34], as the strongest program that is a consequence of the given program P and independent of the literal l to be forgotten. This, however, requires suitable notions of consequence and independence, and these are not straightforward. The former may well depend on the application purpose of a logic program, be it for query answering, or for representing solutions of a problem in its answer sets according to the ASP paradigm. Furthermore, answer set semantics is not monotonic, and thus a simple consequence operator $\models$ on a rule by rule bases, i.e., $P \models Q$ if $P \models r$ for each rule $r \in Q$, is not suitable. For example, the rule $p \leftarrow not\ p$ is true in all answer sets of $P = \{p \leftarrow .\ \}$, and vice versa $p \leftarrow$ is true in all answer sets of $Q = \{p \leftarrow not\ p\}$; however, P and Q are not equivalent since they have different answer sets. This problem may be overcome by using a consequence operator such as SE-consequence (equivalently, the Logic of Here-and-There [45]) or UE-consequence [20].

Also the notion of independence of a literal from a logic program can be defined in different ways. An obvious notion is syntactic independence, which is given if l does not occur in P. However, also semantic notions of independence are conceivable, and one of them can be defined in terms of forgetting; we discuss this further in Section 6.4.

Technically, we can reconstruct our notion of forgetting from a logic program in terms of syntactic independence and the following consequence relation between logic programs: $P \models_{\supseteq} Q$ if for each answer set S of P there exists some answer set S' of Q such that $S \supseteq S'$. Clearly, $\models_{\supseteq}$ is reflexive and transitive, and $P \models_{\supseteq} Q$,

$Q \models_{\supseteq} P$ implies that P and Q have the same answer sets. Then, it is not hard to see that each strongest program Q on literals $Lit_P \setminus \{l\}$ such that $P \models_{\supseteq} Q$ (i.e., for each other program Q' over $Lit_P \setminus \{l\}$ such that $P \models_{\supseteq} Q'$ it holds that $Q \models_{\supseteq} Q'$) represents $\mathsf{forget}(P, l)$. We leave the issue of defining forgetting through other notions of consequence for further investigation.

The following proposition summarizes some simple but helpful properties of forgetting.

Proposition 6 *For every consistent program P and every literal l in Lit_P, the following holds:*

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

The next proposition says that, after forgetting about a literal in a logic program, the resulting program is equivalent to the original one under skeptical reasoning, but weaker under credulous reasoning (i.e., inferences are lost).

Proposition 7 *Let P be a consistent logic program and let l, l' be literals in Lit_P such that $l' \neq l$. Then,*

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

The above definition of forgetting about a single literal l in a logic program P can be straightforwardly extended to a set F of literals. We can similarly define $X_1 \subseteq_F X_2$, $X_1 \sim_F X_2$, and F-answer sets of a logic program, and the properties of forgetting about a single literal can be generalized to this setting. Furthermore, the result of forgetting about a set F can be obtained by forgetting the literals in F one by one.

Proposition 8 *Let P be a consistent logic program and let $F = \{l_1, \ldots, l_m\}$ be a set of literals. Then $\mathsf{forget}(P, F) \equiv \mathsf{forget}(\mathsf{forget}(\mathsf{forget}(P, l_1), l_2), \ldots, l_m)$.*

Notice that the particular indexing of the literals in F does not matter. This result, which allows to reduce forgetting to the basic operation for a single literal, is quite useful, but its proof (which is given in the Appendix) requires some technicalities.

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 us consider a simple logic program which contains a pair of complementary literals.

Example 3 *Let P be the following logic program:*

$$\begin{aligned} \mathit{flies}(Tweety) &\leftarrow pigeon(Tweety). \\ \neg\mathit{flies}(Tweety) &\leftarrow penguin(Tweety). \\ pigeon(Tweety) \vee penguin(Tweety) &\leftarrow . \end{aligned}$$

This program has the answer sets $\{pigeon(Tweety),\ \mathit{flies}(Tweety)\}$ and $\{penguin(Tweety), \neg\mathit{flies}(Tweety)\}$. If we forget about only one of $\mathit{flies}(Tweety)$ and $\neg\mathit{flies}(Tweety)$, then the complementary literal must still be in the result of forgetting. For instance, $\mathsf{forget}(P, \mathit{flies}(Tweety))$ *has the answer sets $\{pigeon(Tweety)\}$ and $\{penguin(Tweety),\ \neg\mathit{flies}(Tweety)\}$, where the second one still contains $\neg flies(Tweety)$; a possible program representing* $\mathsf{forget}(P, \mathit{flies}(Tweety))$ *is*

$$\begin{aligned} \neg\mathit{flies}(Tweety) &\leftarrow not\ penguin(Tweety). \\ pigeon(Tweety) \vee penguin(Tweety) &\leftarrow . \end{aligned}$$

However, by Proposition 8 $\mathsf{forget}(\mathsf{forget}(P, \mathit{flies}(Tweety)), \neg\mathit{flies}(Tweety))$ *and* $\mathsf{forget}(\mathsf{forget}(P, \neg\mathit{flies}(Tweety)), flies(Tweety))$ *are equivalent and have the answer sets $\{pigeon(Tweety)\}$ and $\{penguin(Tweety)\}$, which are represented by the program*

$$pigeon(Tweety) \vee penguin(Tweety) \leftarrow .$$

3.3 Relation to Classical Forgetting

We now consider the relationship between classical forgetting $\mathsf{forget}(T, p)$ and logic programming forgetting $\mathsf{forget}(P, p)$. Besides the stable and the answer set semantics [28; 30], another influential semantics for nonmonotonic logic programming is the Clarke's completion semantics [14], which defines the semantics of logic programs in terms of classical logic. It is well-known that answer set and completion semantics are different in general. For example, the logic program $P = \{p \leftarrow q;\ q \leftarrow p\}$ has a unique answer set $\{\}$. However, Clarke's completion for P gives $\{p \equiv q\}$, which has two models $\{\}$ and $\{p, q\}$. Lin and Zhao [50; 51] showed that the answer set semantics for a logic program can be characterized by a simple extension of Clarke's program completion by adding so called *loop formulas*. They consider normal logic programs that may contain constraints. This

approach allows to compute the answer sets of a normal logic program using a classical SAT solver. Lee and Lifschitz [39] extended this characterization to the class of disjunctive logic programs. For simplicity, we assume in this subsection that programs have no strong negation, which can be compiled away in the standard way as usual.

For every logic program P, its *completion* $\mathsf{comp}(P)$ is the set of propositional formulas containing

- $body(r) \rightarrow head(r)$ for every rule r in P,
- and the formula $a \rightarrow \bigvee_{r \in P, a \in head(r)} (body(r) \wedge \bigwedge_{p \in (head(r) \setminus \{a\})} \neg p)$, for every atom a.

Here $head(r) \setminus L$ is, for any the set of atoms L, the head set of atoms that occur in $head(r)$ but not in L, and "not " and comma "," in rule bodies are translated into the negation "$\neg$" and conjunction "$\wedge$" in classical propositional logic, respectively. An empty head $head(r)$ is translated into $\perp$.

Given a logic program P, its *positive dependency graph* G_P is the directed graph whose vertices are the atoms occurring in P and where there is an edge from p to q iff there exists some rule r in P such that $p \in head(r)$ and $q \in body^+(r)$. A nonempty set $L = \{p_1, \ldots, p_k\}$ of atoms is a *loop* of P, if for every distinct $p_i, p_j \in L$ there exists a path of nonzero length from p_i to p_j in G_P such that all vertices in this path belong to L. The *conjunctive loop formula* for L is

$$CLF(L) = (p_1 \wedge \cdots \wedge p_k) \rightarrow (\bigvee_{\phi \in R(L)} \phi)$$

where $R(L)$ is the set of formulas $body(r) \wedge \bigwedge_{p \in head(r) \setminus L} \neg p$ for all rules $r \in P$ with $head(r) \cap L \neq \emptyset$ and $body^+(r) \cap L = \emptyset$.

For example, $L = \{p, q\}$ is a loop of the logic program $P = \{r \vee p \leftarrow q;\ q \leftarrow p\}$. In this example, $R(L) = \{\}$ and thus the right hand side of $CLF(L)$ is an empty disjunction. Thus $CLF(L)$ is the propositional formula $(p \wedge q) \rightarrow \perp$.

Let $\mathsf{lcomp}(P) = \mathsf{comp}(P) \cup CLF(P)$, where $CLF(P)$ is the set of all loop formulas. A fundamental result established by Lin and Zhao [50] shows that the generalized completion $\mathsf{lcomp}(P)$ exactly characterizes the answer set semantics. This was extended to disjunctive logic programs by Lee and Lifschitz [39] as follows.

Theorem 1 ([39]) *Let P be a (disjunctive) logic program and let $X \subseteq Lit_P$ be a set of atoms. Then X is an answer set of P iff X is a model of* $\mathsf{lcomp}(P)$.

Since $L = \{p, q\}$ is the only loop of $P = \{r \vee p \leftarrow q;\ q \leftarrow p\}$, we have $CLF(P) = \{(p \wedge q) \rightarrow \perp\}$. Since we have $\mathsf{comp}(P) = \{r \rightarrow q \wedge \neg q, p \rightarrow q \wedge \neg r, q \rightarrow p\}$, we obtain (after some simplifications) that $\mathsf{lcomp}(P) = \{p \equiv q, r \rightarrow \perp, p \rightarrow \neg r,$

$q \rightarrow r$, $(p \wedge q) \rightarrow \bot\}$. This theory has exactly one model, namely $\{\}$, which is the unique answer set of P.

It is easy to see that $\mathsf{lcomp}(P) = \mathsf{comp}(P)$ if P is negative. Hence,

Corollary 2 *Let Q be a negative program (without strong negation) and let $X \subseteq Lit_P$ be a set of literals. Then X is an answer set of Q if and only if X is a model of* $\mathsf{comp}(Q)$.

This corollary is a special case of some previous results in [4; 26].

Since for a logic program P and an atom p, the two classical theories $\mathsf{lcomp}(\mathsf{forget}(P, p))$ and $\mathsf{forget}(\mathsf{lcomp}(P), p)$ can be formed, where in the former logic programming forgetting is applied to P and in the latter classical forgetting to the theory $\mathsf{lcomp}(P)$, the natural question is how these two theories are related. Intuitively, the models of the first theory are all incomparable, while the models of the second theory may be not.

For example, let $P = \{p \leftarrow not\ q.\ q \leftarrow not\ p\}$. Then $\mathsf{lcomp}(\mathsf{forget}(P, p)) = \{\neg q\}$, which has the single model $\{\}$, while $\mathsf{forget}(\mathsf{lcomp}(P), p) = \{(\mathsf{T} \leftrightarrow \neg q) \vee (\mathsf{F} \leftrightarrow \neg q)\} \equiv \mathsf{T}$, which has two comparable models $\{q\}$ and $\emptyset$. However, 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 holds in general.

Theorem 3 *Let P be a consistent (disjunctive) logic program and let $p \in Lit_P$ be an atom. Then $X \subseteq Lit_P$ is an answer set of* $\mathsf{forget}(P, p)$ *iff X is a minimal model of* $\mathsf{forget}(\mathsf{lcomp}(P), p)$. *That is,*

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

where $MMod(T)$ denotes the set of all minimal models (w.r.t. $\subseteq$) of a theory T in classical logic.

The proof of this theorem is given in the Appendix. We remark that Lang et al. considered in [34] also a notion of forgetting a literal l from a formula T in classical logic, which can be semantically defined by $\mathsf{forget}(T, l) = T(p/\alpha) \vee (\neg.l \wedge T(p/\neg\alpha))$, where $\alpha = \top$ if $l = p$ is positive and $\alpha = \bot$ if $l = \neg p$ is negative. Note that forgetting an atom p as a literal is stronger than forgetting p as a propositional variable in general; e.g., forgetting p from $\neg p$ as a literal yields $\neg p$, while as a variable yields $\top$. It is easy to see that Theorem 3 remains valid for this notion of literal forgetting.

The result in Theorem 3 means that $\mathsf{forget}(P, p)$ can be characterized by forgetting in classical logic. If 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

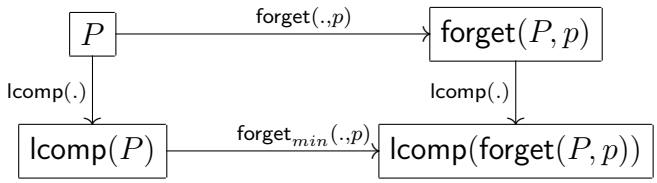

Figure 1. Commuting logic program completion and forgetting

the equation in Theorem 3 can be reformulated as

$$\mathsf{lcomp}(\mathsf{forget}(P,p)) \equiv \mathsf{forget}_{min}(\mathsf{lcomp}(P),p),$$

where $\equiv$ denotes classical equivalence. This result is graphically represented in the commutative diagram in Figure 1. It is quite useful, since it implies that one can "bypass" the use of a logic programming engine entirely and represent the answer sets of $\mathsf{forget}(P,p)$ in the frameworks of circumscription and closed world reasoning [54; 43; 27]. This can be done by applying circumscription to $\mathsf{lcomp}(P)$, which we explain in more detail.

In circumscription, minimality is understood as the impossibility of making, in the context of predicate logic, the extent of the predicates $p_1,\ldots,p_k$ which are circumscribed in a theory T smaller without changing the extent of the other predicates. Furthermore, some predicates $z_1,\ldots,z_l$ among them may be allowed to vary in the process of minimizing $p_1,\ldots,p_k$, which is needed in many applications. As for our concern of propositional logic, all p_i and z_j are atoms and $\mathsf{Circ}(T;\vec{p},\vec{z})$ is a propositional formula with quantifiers on atoms which semantically captures the circumscription of a finite theory T with respect to $\vec{p}=(p_1,\ldots,p_k)$ and $\vec{z}=(z_1,\ldots,z_l)$. As well known, $\mathsf{Circ}(T;\vec{p},\vec{z})$ is logically equivalent to Gelfond et al.'s extended closed world assumption $\mathsf{ECWA}(T;\vec{p},\vec{z})$ [27], which augments T by additional formulas.

Let us write $\mathsf{Circ}(T,z)$ respectively $\mathsf{ECWA}(T,z)$ for the case where $\vec{z}$ contains a single atom z and $\vec{p}$ all other atoms. Then, M is a model of $\mathsf{Circ}(T;z)$ respectively $\mathsf{ECWA}(T;z)$, exactly if M is a z-minimal model of T in the sense that $M' \subseteq_z M$ implies $M' \sim_z M$ for every model M' of T. We have the following result.

Theorem 4 *Let P be a consistent (disjunctive) logic program and let $p \in Lit_P$ be an atom. Then $X \subseteq Lit_P \setminus \{p\}$ is an answer set of* $\mathsf{forget}(P,p)$ *if and only if either X or $X \cup \{p\}$ is a model of* $\mathsf{Circ}(\mathsf{lcomp}(P),p)$ *(resp.,* $\mathsf{ECWA}(\mathsf{lcomp}(P),p)$*).*

(A proof is given in the Appendix.) By this rather intuitive result, we may exploit also circumscription engines for reasoning from $\mathsf{forget}(P,p)$,[2] into which we feed

[2] E.g., circ2dlp, `http://www.tcs.hut.fi/Software/circ2dlp/`, or circum1, circum2, `http://www.ailab.se.shibaura-it.ac.jp/circumscription.html`

Algorithm $\mathsf{forget}_1(P, l)$

Input: Consistent (disjunctive) logic program P and a literal l in P.

Output: A normal logic program representing $\mathsf{forget}(P, l)$.

Method:

Step 1. Compute $\mathcal{AS}(P)$ using an ASP solver (e.g., DLV or gnT).
Step 2. Remove l from every set in $\mathcal{AS}(P)$ and denote the resulting collection as $\mathcal{A}'$.
Step 3. Obtain $\mathcal{A}''$ by removing all non-minimal sets from $\mathcal{A}'$.
Step 4. Construct P' whose answer sets are exactly $\mathcal{A}'' = \{A_1, ..., A_m\}$:

- For each A_i, let $P_i = \{l' \leftarrow not\ \bar{A}_i \mid l' \in A_i\}$, where $\bar{A}_i = Lit_P \setminus A_i$.

- Let $P' = P_1 \cup \cdots \cup P_n$.

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

Figure 2. Algorithm $\mathsf{forget}_1(P, l)$

the loop completion $\mathsf{lcomp}(P)$ of P. Note that the latter is exploited by some ASP solvers (viz. ASSAT, Cmodels) as a stepping stone to compute answer sets, and that classical forgetting of p from $\mathsf{lcomp}(P)$ can be performed efficiently; indeed, the formula $\mathsf{lcomp}(P)(p/\top) \vee \mathsf{lcomp}(P)(p/\bot)$ which represents the classical forgetting $\mathsf{forget}(\mathsf{lcomp}(P), p)$, where $\mathsf{lcomp}(P)$ is viewed as a single formula and (p/α) denotes substitution of p by α, is computable in linear time. On the other hand, $\mathsf{lcomp}(P)$ is exponential in the size of P in general, as P can have exponentially many loops; see also Section 4.3.3 for further discussion regarding the size of $\mathsf{lcomp}(P)$.

4 Computation of Forgetting

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

4.1 Naive Algorithm

By Definition 2, we can easily obtain a naive algorithm for computing $\mathsf{forget}(P, l)$ using an ASP solver for logic programs, like DLV [40] or GnT [33], which is shown in Figure 2.

It is well-known that any collection $\mathcal{S}$ of sets of consistent literals which are pairwise incomparable, can be represented by some logic program P such that $\mathcal{AS}(P) = \mathcal{S}$. In fact, such P can be constructed from $\mathcal{S}$ in polynomial time.

Algorithm $\mathsf{forget}_1(P, l)$ is sound and complete w.r.t. forgetting as in Definition 2.

Theorem 5 *Given a consistent (disjunctive) logic program P and a literal l, Algorithm* $\mathsf{forget}_1(P, l)$ *outputs a correct representation of* $\mathsf{forget}(P, l)$.

The above Step 2 may return many answer sets in $\mathcal{A}'$ that are not minimal; in fact, it is easy to find examples where $\mathcal{A}'$ contains exponentially many sets but only few of them are minimal. For example, let $P = \{p \vee q \leftarrow .\ a_i \vee b_i \leftarrow q, 1 \leq i \leq n\}$. Then P has the single p-minimal answer set $\{p\}$, but exponentially many other answer sets $\{q, l_1, \ldots, l_n\}$, where l_i is either a_i or b_i, which all lead to non-minimal sets in $\mathcal{A}'$.

To avoid this problem, we present an improved version of Algorithm $\mathsf{forget}_1(P, l)$ in the next subsection.

4.2 Improved Algorithm

An improved version of Algorithm $\mathsf{forget}(P, l)$, which is shown in Figure 3, pushes the task of minimality checking for candidate l-answer sets of P into constraint satisfiability of an augmented program. It exploits in this way the constraint solving capabilities offered by some ASP solvers.

Since the answer sets of P' are exactly $\mathcal{A}$, Algorithm $\mathsf{forget}_2(P, l)$ is sound and complete w.r.t. semantic forgetting.

Theorem 6 *For every consistent (disjunctive) logic program P and a literal l, Algorithm* $\mathsf{forget}_2(P, l)$ *outputs a correct representation of* $\mathsf{forget}(P, l)$.

The advantage of Algorithm $\mathsf{forget}_2(P, L)$ is that the strategy in the Steps 1 and 2 makes l-minimization obsolete and thus no blowup into a large intermediate result with respect to the number of answer sets as in Algorithm $\mathsf{forget}_1(P, l)$ can happen. Still, however, the resulting program P' may be large compared to a small program representing $\mathsf{forget}(P, l)$ (in fact, exponentially larger), which might be constructed from P by other means (we refer here to the discussion of Algorithm $\mathsf{forget}_3(P, l)$ below). In the next subsection, we discuss how to construct a representation of $\mathsf{forget}(P, l)$ in a more syntactic manner by program transformations.

However, the "semantic" constructions by $\mathsf{forget}_1(P, l)$ and $\mathsf{forget}_2(P, l)$ also have an advantage: they clearly preserve the equivalence of logic programs under any notion of equivalence $\equiv_X$ between logic programs that is stronger than ordinary equivalence, and thus in particular under strong and weak equivalence. This is a simple consequence of the fact that for all programs P which are ordinarily equivalent the output of $\mathsf{forget}_1(P, l)$ is the same, and similarly for $\mathsf{forget}_2(P, l)$.

Algorithm $\mathsf{forget}_2(P, l)$

Input: Consistent (disjunctive) logic program P and a literal l in P.

Output: A normal logic program representing $\mathsf{forget}(P, l)$.

Method:

Step 1. Let $P_1 = P \cup \{ \leftarrow not\ l\}$.

- Compute the answer sets of P_1 (e.g., using DLV or gnT).
- Remove l from each obtained answer set. Let $\mathcal{A}_1$ be the resulting collection of sets. (Each set in $\mathcal{A}_1$ is an answer set of forgetting about l from P.)

Step 2. Let $P_2 = P \cup \{ \leftarrow l \} \cup \{ \leftarrow a_1, \ldots, a_k \mid \{a_1, \ldots, a_k\} \in \mathcal{A}_1 \}$.

- Compute the answer sets of P_2 (e.g., using DLV or gnT), and let $\mathcal{A}_2$ be the result.
- Set then $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$.

Step 3. Construct P' whose answer sets are exactly $\mathcal{A} = \{A_1, ..., A_m\}$:

- For each A_i, let $P_i = \{l' \leftarrow not\ \bar{A}_i \mid l' \in A_i\}$, where $\bar{A}_i = Lit_P \setminus A_i$.
- Let $P' = P_1 \cup \cdots \cup P_n$.

Step 4. Output P' as $\mathsf{forget}(P, l)$.

Figure 3. Improvement to Algorithm $\mathsf{forget}_1(P, l)$

4.3 Transformation-Based Algorithm

The algorithm $\mathsf{forget}_1(P, l)$ and $\mathsf{forget}_2(P, l)$ are based on the semantic view of forgetting, and do not aim at computing the result of forgetting in a more syntax-oriented manner, by modifying the rules in P. In this subsection, we present an algorithm $\mathsf{forget}_3(P, l)$ of this kind that is based on program transformations. This algorithm outputs, differently from $\mathsf{forget}_1(P, l)$ and $\mathsf{forget}_2(P, l)$, not always ordinary logic programs but sometime logic programs in which some literals are under double negation as failure. They inherit their semantics from the more general class of nested logic programs [46].

Before presenting algorithm $\mathsf{forget}_3(P, l)$, we need further preliminaries on program transformations and programs with double negation as failure.

4.3.1 Basic program transformations

We start with recalling program transformations that were discussed in [8; 66]. More precisely, we consider the following collection $\mathbf{T}^*$ of program transforma-

tions:

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 $P' = P \setminus \{r\}$.

Example 4 *Let P_1 consist of the following rules:*

$$
\begin{aligned}
r_1 :&\ p \vee p_1 \leftarrow not\ p_1 \\
r_2 :&\ p \leftarrow p, not\ q_1 \\
r_3 :&\ p \leftarrow p_1, not\ q_1 \\
r_4 :&\ p_1 \leftarrow not\ q, not\ p_2 \\
r_5 :&\ q_1 \leftarrow p_2, not\ q \\
r_6 :&\ p \vee q_1 \leftarrow \\
r_7 :&\ p_1 \vee p_3 \leftarrow \\
r_8 :&\ p_3 \leftarrow p, not\ p.
\end{aligned}
$$

Then r_2 is a tautology and thus $P_2 = \{r_1, r_3, r_4, r_5, r_6, r_7, r_8\}$ can be obtained from P by the elimination of tautologies.

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 literal l is in both $head(r)$ and $body^-(r)$ and $P' = P \setminus \{r\} \cup \{(head(r) - l) \leftarrow body(r)\}$. Here $head(r) - l$ is the disjunction obtained by removing l from $head(r)$.

By the elimination of head redundancy, r_1 is simplified into $r_1' : p \leftarrow not\ p_1$ and thus P_2 is transformed into $P_3 = \{r_1', r_3, r_4, r_5, r_6, r_7, r_8\}$.

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, $P' = P \setminus \{r\} \cup \{head(r) \leftarrow body^+(r), not\ (body^-(r) \setminus \{c\})\}$. Here $head(P) = \cup_{r \in P} head(r)$.

$P_4 = \{r_1', r_3, r_4', r_5', r_6, r_7, r_8\}$ is obtained from $P_3 = \{r_1', r_3, r_4, r_5, r_6, r_7, r_8\}$ by positive reduction, where r_4' is the rule $p_1 \leftarrow$ and r_5' is $q_1 \leftarrow p_2$.

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 $P' = P \setminus \{r\}$.

In our example, $P_5 = \{r_3, r'_4, r'_5, r_6, r_7, r_8\}$ is obtained from P_4 by negative reduction, where $r = r'_1$ and $r' = r'_4$.

For defining the next program transformation, we need the notion of *implications* for rules defined in [8]. We say r' is an implication of r if $head(r) \subseteq head(r')$, $body(r) \subseteq body(r')$ and at least one of the inclusions is strict.

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 $P' = P \setminus \{r'\}$.

In the above example, r_7 is an implication of r'_4 and thus $P_6 = \{r_3, r'_4, r'_5, r_6, r_8\}$ is obtained from P_5 by the elimination of implications.

Elimination of Contradictions P' is obtained from P by elimination of contradictions if there is a rule r in P such that $body^+(r) \cap body^-(r) \neq \emptyset$ and $P' = P \setminus \{r\}$.

By the elimination of contradictions, r_8 can be removed from P_6 and thus we obtain $P_7 = \{r_3, r'_4, r'_5, r_6\}$.

Unfolding P' is obtained from P by unfolding if there is a rule r with $body^+(r) \neq \emptyset$ such that

$$\begin{aligned} P' = P \setminus \{r\} & \\ \cup \{\ H(r,r',b) \leftarrow B(r,r',b) \ \mid\ & b \in body^+(r),\ r' \in P,\ b \in head(r'), \\ r' \neq r \qquad & H(r,r',b) = head(r) \cup (head(r') \setminus \{b\}), \\ & B(r,r',b) = body^+(r) \setminus \{b\}, not\ body^-(r), body(r') \quad \}. \end{aligned}$$

P_7 is further transformed into $P_8 = \{r'_3, r'_4, r_6\}$ where $r'_3 : p \leftarrow not\ q_1$ is obtained from P_7 by unfolding r_3 and r'_4, and in particular, r'_5 is removed as a special case of the unfolding since there is no rule in P_7 whose head can be unfolded with the positive body literal p_2 of r'_5.

Note that the above example is only used to illustrate the definitions of program transformations. In practice, one could make the process of simplifying P_1 more efficient by choosing different transformations and different orderings.

As shown by [7], the program transformations from $\mathbf{T}^*$ considered there, i.e., all except the elimination of head redundancy, preserve the answer set semantics. The same also holds for the latter transformation.

Proposition 9 *Let P be a disjunctive program. If P' is obtained by the elimination of head redundancy from P, then P and P' have the same answer sets.*

Furthermore, by [8] every disjunctive program can be converted with the transformations in $\mathbf{T}^*$ considered there into an equivalent negative disjunctive program. Moreover, by using the elimination of tautologies and the elimination of head redundancy (which preserves negative rules), every literal that occurs both in the head and the body of a rule can be removed. Thus, we have the following result.

Lemma 1 *Every logic program P can be transformed into an equivalent negative program N via $\mathbf{T}^*$ such that every rule r in N fulfills $head(r) \cap body(r) = \emptyset$.*

In fact, transformations can be applied in arbitrary manner and no backtracking is necessary to construct such a negative program N (i.e., choices for transformations are "don't care"). In addition, we introduce only basic program transformations here. One can introduce some other program transformations, such as the elimination of s-implications in [66], to further simplify the negative program obtained from basic transformations.

4.3.2 Logic programs with double negation

A *disjunctive logic program with double negation as failure* (*DDLP*) is a finite set of rules r of the form

$$a_1 \vee \cdots \vee a_s \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n, not\ not\ d_1, \ldots, not\ not\ d_t \quad (2)$$

where $s, m, n, t \geq 0$ and all a_i, b_j, c_k, and d_l are from a set Lit of classical literals; as above, we assume that all a_i, and similarly all b_j, all c_k, and all d_l, are pairwise distinct. (Note that form (1) results for $t = 0$.) The definition of $head(r)$, $body^+(r)$, and $body^-(r)$ is analogous to ordinary rules (1), and $body^{--}(r) = \{d_1, \ldots, d_t\}$. Thus, r can be denoted $head(r) \leftarrow body^+(r), not\ body^-(r), not\ not\ body^{--}(r)$, where $not\ not\ body^{--}(r) = \{not\ not\ l \mid l \in body^{--}(r)\}$.

Every DDLP P is a nested logic program [46], and inherits answer set semantics from such programs. Formally, the *reduct* of P w.r.t. an interpretation X is defined as $P^X = \{head(r) \leftarrow body^+(r) \mid r \in P, body^-(r) \cap X = \emptyset, body^{--}(r) \subseteq X\}$. As usual, X is an *answer set* of P iff X is a minimal model of P^X.

Different from ordinary logic programs, the answer sets of DDLP may be comparable. For example, the DDLP $P = \{p \leftarrow not\ not\ p\}$ has the two answer sets $\emptyset$ and $\{p\}$. For our purposes, we will use those answer sets of P which are minimal.

Clearly, all answer sets of P are minimal iff they are all incomparable, and the latter is equivalent to the fact that P can be rewritten to an ordinary logic program. While this is difficult and expensive to check in general,[3] there is an attractive syntactic

[3] More precisely, this problem is Π_2^p-complete, as two comparable answer sets of P can be guessed and checked in polynomial time with an NP oracle. The Π_2^p-hardness can be

class for which a simple rewriting exists.

Similar as for logic programs with default negation in rule heads [32], double negations can be safely eliminated from a DDLP without changing its semantics, if there is no cycle through positive and double negated dependencies.

Definition 3 *A DDLP P is* N-acyclic*, if there is a level mapping $L : Lit_P \rightarrow \{0, 1, \ldots\}$ of the literals in P to the non-negative integers such that for every rule r in P, the following two conditions hold:*

(i) $L(l) \geq L(l')$ *for all* $l \in head(r)$ *and* $l' \in body^+(r)$.
(ii) $L(l) > L(l')$ *for all* $l \in head(r)$ *and* $l' \in body^{--}(r)$.

Note that there are no conditions $L(l) > L(l')$, for all $l \in head(r)$ and $l' \in body^-(r)$, and $L(l) = L(l')$, for all literals $l, l' \in head(r)$ as in the familiar definition of stratified logic programs. By exploiting standard methods for testing whether a logic program is stratified (cf. [38]), one can efficiently decide whether a given DDLP program P is N-acyclic in linear time in the size of P.

Given a DDLP P, let $\mathcal{T}(P)$ be the logic program obtained from P by canceling every double negation $not\, not$ in it. For example, if $P_0 = \{p \leftarrow q, not\ q', not\, not\, q''\}$, then $\mathcal{T}(P_0) = \{p \leftarrow q, not\ q', q''\}$. We have the following result.

Theorem 7 *For every N-acyclic DDLP P, it holds that $\mathcal{AS}(P) = \mathcal{AS}(\mathcal{T}(P))$.*

(See the Appendix for the proof.) Note that since every ordinary logic program is trivially N-acyclic, we can view N-acyclic DDPs as a syntactic extension of ordinary logic programs. The fact that N-acyclic DDLPs can be easily cast to ordinary logic programs will be used for transformation-based forgetting.

4.3.3 The algorithm

We are now in a position to present a syntax-based algorithm for computing forgetting in a logic program. The algorithm $\mathsf{forget}_3(P, l)$, which is shown in Figure 4, first translates the input program P into a negative program N (Step 1) and then separates l from head disjunction (Step 2). After that, l is eliminated from rule bodies (Step 3), and finally from rule heads (Step 4). The resulting output program is, in general, a logic program with double negation as failure.

Example 5 *Consider $P_4 = \{c \leftarrow not\ q.\ \ p \leftarrow not\ q.\ \ q \leftarrow not\ p\}$. Then, in Step 1 we have $N_0 = P_4$ since P_4 is already negative, and in Step 2 $N = P_4$ since P_4 is normal. In Step 3, we have $n = 1$ and $r_1 = p \leftarrow not\ q$. Thus, case three applies*

shown by an easy reduction from deciding whether a given logic program (without strong negation) has no answer set, which is Π_2^p-complete [22].

Algorithm $\mathsf{forget}_3(P, l)$

Input: Consistent (disjunctive) logic program P and a literal l in P.

Output: DDLP logic program N' whose minimal answer sets are $\mathcal{AS}(\mathsf{forget}(P, l))$.

Method:

Step 1. Apply program transformations in $\mathbf{T}^*$ on P to obtain a negative program N_0.
Step 2. Separate l from head disjunction via semi-shifting:

- Replace each rule $r \in N_0$ such that $head(r) = l \vee l_1 \vee \cdots \vee l_k$, where $k \geq 1$, by the two rules $l \leftarrow not\ l_1, \ldots, not\ l_k, body(r)$ and $l_1 \vee \cdots \vee l_k \leftarrow not\ l, body(r)$.

Let N be the resulting logic program.
Step 3. Suppose that $r_1, \ldots, r_n$ are the rules in N with head l, where $r_j : l \leftarrow not\ l_{j1}$, ..., $not\ l_{jm_j}$ and $m_j \geq 0$ for $1 \leq j \leq n$. Distinguish three cases:

3.1 If $n = 0$, then obtain the program Q by removing in N all literals $not\ l$.

3.2 If $n > 0$ and $m_j = 0$ for some $1 \leq j \leq n$ (i.e., $l \leftarrow$ is a rule in N), then obtain the program Q by removing from N all rules whose bodies contain $not\ l$.

3.3 If $n > 0$ and $m_j > 0$ for all $1 \leq j \leq n$, let $D_1, \ldots, D_s$ be all possible conjunctions of form $not\ not\ l_{1k_1}, \cdots, not\ not\ l_{nk_n}$ where $0 \leq k_1 \leq m_j$, $1 \leq j \leq n$.

Obtain the program Q by replacing in N each $not\ l$ by all D_i (one at a time).

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

Figure 4. Syntax-based algorithm to compute forgetting

and we have $D_1 = not\, not\, q$*; we obtain* $Q = \{c \leftarrow not\ q.\ p \leftarrow not\ q.\ q \leftarrow not\ not\ q\}$*. In Step 4, the program* $N' = \{c \leftarrow not\ q.\ q \leftarrow not\, not\, q\}$ *is output. This program has the answer sets* $\{c\}$ *and* $\{q\}$*, which are both minimal. They are the same as the answer sets of* $\mathsf{forget}(P_4, p)$.

Note that Algorithm 1 in [65] outputs on the input of P_4 and p the program $N' = \{c \leftarrow not\ q.\ q \leftarrow q\}$, which has the single answer set $\{c\}$. However, the semantic result of forgetting about p in P_4, as defined [65] and in this paper, has another answer set, viz. $\{q\}$. This shows that Algorithm 1 in [65] is incomplete, i.e., outputs in general a logic program that represents only a subset of all answer sets after forgetting.

Several remarks on Algorithm $\mathsf{forget}_3(P, l)$ are in order.

(1) As formulated here, the algorithm is stated in a very general form. A number of refinements and improvements can be made in order to make it more efficient and the result more compact. For example, in Step 1 some program transformations could be omitted for special programs and various heuristics could also be

employed. In Step 3, only those D_i need to be considered which, after removal of duplicate literals, are not properly contained in some other D_j. To compute them, one can use efficient hypergraph transversal algorithms (see [25]).

(2) In the construction of D_i, $not\, not\, l_{ij}$ cannot be replaced with l_{ij} (even for a normal logic program). As shown by Example 5, the resulting output program $\{c \leftarrow not\ q.\ q \leftarrow q\}$ would only represent a subset of $\mathcal{AS}_l(P)$, and thus would be incorrect. The use of double negation as failure, which remedies this problem, seems to be intuitive. It remains as an interesting issue whether this can be avoided in a similar transformation based algorithm.

(3) The running time of algorithm $\mathsf{forget}_3(P, l)$ is worst case exponential, and the output program may be exponentially large. As follows from complexity considerations in Section 5, there is no ordinary or nested logic program P' representing $\mathsf{forget}(P, l)$ which can be constructed in polynomial time, even if auxiliary literals might be used which are projected from the answer sets of P'.

(4) In essence, algorithm $\mathsf{forget}_3(P, l)$ improves the corresponding Algorithm 1 in [65] in at least two ways: it works for the more expressive class of disjunctive logic programs, and importantly, its output correctly represents the result of forgetting. This is shown formally by the following result.

Theorem 8 *Let P be a consistent disjunctive logic program and let $l \in Lit_P$ be a literal. Then $\mathsf{forget}_3(P, l)$ correctly represents $\mathsf{forget}(P, l)$, i.e., X is an answer set of $\mathsf{forget}(P, l)$ iff X is a minimal answer set of N'.*

(For a proof, see the Appendix.) While the output program of $\mathsf{forget}_3(P, l)$ generally contains double negation as failure and its minimal answer sets have to be considered, Theorem 7 provides a simple condition for transforming the output of Algorithm $\mathsf{forget}_3(P, l)$ into an ordinary logic program in some cases.

Proposition 10 *Let P be a consistent logic program, let $l \in Lit_P$ be a literal, and let N' be the output of Algorithm $\mathsf{forget}_3(P, l)$. If N' is N-acyclic, then $\mathcal{AS}_l(P) = \mathcal{AS}(\mathcal{T}(P'))$, i.e., the ordinary logic program $\mathcal{T}(P')$ correctly represents $\mathsf{forget}(P, l)$.*

For arbitrary inputs, it is not clear whether the output of $\mathsf{forget}_3(P, l)$ is an N-acyclic program. We note here a relevant subclass which has this property, given by a simple syntactic condition that can be efficiently recognized.

Recall that the standard dependency graph of a program P, denoted DG_P, has Lit_P as vertices and a positive edge from literal l to literal l' if $l \in head(r)$ and $l' \in body^+(r) \cup (head(r) - l)$ for some rule $r \in P$ and a negative edge from l to l' if $l \in head(r)$ and $l' \in body^-(r)$ for some rule $r \in P$. A (directed) cycle in the graph DG_P is negative, if it contains at least one negative edge.

Proposition 11 *Let P be a consistent normal logic program and let $l \in Lit_P$ be a literal. If no negative cycle of DG_P contains l, then* $\mathsf{forget}_3(P, l)$ *outputs an N-acyclic program.*

In Example 2, for $l = p$ the programs P_1, P_3, and P_4 have this property (as well as P_6 after removal of the redundant disjunctive rule). Hence, the outputs of the respective calls $\mathsf{forget}_3(P_i, p)$ can be cast to ordinary logic programs.

Unfortunately, the extension of Proposition 11 from normal to disjunctive logic programs fails, even for the simple case where $P = \{p \vee q \leftarrow .\}$ and $l = p$. However, it is possible to single out fragments for which this is possible, which we leave for future work.

Computing a representation of $\mathsf{forget}(P, l)$ can be refined in different directions. One is to localize the computation, such that only a relevant part of the program P is subject to modification when forgetting a literal l, while the rest of P remains untouched. This is, for example, easy if P is a normal logic program; in that case, merely the value of l in the single answer set of P has to be plugged in for all occurrences of l in P and and the resulting program be simplified.

Another case is if P splits into separate components $P_1 \cup \cdots \cup P_n$ such that the answer sets of P can be obtained by combining the local answer sets of each P_i.

To this end, we call a literal l' in a program P *unaffected* by the forgetting of literal l, if there is no path between l and l' in the undirected version of the dependency graph DG_P. Suppose $U(P, l)$ is the set of all rules in P that only involve such literals l'. Then the following easy property holds.

Proposition 12 *For every (consistent) program P and literal l, it holds that* $\mathsf{forget}(P, l) \equiv U(P, l) \cup \mathsf{forget}(P \setminus U(P), l)$.

This property is in line with N-acyclicity: disconnected components cannot destroy the N-acyclicity of the (rewritten) program. Furthermore, it can be combined with Proposition 11 to enlarge the classes of programs P for which $\mathsf{forget}(P, l)$ is representable by an ordinary logic program.

Note that Proposition 12 is independent of the concrete syntactic form of $\mathsf{forget}(P, l)$; it may well be generalized for specific such forms and/or classes of programs. For example, if P can be split into programs P_1 and P_2 such that l occurs only in P_1, program P_2 has a single answer set S (e.g. if P_2 is normal and stratified), and no head of a rule in P_1 occurs in P_2, then

$$\mathsf{forget}_3(P, l) \equiv \mathsf{forget}_3(P_1 \cup \{l' \leftarrow . \mid l' \in S\}, l) \cup P_2.$$

A detailed study of this issue remains for future work.

We close this section with a brief comparison of the representation size of $\mathsf{forget}(P, l)$ in terms of the programs output by the algorithms $\mathsf{forget}_1(P, l)$, $\mathsf{forget}_2(P, l)$, and $\mathsf{forget}_3(P, l)$, and the formula $\mathsf{Circ}(\mathsf{lcomp}(P), l)$ from Section 3.3, where l is an atom p and $\mathsf{Circ}(\mathsf{lcomp}(P), l)$ is viewed as a formula with quantifiers as usual (cf. [42; 43]); its size is polynomial in the size of $\mathsf{lcomp}(P)$. Note that $\mathsf{forget}_1(P, l)$ and $\mathsf{forget}_2(P, l)$ have always the same output.

Compared to the algorithms $\mathsf{forget}_1(P, l)$ and $\mathsf{forget}_2(P, l)$, the size of the output of $\mathsf{forget}_3(P, l)$ is orthogonal, in the sense that there instances where the former produce an output program that is exponentially smaller respectively larger than the one produced by $\mathsf{forget}_3(P, l)$.

For example, the program $P = \{a_i \leftarrow not\ b_i \mid 1 \leq i \leq n\} \cup \{p \leftarrow\}$ has exponentially many answer sets and each contains p; the output of $\mathsf{forget}_1(P, p)$ and $\mathsf{forget}_2(P, p)$ thus contains exponentially many rules, while $\mathsf{forget}_3(P, p)$ consists of $P \setminus \{p \leftarrow\}$. On the other hand, the program $Q = \{p_i \leftarrow not\ p.\ q_i \leftarrow not\ p.\ p \leftarrow not\ p_i, not\ q_i \mid 1 \leq i \leq n\}$, has two answer sets, viz. $\{p\}$ and $\{p_1, q_1, \ldots, p_n, q_n\}$, and thus $\mathsf{forget}(Q, p)$ has the empty answer set; here, $\mathsf{forget}_1(Q, p)$ and $\mathsf{forget}_2(Q, p)$ consist of a single rule, while $\mathsf{forget}_3(Q, p)$ contains exponentially many rules of the form $p_i \leftarrow not\ not\ a_1, \ldots, not\ not\ a_n$ and $q_i \leftarrow not\ not\ a_1, \ldots, not\ not\ a_n$, where each a_j is either p_j or q_j, for all $i = 1, \ldots, n$.

Also the representation of $\mathsf{forget}(Q, p)$ in terms of minimal models described in Section 3.3, $\mathsf{Circ}(\mathsf{lcomp}(Q), p)$, is exponentially smaller than $\mathsf{forget}_3(Q, p)$: as Q is negative, $\mathsf{lcomp}(Q) = \mathsf{comp}(Q)$. On the other hand, for the program P above, $\mathsf{lcomp}(Q)$ is like $\mathsf{forget}_3(P, p)$ small while $\mathsf{forget}_1(P, p)$ and $\mathsf{forget}_2(P, p)$ are exponentially larger. Finally, it is easy to find programs P such that, for some atom p, $\mathsf{forget}_3(P, p)$ is exponentially smaller than $\mathsf{lcomp}(P)$, if no optimization are applied. this also holds if P is negative (where $\mathsf{lcomp}(P)$ is easily formed.

Thus in summary, the three representations obtained by $\mathsf{forget}_1(P, l)$/$\mathsf{forget}_2(P, l)$, $\mathsf{forget}_3(P, l)$, and $\mathsf{Circ}(\mathsf{lcomp}(P), p)$ are, in their basic form, orthogonal in size.

5 Computational Complexity

In this section, we address the computational complexity of forgetting for different classes of logic programs. Our main complexity results for forgetting are compactly summarized in Table 1. They show that for general logic programs, (1) model checking under forgetting is Π_2^p-complete; (2) credulous reasoning under forgetting is Σ_3^p-complete; and (3) skeptical reasoning under forgetting is Π_2^p-complete.

Intuitively, this complexity is explained by two respectively three intermingled

program P	disjunctive	negative	normal
$X \in \mathcal{AS}_l(P)$? (model checking)	Π_2^p	co-NP	co-NP
$\mathsf{forget}(P, l) \models_c l'$?	Σ_3^p	Σ_2^p	Σ_2^p
$\mathsf{forget}(P, l) \models_s l'$?	Π_2^p	co-NP	co-NP

Table 1
Complexity of forgetting (entries are completeness results; l, l' are literals, P is a logic program, X is a set of literals)

sources of complexity (i)–(iii): For problem (1), given an answer set S of a program P and a literal l, (i) the number of candidate answer sets S' such that $S' \subset_l S$ and (ii) the test whether a given such S' is in fact an answer set of P; for problem (2), in addition (iii) the number of candidate l-answer sets S containing the query literal l'. Note, however, that for problem (3) source (i) is absent (by Proposition 7) and only (ii) and (iii) (in dual form) are present, causing the same complexity as for standard skeptical reasoning (without forgetting).

For normal programs and negative logic programs, the complexity of all problems is lowered by one level of the Polynomial Hierarchy. Intuitively, the reason is that source (ii), i.e., model checking for such programs, is polynomial in both cases.

In the rest of this section, we state and develop the complexity results formally, and also argue that space-efficient representations of $\mathsf{forget}(P, l)$ in terms of ordinary (disjunctive) logic programs are unlikely to exist. The design of Algorithm $\mathsf{forget}_3(P, l)$ in Section 4 is heavily influenced by the complexity analysis.

Theorem 9 *Given a consistent (disjunctive) logic 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 9. The hardness result is shown by a reduction from deciding whether a given logic program P (without strong negations) has no answer set, which is Π_2^p-complete [22].

If P is either negative or normal, l-answer checking is co-NP-complete.

Theorem 10 *Given a consistent normal logic 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.*

The proof of this theorem exploits that the reduction in the proof of Theorem 9 still works for normal programs, and that deciding whether a normal logic program has an answer set is well-known to be NP-complete [52; 4].

Using a minor modification of the reduction in the proof of Theorem 9, we can show the co-NP-completeness for negative programs. Notice that, as already mentioned, deciding whether a given set of literals is an answer set of negative program is feasible in polynomial time, which explains the complexity drop.

Theorem 11 *Given a consistent negative 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.*

The following theorem shows that credulous reasoning with forgetting has a higher complexity.

Theorem 12 *Given a consistent (disjunctive) logic program P and literals l and l', deciding whether* $\mathsf{forget}(P, l) \models_c l'$ *is Σ_3^p-complete.*

In Theorem 12, a suitable l-answer set containing l' can be guessed and checked, by Theorem 9 using Σ_2^p-oracle. Hence, credulous inference $\mathsf{forget}(P, l) \models_c l'$ is in Σ_3^p. The Σ_3^p-hardness is shown by an encoding of quantified Boolean formulas (QBFs) of the form $\exists Z \forall X \exists Y \phi$.

The construction in the proof of Theorem 11 can be lifted to show that credulous inference with forgetting is Σ_2^p-complete for negative programs.

Theorem 13 *Given a consistent negative program P and literals l and l', deciding whether* $\mathsf{forget}(N, l) \models_c l'$ *is Σ_2^p-complete.*

In fact, the program constructed to show the hardness part of this result is normal. Therefore, we easily derive the following result.

Theorem 14 *Given a consistent normal program N and literals l and l', deciding whether* $\mathsf{forget}(N, l) \models_c l'$ *is Σ_2^p-complete.*

The complexity results for skeptical reasoning with forgetting are straightforward from Proposition 7 and well-known results about the complexity of normal logic programs (see [22; 52; 15]).

Theorem 15 *Given a consistent logic program P and literals l and l', deciding whether* $\mathsf{forget}(P, l) \models_s l'$ *is (i) Π_2^p-complete for arbitrary disjunctive logic programs P, and (ii) co-NP-complete for normal logic programs and for negative logic programs P.*

By applying techniques that build on non-uniform complexity classes from [12], one can show that for a given (disjunctive) program P and literal l there is generally no ordinary disjunctive program P' representing $\mathsf{forget}(P, l)$ that has size polynomial in the size of P, unless the Polynomial Hierarchy collapses. This remains true even if auxiliary literals might be used in P' for the representation which are projected off the models of P' to obtain the models of $\mathsf{forget}(P, l)$. This means that the

exponential blow up of $\mathsf{forget}(P, l)$ is, in a formal sense, unavoidable in general.

More precisely, it can be shown that the model checking problem for forgetting is complete for the complexity class $\|\!\!\rightsquigarrow\Pi_2^p$ defined in [12; 13]. Informally, this means that problem is among the hardest in $\|\!\!\rightsquigarrow\Pi_2^p$, which contains those problems that are decidable in Π_2^p with preprocessing of the input (which depends on a "fixed" part and the size of the input). The preprocessing can resort to precompiled knowledge in polynomial-size data structures, where the compilation cost does not count. Technically, the Π_2^p-completness of model checking for forgetting implies that (a syntactic variant of) the problem belongs to $\|\!\!\rightsquigarrow\Pi_2^p$. On the other hand, the problem is $\|\!\!\rightsquigarrow\Pi_2^p$-hard, as evaluting a QBF of the form $\forall X \exists Y \phi$ can be reduced (under the suitable notion of reduction) to model checking for forgetting. The proof is similar in spirit to the one of Theorem 3.2 in [13], which shows $\|\!\!\rightsquigarrow\Pi_2^p$ hardness of clause inference from the minimal models of a propositional CNF, but uses the encoding of QBFs into model checking for forgetting given in the proof of Theorem 9 via [22]. We refrain here from further details.

Now while model checking for forgetting is $\|\!\!\rightsquigarrow\Pi_2^p$-hard, model checking for ordinary disjunctive programs is well-known to be in co-NP (cf. [21; 15]). From Theorem 5 in [12], it follows that for arbitrary logic programs P and literal l there exists no representation of $\mathsf{forget}(P, l)$ by an ordinary (disjunctive) logic program of size polynomial in the size of P unless the Polynomial Hierarchy collapses (which is considered to be unlikely).

Analogously, one can show that for normal programs P, $\mathsf{forget}(P, l)$ is not representable by normal programs of polynomial size in the size of P, unless the Polynomial Hierarchy collapses, again even if auxiliary literals might be used as above.

However, we point out that if auxiliary literals would be allowed, then we can represent $\mathsf{forget}(P, l)$ in terms of the minimal answer sets of a polynomial-size logic program with double negation as failure. More precisely, let s and p_l be fresh auxiliary literals, and use l itself as an auxiliary literal. Let

- P_1 be the program resulting from P by adding $not\ s$ in each rule body, let
- P_2 be the program resulting from P by replacing each occurrence of l with p_l and by adding s in each rule body, and let
- $Q = P_1 \cup P_2 \cup \{s \leftarrow not\ not\ s.\ \ l \leftarrow s.\quad \leftarrow not\ l, not\ s.\quad \leftarrow p_l, s\}$.

Note that Q is easily constructed from P and l (in linear time). Informally, s is a switch between P_1 and P_2, to compute the answer sets of P where l is true (via P_1 and the constraint $\leftarrow not\ l, not\ s$, when s is false) respectively false (via P_2, when s is true). In order to make these answer sets l-comparable, in the computations of P_2 the literal p_l replaces l and l is artificially included (by $l \leftarrow s$) while p_l is excluded (through $\leftarrow p_l, s$). The answer sets S of Q (which all contain l and but not p_l) correspond then one-to-one to the answer sets of P. Now every S such that $s \notin S$ is a minimal answer set of Q, while if $s \in S$, then S is minimal iff it contains

no answer set S' of Q such that $s \notin S'$ properly. Consequently, the minimal answer sets S of Q encode the l-answer sets of P, and thus the answer sets of $\mathsf{forget}(P, l)$, which are given by $S \setminus \{l, s\}$.

6 Applications

In this section, we present some applications of the results on forgetting from logic programs in the previous sections. In particular, we consider applications to conflict resolution in logic-based multi-agent systems, which is obviously an important task, to inheritance logic programs, which model objects and classes with inheritance of properties, and to logic program updates. A further application in the area of ontology merging and alignment (in an extended framework), is described in [24]. The applications show the usefulness of our results about forgetting in different respects: On the one hand, for finding novel solutions to problems (like in conflict resolution and ontology merging), and on the other hand, to obtain novel characterizations (and thus interpretations) of existing concepts (like for inheritance logic programs and logic program updates).

6.1 Resolving Conflicts in Multi-Agent Systems

As the first application, we present a general framework for resolving conflicts in multi-agents systems, which is inspired from the *preference recovery* problem [35]. In particular, an example is given to show the elegance of using the semantic forgetting in answer set programming to solve the problem of preference recovery for multi-agents.

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 intuitive criteria such that preferences with higher priorities are satisfied. Consider the following example.

Example 6 ([35]) *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 and find possible agreements with minimal costs using the theory of forgetting.

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

As shown in Example 6, $P_1 \cup P_2 \cup \cdots \cup P_n$ 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 logic program* $\mathsf{forget}(\mathcal{S}, C)$ *where* $\mathsf{forget}(\mathcal{S}, C) = \mathsf{forget}(P_1, F_1) \cup \mathsf{forget}(P_2, F_2) \cup \cdots \cup \mathsf{forget}(P_n, F_n)$.

Intuitively, the set F_i in a compromise contains those aspects which agents i does not care much about. For a specific application, we may need to impose certain conditions on each F_i.

Example 7 *(Example 6 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*

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

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 $F = \{s, blue, red\}$: 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* $F = \{u_0, u_1, u_2, blue, red\}$*: 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* $\{s, t, red\}$ *and* $\{s, t, blue\}$ *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) $C_3 = (\emptyset, \{blue, red\}, \emptyset, \emptyset, \{t\})$*: 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* $\{s, blue, u_1\}$ *on* C_3*. That is, only a swimming pool will be built and its colour is blue.*

(4) $C_4 = (\emptyset, \{blue, red\}, \{blue, red\}, \{s, t\}, \{s, t\})$*: The first and second residents can weaken her preference on pool colour; the third and fourth residents would not mind if tennis court or swimming pool is built. Since* $\mathsf{forget}(\mathcal{S}, C_4) = P_0 \cup \{u_0 \vee u_1 \leftarrow .\ s \vee t \leftarrow\}$*,* $\mathcal{S}$ *has three possible agreements* $\{u_1, t\}$*,* $\{u_1, s, blue\}$*,* $\{u_1, s, red\}$ *on* C_4*.*

It should be noted that a solution to this problem is also provided in [35] where the forgetting for propositional logic is used where a theory in propositional logic rather than possible agreements are produced. A model of that theory may not represent an agreement in the sense of Definition 4. For example, the solution for C_4 given in [35] is the theory $T_4 = \{s \rightarrow red \vee blue, (red \wedge blue) \rightarrow \bot, (\neg s \wedge t) \rightarrow u_1, (s \wedge \neg t) \rightarrow u_1, (s \wedge t) \rightarrow u_2, (\neg s \wedge \neg t) \rightarrow u_0\}$ and its models also include some agreements that have non-minimal costs (e.g. $\{u_2, s, t, red\}$ is a model of T_4).

However, in our approach each answer set corresponds to exactly one agreement with minimal cost. In addition, the issue of resolving conflicts in multi-agent systems has been challenging and numerous proposals have been suggested for different systems (for example, see [64]). Thus it would be interesting to explore applications of our technique in practical multi-agent systems.

6.2 Inheritance Logic Programs

In this section, we investigate relationships between forgetting in logic programs and inheritance logic programs [11]. As we show, the semantics of such programs can be expressed by forgetting from a logic program.

Let P be a logic program with classical negation and each rule r of P is labeled with either the symbol '.' or the symbol '!'. The symbol '.' means that r is a *defeasible* rule and the symbol '!' means that r is a *strict* rule. In the approach proposed in [11], P is an *inheritance program* if P is classified into different objects and an

object may have higher priority than another object. For any two objects o_1 and o_2, $o_1 < o_2$ denotes that o_1 has higher priority over o_2. This priority relation naturally defines a priority for rules in these two objects: $r_1 < r_2$ if $r_1 \in o_1$, $r_2 \in o_2$ and $o_1 < o_2$.

Example 8 *Let $(P, <)$ be an inheritance logic program that consists of three objects: o_1, o_2, o_3 where*

$$o_1 = \{\ penguin(Tweety) \leftarrow!\ \}$$
$$o_2 = \{\ bird(x) \leftarrow penguin(x)!,\ \ \neg flies(x) \leftarrow penguin(x).\ \}$$
$$o_3 = \{\ flies(x) \leftarrow bird(x).\ \}.$$

$o_1 < o_2 < o_3$ since more specific rules have higher priority.

In the rest of this section, we view in accordance with [11] inheritance programs as pairs $(P, <)$, where P is a ground (propositional) logic program with classical negation and $<$ is a strict preorder (irreflexive and transitive relation) on the rules in P, such that $r < r'$ iff r has higher priority than r'. [4]

The semantics of inheritance programs is defined in terms of *inheritance answer sets*, which are based on the notion of *models* of an inheritance program. The notion of satisfiability of rules for inheritance programs encodes priority information and thus is quite different from the traditional notion.

Given two (ground) rules r_1 and r_2, we say r_1 *threatens* r_2 *on a literal* l if (1) $\neg.l \in head(r_1)$, (2) $r_1 < r_2$, and (3) r_2 is defeasible (recall that $\neg.l$ denotes the complement of literal l).

Definition 5 *Given an inheritance program $(P, <)$ and an interpretation S, a rule* r_1 overrides r_2 *in S if (1) r_1 threatens r_2 on a literal l, (2) $\neg.l \in S$ and (3) $S \models body(r_2)$. A rule r is* overridden *in S if for each $l \in head(r)$ there exists a rule r' in P such that r' overrides r on l in S.*

Informally, a rule r is overridden by another rule r', if it has lower priority than r' and is in conflict with r'. Obviously, a strict rule cannot be overridden.

An interpretation S is a model of $(P, <)$, if every rule in P is either satisfied or overridden in S. The Gelfond-Lifschitz reduct is extended to inheritance programs as follows.

Definition 6 *Given an inheritance program $(P, <)$ and an interpretation S, the* reduction *of $(P, <)$ w.r.t. S, denoted $(P, <)^S$, is the set of rules obtained from P by*

[4] Cf. [11, p.297] for technical assumptions to ensure this.

removing (1) every rule overridden in S, (2) every rule r such that $body^-(r) \cap S \neq \emptyset$, and (3) $body^-(r)$ from each remaining rule r.

An interpretation S is an inheritance answer set of $(P, <)$, if S is a minimal model of $(P, <)^S$.

Example 9 *The ground version of the inheritance program in Example 8, $(ground(P), <)$, has the single inheritance answer set $S = \{penguin(Tweety), bird(Tweety), \neg flies(Tweety)\}$. Indeed, the instance of the rule in o_3, $flies(Tweety) \leftarrow bird(Tweety)$, is overridden by the rule instance $\neg flies(Tweety) \leftarrow penguin(Tweety)$ from o_2 in S; $(ground(P), <)^S$ consists of the three rules $penguin(Tweety) \leftarrow$, $bird(Tweety) \leftarrow penguin(Tweety)$, and $\neg flies(Tweety) \leftarrow penguin(Tweety)$. Clearly, S is their unique answer set, which means that S is an inheritance answer set of $(ground(P), <)$.*

Let $(P, <)$ be an inheritance program and S be a set of literals. We introduce a new literal l' for each literal l in P. For each rule r in P, if r is overridden in S, then every literal l in $head(r)$ is replaced with l'. The resulting program from P is denoted P'. Let $F = \{l' \mid l'$ is a new literal and $l' \in head(r')$ for some $r' \in P'\}$.

The following theorem provides a semantic characterization of inheritance programs in terms of semantic forgetting.

Theorem 16 *Let $(P, <)$ be an inheritance program and let S be a set of literals. Then S is an inheritance answer set of $(P, <)$ iff S is an answer set of $\mathsf{forget}(P', F)$ where P' is obtained as above.*

Example 10 *Continuing our birds example, for $S = \{penguin(Tweety), bird(Tweety), \neg flies(Tweety)\}$ the corresponding ordinary logic program $ground(P)'$ consists of the following rules:*

$$
\begin{aligned}
penguin(Tweety) &\leftarrow , \\
bird(Tweety) &\leftarrow penguin(Tweety), \\
\neg flies(Tweety) &\leftarrow penguin(Tweety), \\
flies(Tweety)' &\leftarrow bird(Tweety).
\end{aligned}
$$

For $F = \{flies(Tweety)'\}$, we obtain that $\mathsf{forget}(ground(P)', F)$ is represented by the first three rules above. This program has the unique answer set S, as stated by Theorem 16.

The proof of Theorem 16 is based on the following result, which is of independent interest.

Proposition 13 *Let P be a logic program and let F be a consistent set of literals.*

Suppose that (1) no literal in F occurs in a rule body in P, and (2) for each rule r, either no or every literal in $head(r)$ is in F. Then $\mathsf{forget}(P, F) = P \setminus R(F)$*, where* $R(F) = \{r \in P \mid r$ *contains a literal of* $F\}$.

Note that the conclusion of Proposition 13 may not be true if F contains opposite literals. For example, consider the logic program $P = \{p \leftarrow a;\ \neg p \leftarrow a;\ a \leftarrow not\ b;\ b \leftarrow not\ a\}$ and $F = \{p, \neg p\}$. Then $P \setminus R(F) = \{b \leftarrow not\ a;\ a \leftarrow not\ b\}$, which has two answer sets $\{a\}$ and $\{b\}$, while $\mathsf{forget}(P, F)$ has only one answer set $\{b\}$.

6.3 Update Logic Programs

Update programs [18; 19] and dynamic logic programs [1; 2] are besides [53; 62; 69] major approaches to updating nonmonotonic logic programs, and are in particular geared towards modeling sequences of updates (see [68] for a recent survey and comparison of these and other approaches).

An *update program* is a sequence $\mathcal{P} = [P_1, P_2, \ldots, P_t]$, $t \geq 1$, where each P_i is a logic program for $1 \leq i \leq t$. 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 the rules in P_{i+1} are assigned higher priority in case of conflicts. The semantics of update programs has been given by means of a translation into an ordinary logic program $P_{\lhd}$ although it can also be characterized in a purely declarative way.

Let $\mathcal{P} = [P_1, P_2, \ldots, P_t]$ be an update program. We introduce new atom $rej(r)$ for each rule r in $\mathcal{P}$, new atom a_i for each atom a in $\mathcal{P}$ and each i with $1 \leq i \leq t$. If $l = \neg a$, $\neg a_i$ is denoted l_i. The transformation $P_{\lhd}$ is defined as an ordinary logic program consisting of the following rules:

(i) all constraints in P_i, $1 \leq i \leq t$.

(ii) for each $r \in P_i$ with head h, $1 \leq i \leq t$: $h_i \leftarrow body(r), not\ rej(r)$.

(iii) for each $r \in P_i$ with head h, $1 \leq i \leq t$: $rej(r) \leftarrow body(r), \neg h_{i+1}$.

(iv) for each literal l occurring in $\mathcal{P}$ and $1 \leq i \leq t$: $l \leftarrow l_1;\quad l_i \leftarrow l_{i+1}$.

Definition 7 *[19] Let $\mathcal{P} = [P_1, P_2, \ldots, P_t]$ be an update program and let X be a set of literals. Then X is an answer set of $\mathcal{P}$ if it is an answer set of the transformation $P_{\lhd}$.*

It has been shown in [18; 19] that every update program $\mathcal{P}$ can be naturally translated into an equivalent inheritance logic program $ihp(\mathcal{P}) = (P, <)$ where $r < r'$

if $r \in P_i$ and $r' \in P_j$ such that $1 \leq i < j \leq t$. More precisely,

Lemma 2 ([19]) *For every update program $\mathcal{P} = [P_1, P_2, \ldots, P_t]$, a set S is an answer set of $\mathcal{P}$ iff S is an answer set of the inheritance program $ihp(\mathcal{P})$.*

Combining this lemma and Theorem 16 gives us a characterization of update programs in terms of semantic forgetting from a logic program.

Corollary 17 *Let $\mathcal{P}$ be an update program and let S be a set of literals. Then S is an answer set of $\mathcal{P}$ if and only S is an answer set of* $\mathsf{forget}(P', F)$*, where $ihp(\mathcal{P}) = (P, <)$ and P' is obtained from $(P, <)$ as described above.*

Finally, results in [19] show that for certain classes of update programs, the semantics coincides with the one under dynamic logic programming as in [1; 2]. Hence, as a consequence of our results and those in [19], the respective classes of dynamic logic programs can also be characterized by semantic forgetting in the way described.

6.4 Forgetting vs. Independence

As it is argued in [34], the notion of *independence* is important in automated deduction, query answering, and belief revision. For example, an intelligent agent must possess the ability of determining and discarding irrelevant information efficiently. When reasoning is involved, the issue of independence (or irrelevance) becomes more delicate and complex. The idea of forgetting about independent literals can be useful in improving reasoning procedures.

Informally, if a logic program P is independent of a literal l, then the answer set semantics of P should be unchanged if we forget about l from P. So, it is natural and reasonable to formally define the notion of *semantic independence* as follows.

Definition 8 *Let P be a consistent logic program. P is* semantically independent *of a literal $l \in Lit_P$, if $\mathcal{AS}(\mathsf{forget}(P, l)) = \mathcal{AS}(P)$.*

Obviously, if P is semantically independent of a literal l, then l can be safely "forgotten from P."

Example 11 *Consider the program $P = \{p \leftarrow not\ q.\ s \leftarrow s\}$. Clearly, P,* $\mathsf{forget}(P, q)$*, and* $\mathsf{forget}(P, s)$ *all have the single answer set $\{p\}$. Hence, P is semantically independent of both s and q.*

The following proposition provides an intuitive characterization for semantic independence.

Proposition 14 *Let P be a logic program. P is semantically independent of a lit-*

eral $l \in Lit_P$ if and only if $l \notin S$ for every answer set S of P.

This result, which is straightforward from Definition 2, is intuitive: P is independent of a literal l if and only l is false with respect to every answer set of P, that is, $P \not\models_c l$. In some cases, semantic independence of literals can be verified syntactically, as shown by the next result. Let $\mathbf{T}^*$ be the program transformations introduced in Section 4.3.1 and let $\mathbf{T}^*(P)$ be the respective canonical form of program P.

Proposition 15 *Let P be a logic program and let $l \in Lit_P$ be a literal. Suppose P' is any negative program obtained from P by transformations from $\mathbf{T}^*$ such that l does not occur in P'. Then P is semantically independent of l. Furthermore, such a program P' exists iff l does not occur in $\mathbf{T}^*(P)$.*

The converse of Proposition 15 is not true in general. For example, consider the program $P = \{p \leftarrow not\ q.\ q \leftarrow not\ p.\ q \leftarrow p.\}$. Then P is semantically independent of p. On the other hand, p occurs in every program P' resulting from P by (repeated) transformations in $\mathbf{T}^*$, and in particular in the canonical form $\mathbf{T}^*(P) = \{p \leftarrow not\ q.\ q \leftarrow not\ p.\ q \leftarrow not\ q.\}$. It remains as an interesting issue whether there is a set of program transformations that is strong enough to syntactically characterize the notion of semantic independence of literals.

7 Conclusion

While it is widely acknowledged that forgetting about atomic propositions in knowledge bases is an important technique for many AI applications, it has been less clear how this should materialize in the context of nonmonotonic reasoning and logic programming. To the best of our knowledge, this paper is the first attempt towards identifying criteria for this operation in this context. In particular, we have specified some desirable properties for forgetting in nonmonotonic logic programming. Based on these criteria, we have then proposed a semantics-based theory of forgetting literals in (disjunctive) logic programming. Compared to preliminary work, a distinguishing feature of our approach is that forgetting is defined in purely semantical terms. However, we have shown that this declarative approach to forgetting has a syntactic counterpart based on program transformations.

The properties of forgetting show that the approach in this paper extends the classical notion of forgetting and, moreover, satisfies all criteria that we have identified. As we have explained before, it also naturally generalizes the notion of forgetting for normal programs investigated in [65]. Furthermore, we have presented algorithms and analyzed the computational complexity of major reasoning tasks under forgetting.

Another approach to forgetting for normal logic programs was proposed in [71; 70]. Different from ours, the approach by Zhang and colleagues is procedural. The result of forgetting is obtained by removing some rules and/or literals, but little semantic justification for the removals is provided from a global perspective of logic programming. For their approaches of weak and strong forgetting, our criteria (F3) and (F4), which foster a semantic justification of forgetting and irrelevance of syntax under answer set semantics, do not hold. In Section 1, we have already shown that strong and weak forgetting do not have property (F4). To see that (F3) is not satisfied by strong and weak forgetting, we reconsider the example programs in Section 1. For the program $P = \{p \leftarrow .\ q \leftarrow not\ p\}$, the result of weak forgetting is $\mathsf{WForgetLP}(P, p) = \{q \leftarrow\}$; hence, $\mathsf{WForgetLP}(P, p) \models_s q$ while $P \not\models_s q$. Similarly, if we take $P = \{q \leftarrow not\ p.\ q \leftarrow not\ q\}$, then $\mathsf{SForgetLP}(P, p) = \{q \leftarrow not\ q\}$, which inconsistent; hence, $\mathsf{SForgetLP}(P, p) \models_s q$ but $P \not\models_s q$. If one adopts the natural reading of rules $a \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n$ as formulas $(b_1 \wedge \cdots \wedge b_m \wedge \neg c_1 \wedge \cdots \wedge \neg c_n) \supset a$ in propositional logic, then weak and strong forgetting appear to be intimately related to Lin's weakest sufficient and strongest necessary conditions [48], respectively, and thus have a strong classical flavor.

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. Furthermore, our results show that the semantic forgetting has a close relationship with inheritance programs [10], update programs [18; 19] and fragments of dynamic logic programs [1; 2].

Semantic forgetting has been extended [24] to HEX-programs, which allow to combine logic programs with Description Logics [23] and, applied in defining a notion of forgetting for the Web Ontology Language (OWL). [5] Furthermore, a system prototype for our semantic forgetting, called LPForget, which comprises two modules has been implemented and is available for experiments. [6] The module Forgetting serves for computing the result of forgetting about certain literals in a logic program under the answer set semantics; all algorithms introduced in this paper have been implemented in it. The other module, named CRS, facilitates conflict resolution (or preference recovery) in multi-agent systems along the approach in Section 6.1. In our system, once the constraints for different agents are specified, CRS will first check whether these constraints are consistent. If they are not consistent, the user can make compromises by forgetting about some literals such that an agreement is reached. The system can also make recommendations for the set of literals to be forgotten. However, the current recommendation algorithm is not optimized yet, and it remains to explore more efficient algorithms.

Several interesting issues remain for further research. One issue are more effi-

[5] `http://www.w3.org/2004/OWL/`

[6] The LPForget website is `http://www.cit.gu.edu.au/~kewen/LPForget`.

cient implementations and improved algorithms for computing the result of forgetting. In particular, given a disjunctive logic program P and a literal l, Algorithm $\mathsf{forget}_3(P, l)$ outputs generally a nested logic program which represents the result of forgetting about l from P by its minimal answer sets. It is well-known that nested logic programs can be efficiently transformed into an equivalent ordinary disjunctive programs if new symbols are allowed [59]. However, this does not carry over to minimal answer sets, and further minimization is needed. It would be interesting to see a syntax-based algorithm (i.e., based merely on program transformations but not on the actual answer sets) that outputs an ordinary disjunctive logic program on the original vocabulary as the result of forgetting.

Another issue is the application of forgetting in various scenarios of conflict resolving, such as ontology merging and alignment in the Semantic Web [58]. In these applications, both closed and open world reasoning are involved. Exploring a theory of forgetting in such a setting is an interesting issue.

Strong equivalence [45] has received a lot of attention in nonmonotonic logic programming because of its importance for program modularity and in applications like information integration. It would be interesting to introduce a notion of semantics forgetting that preserves strong equivalence, in line with the properties and the approach in this paper.

Finally, an extension of the approach in this paper to other semantics of nonmonotonic logic programming and more general formalisms, such as default logic or autoepistemic logic, is an intriguing issue.

Acknowledgments

The authors would like to thank Fu-Leung Cheng, Esra Erdem, Paolo Ferraris, Fangzhen Lin, Abdul Sattar, Kaile Su, Rodney Topor and Yan Zhang for helpful comments and discussions. We are also grateful to the reviewers for their helpful and constructive comments to clarify some aspects of this paper. This work was partially supported by the Austrian Science Funds (FWF) Projects P17212 and P18019, the EC project REWERSE (IST-2003-506779), and the Australia Research Council (ARC) Discovery Projects DP0666107 and DP0666540.

A Appendix

Proposition 5 *Let $\equiv_X$ be an equivalence relation on a collection of logic programs on Lit that is stronger than ordinary equivalence and invariant under literal extensions. Then forgetting does not preserve $\equiv_X$.*

Proof Since $\equiv_X$ is stronger than ordinary equivalence of logic programs, there must exist two programs P and P' such that $P \equiv P'$ but $P \not\equiv_X P'$. Let l be a new literal that appears neither in P nor P'. Then $\mathsf{forget}(P, l) \equiv P$ and thus P' is also a result of forgetting about l in P. Obviously, $P \equiv_X P$ but their results of forgetting (i.e. P and P') are not equivalent under $\equiv_X$. $\square$

Proposition 8 *Let P be a consistent logic program and let $F = \{l_1, \ldots, l_m\}$ be a set of literals. Then* $\mathsf{forget}(P, F) \equiv \mathsf{forget}(\mathsf{forget}(\mathsf{forget}(P, l_1), l_2), \ldots, l_m)$.

Proof Assume that $m > 1$. Let $M = \mathcal{AS}(\mathsf{forget}(P, F))$, $M' = \mathcal{AS}(\mathsf{forget}(P, F'))$ and $M'' = \mathcal{AS}(\mathsf{forget}(\mathsf{forget}(P, F'), l_m))$ where $F' = \{l_1, \ldots, l_{m-1}\}$.

We claim that $M = M''$. To prove this, we first show that $M \subseteq M''$. Consider any set $X \in M$. Then, by an analogue of item 5 in Proposition 6, there exists a stable model S of P such that $X = S - F$. Now in the process of iterative construction, let

$$P_0 = P,$$
$$P_i = \mathsf{forget}(P_{i-1}, l_i), \text{ for } i = 1, \ldots, m.$$

By item 5 of Proposition 6 and the fact that l_i does not occur in P_i, we can show by induction on $i = 1, 2, \ldots, m$ that there must exist some answer set S_i of P_i such that $S_i \subseteq S_{i-1} - \{l_i\}$. Consequently, $S_m \subseteq S - \{l_1, \ldots, l_m\} = S - F$. Hence, there exists an answer set S_m of P_m such that $S_m \subseteq X$.

Furthermore, $S_m \subset X$ is impossible. Otherwise, $S_m \cap X \subset X$ would hold. Thus, $S_m \cup F'$, for some $F' \subseteq F$, is an answer set of P, and $S_m \cup F' \subset S \setminus F$. Hence, S is not an F-answer set of P, which is a contradiction. Thus, we have $S_m = X$. In conclusion, $M \subseteq M''$.

Conversely, suppose $X \in M''$. That is, X is an answer set of P_m. By the definition of one-literal forgetting, this means that there exists an answer set S of P such that $S \setminus F = X$. Towards a contradiction, suppose that $X \notin M$. Then there exists a set Y in M such that $Y \subset X$. As already shown, Y is an answer set of P_m. But this contradicts that X is an answer set of P_m, as X is not minimal. Hence, $X \in M$. $\square$

Theorem 3 *Let P be a consistent (disjunctive) logic program and let $p \in Lit_P$ be an atom. Then $X \subseteq Lit_P$ is an answer set of* $\mathsf{forget}(P, p)$ *iff X is a minimal model of* $\mathsf{forget}(\mathsf{lcomp}(P), p)$. *That is,*

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

Proof We use the following lemma.

Lemma 3 *Let $X \subseteq Lit_P \setminus \{p\}$ such that $X \models \mathsf{forget}(\mathsf{lcomp}(P), p)$. Then either $X \in \mathcal{AS}(P)$ or $X \cup \{p\} \in \mathcal{AS}(P)$.*

Proof of Lemma 3 By Theorem 1, $X' \in \mathcal{AS}(P)$ iff $X' \models \mathsf{lcomp}(P)$ holds for each $X' \subseteq Lit_P$. Since $\mathsf{forget}(\mathsf{lcomp}(P), p) = \mathsf{lcomp}(P)(p/true) \vee \mathsf{lcomp}(P)(p/false)$, for each $X \subseteq Lit_P \setminus \{p\}$ such that $X \models \mathsf{forget}(\mathsf{lcomp}(P), p)$ thus either $X' = X \in \mathcal{AS}(P)$ (if $X \models \mathsf{lcomp}(P)(p/false)$) or $X' = X \cup \{p\} \in \mathcal{AS}(P)$ (if $X \models \mathsf{lcomp}(P)(p/true)$) holds. □

(i) $\mathcal{AS}(\mathsf{forget}(P, p)) \subseteq MMod(\mathsf{forget}(\mathsf{lcomp}(P), p))$: Let $X \in \mathcal{AS}(\mathsf{forget}(P, p))$. Then, there exists some $S \in \mathcal{AS}_p(P)$ such that $X \sim_p S$. By Theorem 1, $S \models \mathsf{lcomp}(P)$. Since $S \setminus \{p\} = X$, we have $X \models \mathsf{forget}(\mathsf{lcomp}(P), p)$. To show that X is also minimal, assume towards a contradiction that some $X' \subset X$ exists such that $X' \models \mathsf{forget}(\mathsf{lcomp}(P), p)$. By Lemma 3, $S' \in \mathcal{AS}(P)$ holds for either $S' = X'$ or $S' = X' \cup \{p\}$. In both cases, $S' \subset_p S$; however, this contradicts $S \in \mathcal{AS}_p(P)$. This proves $X \in MMod(\mathsf{forget}(\mathsf{lcomp}(P), p))$.

(ii) $MMod(\mathsf{forget}(\mathsf{lcomp}(P), p)) \subseteq \mathcal{AS}(\mathsf{forget}(P, p))$: Let $X \in MMod(\mathsf{forget}(\mathsf{lcomp}(P), p))$. By Lemma 3, $S \in \mathcal{AS}(P)$ for either $S = X$ or $S = X \cup \{p\}$. We show that in both cases $S \in \mathcal{AS}_p(P)$. Towards a contradiction, suppose that $S \notin \mathcal{AS}_p(P)$. Then there exists some $S' \in \mathcal{AS}_p(P)$ such that $S' \subset_p S$ and, by item 1 of Proposition 6, $X' := S' \setminus \{p\} \in \mathcal{AS}(\mathsf{forget}(P, p))$. By part (i), it follows that $X' \in MMod(\mathsf{forget}(\mathsf{lcomp}(P), p))$; Since $X' \subset X$, this contradicts $X \in MMod(\mathsf{forget}(\mathsf{lcomp}(P), p))$. Hence, $S \in \mathcal{AS}_p(P)$. By item 1 of Proposition 6, $S \setminus \{p\} = X \in \mathcal{AS}(\mathsf{forget}(P, p))$. This proves the result. □

Theorem 4 *Let P be a consistent (disjunctive) logic program and let $p \in Lit_P$ be an atom. Then $X \subseteq Lit_P \setminus \{p\}$ is an answer set of $\mathsf{forget}(P, p)$ if and only if either X or $X \cup \{p\}$ is a model of $\mathsf{Circ}(\mathsf{lcomp}(P), p)$ (resp., $\mathsf{ECWA}(\mathsf{lcomp}(P), p)$).*

Proof By Theorem 1, $\mathcal{AS}(P) = \{S \subseteq Lit_P \mid S \models \mathsf{lcomp}(P)\}$. Hence, from the characterization of models of $\mathsf{Circ}(\mathsf{lcomp}(P), p)$ in terms of $\subseteq_p$ and $\sim_p$, it is easily seen that $\mathcal{AS}_p(P) = \{S \subseteq Lit_P \mid S \models \mathsf{Circ}(\mathsf{lcomp}(P), p)\}$.

Consider $X \subseteq Lit_P \setminus \{p\}$. Suppose first that $X \in \mathcal{AS}(\mathsf{forget}(P, p))$. Then $X \sim_p S$ for some $S \in \mathcal{AS}_p(P)$, and either $S = X$ or $S = X \cup \{p\}$ must hold. As $S \models \mathsf{Circ}(\mathsf{lcomp}(P), p)$, the only-if direction of the theorem holds. Conversely, suppose that $S \models \mathsf{Circ}(\mathsf{lcomp}(P), p)$ for either $S = X$ or $S = X \cup \{p\}$. Then $S \in \mathcal{AS}_p(P)$ and by item 1 of Proposition 6, $S \setminus \{p\} = X \in \mathcal{AS}(\mathsf{forget}(P, p))$. This proves the result. □

Theorem 6 *For every consistent (disjunctive) logic program P and a literal l, Algorithm $\mathsf{forget}_2(P, l)$ outputs a correct representation of $\mathsf{forget}(P, l)$.*

Proof By the construction of P_1 in Step 1, $\mathcal{AS}(P_1) = \{X \in \mathcal{AS}(P) \mid l \in X\}$. By Proposition 1, $\mathcal{A}_1$ consists thus of all sets X such that $l \notin X$ and $X \cup \{l\} \in \mathcal{AS}_l(P)$.

In Step 2, the constraint $\leftarrow l$ guarantees that each answer set X of P_2 is an answer set of P such that $l \notin X$. The constraint $\leftarrow a_1, \ldots, a_k$, for $M = \{a_1, \ldots, a_k\} \in \mathcal{A}_1$, enforces that $M \cup \{l\} \not\subset X$ holds. Consequently, $\mathcal{A}_2$ consists of all sets $X \in \mathcal{AS}_l(P)$ such that $l \notin X$.

Combining the two cases, $\mathcal{AS}_l(P) = \{X \cup \{l\} \mid X \in \mathcal{A}_1\} \cup \mathcal{A}_2$; thus by item 1 of Proposition 6, $\mathcal{AS}(\mathsf{forget}(P, l)) = \mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$. It is easy to see that $\mathcal{AS}(P') = \mathcal{A}$; this proves the result. □

Proposition 9 *Let P be a disjunctive program. If P' is obtained by the elimination of head redundancy from P, then P and P' have the same answer sets.*

Proof Let $P = P_0 \cup \{r\}$ with $l \in head(r) \cap body^-(r)$ and let $P' = P_0 \cup \{(head(r) - l) \leftarrow body(r)\}$. We show that for every $S \subseteq Lit_P$, the following statement holds: (*) $S \in \mathcal{AS}(P)$ iff $S \in \mathcal{AS}(P')$. If $l \in S$, then $P^S = (P')^S$ and therefore (*) holds. Otherwise (i.e., $l \notin S$), the rule $r_1 = head(r) \leftarrow body^+(r)$ is in P^S iff the rule $r_2 = (head(r) - l) \leftarrow body^+(r)$ is in $(P')^S$. Furthermore, for each $S' \subseteq S$ it holds that $S' \models r_1$ iff $S' \models r_2$, and therefore $S' \models P^S$ iff $S' \models (P')^S$. Again it follows that (*) holds. □

To prove Theorem 7, we first show a lemma.

Lemma 4 *Let P be a DDLP and let S be an interpretation of P. If $S \in \mathcal{AS}(\mathcal{T}(P))$, then S is a minimal answer set of P.*

Proof Let $S \in \mathcal{AS}(\mathcal{T}(P))$. We first show that $S \in \mathcal{AS}(P)$. Let $r' \in P^S$ such that $S \models body(r')$. Then r' must be of the form $head(r) \leftarrow body^+(r)$ for some $r \in P$ such that $body^-(r) \cap S = \emptyset$ and $body^{--}(r) \subseteq S$. Furthermore, the rule $r'' = head(r) \leftarrow body^+(r), not\ not\ body^{--}(r)$ is in $\mathcal{T}(P)^S$. Now if $body^+(r) \subseteq S$, then $S \models head(r)$ as $S \models body^{--}(r)$. Hence, $S \models r'$. It follows that $S \models P^S$.

Consider any $S' \subseteq S$ such that $S' \models P^S$. Then $S' \models \mathcal{T}(P)^S$, as each rule in $\mathcal{T}(P)^S$ results from a rule in P^S by adding literals in the body. As $S \in \mathcal{AS}(\mathcal{T}(P))$, it follows that $S' = S$. This proves $S \in \mathcal{AS}(P)$. It remains to show that S is minimal. Consider any $S' \in \mathcal{AS}(P)$ such that $S' \subseteq S$. Then $S' \models \mathcal{T}(P)^{S'}$. Since $\mathcal{T}(P)^S \subseteq \mathcal{T}(P)^{S'}$, $S' \models \mathcal{T}(P)^S$. As $S \in \mathcal{AS}(\mathcal{T}(P))$, it follows that $S' = S$. □

Theorem 7 *For every N-acyclic DDLP P, it holds that $\mathcal{AS}(P) = \mathcal{AS}(\mathcal{T}(P))$.*

Proof $\mathcal{AS}(\mathcal{T}(P)) \subseteq \mathcal{AS}(P)$: Immediate from Lemma 4.

$\mathcal{AS}(P) \subseteq \mathcal{AS}(\mathcal{T}(P))$: Suppose $S \in \mathcal{AS}(P)$ but $S \notin \mathcal{AS}(\mathcal{T}(P))$. Then S is not a minimal model of $\mathcal{T}(P)^S$. Note that $S \models \mathcal{T}(P)^S$. Hence, there exists some interpretation $S' \subset S$ such that $S' \models \mathcal{T}(P)^S$. We show that then some interpretation S'' exists such that $S \subseteq S'' \subset S$ and $S'' \models P^S$; this means that S is not a minimal model of P^S and thus contradicts that $S \in \mathcal{AS}(P)$, proving the result.

We first assume that P is normal, i.e., $|head(r)| \leq 1$ for each $r \in P$. Let L be a level mapping witnessing the N-acyclicity of P and let $X = \{l \in S \setminus S' \mid \forall l' \in S \setminus S' : L(l) \leq L(l')\}$ be the set of all literals in $S \setminus S'$ having the smallest level. Since $S' \subset S$, $X \neq \emptyset$. We claim that $S'' := S \setminus X \models P^S$.

Towards a contradiction, suppose that $S'' \not\models r$ for some $r \in P^S$. Since $S \models P^S$, r is of the form $l \leftarrow body^+(r)$ where $l \in X$ and $body^+(r) \subseteq S''$. Since r stems from a rule $r' \in P$ such that $S \models r'$ (i.e., $r = head(r') \leftarrow body^+(r')$) and $S' \models \mathcal{T}(P)^S$, it follows that either (a) $body^+(r) \setminus S' \neq \emptyset$ or (b) $body^{--}(r') \setminus S' \neq \emptyset$. The minimality of $L(l)$ implies that (a) is the case and that $body^+(r) \cap X \neq \emptyset$, and thus $body^+(r) \not\subseteq S''$; this is a contradiction. This proves the claim $S'' \models P^S$, which contradicts that $S \in \mathcal{AS}(P)$. Hence, the result for normal programs P is proved.

The result for arbitrary programs is an easy consequence of this result and the following two facts. Call a program P' a *split* of a DDLP P, if P' results from P by replacing every rule r of form $l_1 \vee \cdots \vee l_k \leftarrow B$, $k \geq 2$, by at least one of the normal rules $l_1 \leftarrow B, \ldots, l_k \leftarrow B$. Now,

Fact A. For every $S \in \mathcal{AS}(P)$ there exists a split P' of P such that $S \in \mathcal{AS}(P')$.

Indeed, every split P' of P such that r is replaced by all rules $l_i \leftarrow B$ with $l_i \in S$ whenever $head(r) \cap S \neq \emptyset$, has S as an answer set. (This is a simple extension of a folklore result for ordinary logic programs [28; 30].)

Fact B. If a DDLP P is N-acyclic, then every split P' of P is N-acyclic. $\square$

Theorem 8 *Let P be a consistent disjunctive logic program and let $l \in Lit_P$ be a literal. Then* $\mathsf{forget}_3(P, l)$ *correctly represents* $\mathsf{forget}(P, l)$*, i.e., X is an answer set of* $\mathsf{forget}(P, l)$ *iff X is a minimal answer set of N'.*

Proof By Lemma 1, P can be transformed into an equivalent negative program N_0. Furthermore, an easy extension of Corollary 2 implies that N_0 is equivalent to N. Hence, $P \equiv N$ and $\mathsf{forget}(P, l) \equiv \mathsf{forget}(N, l)$.

Let $\mathcal{AS}_{min}(N')$ denote the minimal answer sets of program N' (w.r.t. $\subseteq$). To prove the theorem, it is now by item 1 of Proposition 6 sufficient to prove the following claim:

Claim. For every set X' of literals such that $l \notin X'$, $X' \in \mathcal{AS}_{min}(N')$ iff $X' = X \setminus \{l\}$ for some $X \in \mathcal{AS}_l(N)$.

Due to the elimination of tautology and the elimination of head redundancy, $head(r) \cap body(r) = \emptyset$ for each rule $r \in N$. Thus the program 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; $N_2 = \{r \in N \mid l \in head(r), l \notin body^-(r)\}$; and $N_3 = \{r \in N \mid l \notin head(r), l \in body^-(r)\}$. Notice that Step 3 in Algorithm $\mathsf{forget}_3(P, l)$ is performed only on the rules in N_3. Let the program N_3' result from N_3 by the transformations in Step 3. Then $N' = N_1 \cup N_3'$. Let $D_1, D_2, \ldots, D_s$ denote all possible conjunctions constructed from l_i in Step 3; note that $N_2 = \{r_1, \ldots, r_n\}$. We consider the three cases in Step 3.

(3.1) If $n = 0$, then no rule in N has l in the head. Thus l is false in every answer set of N, and hence $N \equiv N'$ and $\mathcal{AS}_l(N) = \mathcal{AS}(N)$. Since N' is an ordinary program, $\mathcal{AS}_{min}(N') = \mathcal{AS}(N')$. It follows that $\mathcal{AS}_{min}(N') = \mathcal{AS}_l(N)$ and the claim holds.

(3.2) If $n > 0$ and $m_j = 0$ for some $j \in \{1, \ldots, n\}$, then the rule $l \leftarrow$ is in N. Hence, every answer set of N contains l, and clearly $X' \in \mathcal{AS}(N')$ iff $X' \cup \{l\} \in \mathcal{AS}(N)$ holds for every set of literals X' with $l \notin X'$. As in the previous case, $\mathcal{AS}_l(N) = \mathcal{AS}(N)$ and $\mathcal{AS}_{min}(N') = \mathcal{AS}(N')$, and the claim holds.

(3.1) $n \geq 1$ and $m_i \geq 1$ for every $i = 1, \ldots, n$. We use the following lemmas.

Lemma 5 *If $l \notin X'$ and $X' \models D_{i_0}$ for some i_0 with $1 \leq i_0 \leq s$, then $(N_3)^{X'} = (N_3')^{X'}$.*

Lemma 6 *For every $X' \in \mathcal{AS}_{min}(N')$,*

(1) if $X' \models D_{i_0}$ for some i_0, $1 \leq i_0 \leq s$, then $X' \in \mathcal{AS}(N)$.
(2) if $X' \not\models D_i$ for all $i = 1, \ldots, s$, then $X' \cup \{l\} \in \mathcal{AS}(N)$.

Proof of Lemma 6 (1) $N^{X'} = (N_1)^{X'} \cup (N_3)^{X'}$ since $X' \models D_{i_0}$ implies $l \notin X'$. By Lemma 5, $(N_3)^{X'} = (N_3')^{X'}$ and thus $N^{X'} = (N')^{X'}$. So $X' \models N^{X'}$.

If $X'' \subseteq X'$ and $X'' \models N^{X'}$, then $X'' \models (N')^{X'}$. Since X' is a minimal model of $N^{X'}$, $X'' = X'$. Therefore, $X' \in \mathcal{AS}(N)$.

(2) Let $X = X' \cup \{l\}$. Since $l \in X$ and l does not appear in N_1, we have $N^X = (N_1 \cup N_2 \cup N_3)^X = (N_1)^X \cup (N_2)^X = (N_1)^{X'} \cup (N_2)^X = (N_1)^{X'} \cup \{l \leftarrow\}$. By the assumption, $X' \models (N_1)^{X'}$ and thus $X \models (N_1)^{X'}$. Obviously, $X \models (N_2)^X$ since $l \in X$ and every rule in N_2 has head l. Thus, $X \models N^X$.

Now suppose that $Y \subseteq X$ and $Y \models N^X$. Since $l \leftarrow$ is in N^X, $l \in Y$. Let $Y = Y' \cup \{l\}$ where $l \notin Y'$. Then $Y' \models (N_1)^{X'}$. By $(N')^{X'} = (N_1)^{X'}$, we have $Y' \models (N')^{X'}$.

Thus, $Y' = X'$ by $Y' \subseteq X'$ and the minimality of X'. This implies $Y = X$. So X is a minimal model of N^X, and thus $X \in \mathcal{AS}(N)$. $\square$

We now prove the claim.

($\Rightarrow$): Let $X' \in \mathcal{AS}_l(N)$. Then $X \in \mathcal{AS}(N)$ for either $X = X'$ or $X = X' \cup \{l\}$, where $l \notin X$. Consider two cases:

Case 1. $X = X'$: Note that $X' \models D_{i_0}$ for some i_0, $1 \leq i_0 \leq s$, since $l \notin X'$. By Lemma 5, $(N_3)^{X'} = (N_3')^{X'}$. Therefore, $(N')^{X'} = (N_1)^{X'} \cup (N_3')^{X'} = (N_1)^{X'} \cup (N_3)^{X'}$. This implies $X' \models (N')^{X'}$.

Suppose now $X'' \subseteq X'$ and $X'' \models (N')^{X'}$. Then $X'' \models N^{X'}$. Since X' is a minimal model of $N^{X'}$, we have $X'' = X'$. Thus, $X' \in \mathcal{AS}(N')$.

Suppose that $Y \in \mathcal{AS}_{min}(N')$ and $Y \subseteq X'$. If $Y \models D_{j_0}$ for some j_0 ($1 \leq j_0 \leq s$), then by Lemma 6 $Y \in \mathcal{AS}(N)$ and thus it follows that $Y = X'$. So $X' \in \mathcal{AS}_{min}(N')$. If $Y \not\models D_i$ for all i ($1 \leq j_0 \leq s$), then $Y \cup \{l\} \in \mathcal{AS}(N)$ again by Lemma 6. This implies $X' \notin \mathcal{AS}_l(N)$, a contradiction. Therefore, $X' \in \mathcal{AS}_{min}(N')$.

Case 2. $X = X' \cup \{l\}$: Then $N^X = N^{X' \cup \{l\}} = (N_1)^{X'} \cup \{l \leftarrow\}$. Since $l \in X$, we have $X \not\models D_i$ for every i, $1 \leq i \leq s$. From $l \notin X'$, it follows that $X' \not\models D_i$ for every i, $1 \leq i \leq s$. Thus, $(N')^{X'} = (N_1)^{X'} \cup (N_3')^{X'} = (N_1)^{X'}$. Since $X \models N^X$, $X' \models (N_1)^{X'}$ and thus $X' \models (N')^{X'}$.

If $X'' \subseteq X'$ and $X'' \models (N')^{X'}$, then $X'' \cup \{l\} \models N^X$. By the minimality of X, $X'' \cup \{l\} = X$. Since $l \notin X'$ and $l \notin X''$, we have $X'' = X'$. Thus, $X' \in \mathcal{AS}(N')$.

If $Y \in \mathcal{AS}_{min}(N')$ and $Y \subseteq X'$, then $Y \not\models D_i$ for all $i = 1, \ldots, s$. Consequently. $Y \cup \{l\} \in \mathcal{AS}(N)$. So $Y \cup \{l\} \subseteq X' \cup \{l\}$. This means $Y = X'$, since the answer sets of N are incomparable under $\subseteq$. Thus $X' \in \mathcal{AS}_{min}(N')$.

($\Leftarrow$): Suppose that $X' \in \mathcal{AS}_{min}(N')$. Consider two possible cases:

Case 1. $X' \not\models D_i$ for all $i = 1, \ldots, s$: By Lemma 6, $X = X' \cup \{l\} \in \mathcal{AS}(N)$. Thus $X' \in \mathcal{AS}_l(N)$ since $l \in X$ by item 3 of Proposition 1.

Case 2. $X' \models D_{i_0}$ for some i_0 with $1 \leq i_0 \leq s$: By Lemma 6, $X' \in \mathcal{AS}(N)$. Consider any $Y \in \mathcal{AS}(N)$, $Y \neq X'$, such that $Y \setminus \{l\} \subseteq X'$. Then $l \in Y$ since $l \notin X'$ and answer sets of N are incomparable. Thus $Y \in \mathcal{AS}_l(N)$ by Proposition 1. From the proof of the only-if part, $Y \setminus \{l\}$ is a (minimal) answer set of N'. By the minimality of X', $Y \setminus \{l\} = X'$. This proves $X' \in \mathcal{AS}_l(N)$.

$\square$

Proposition 11 *Let P be a consistent normal logic program and let $l \in Lit_P$ be a literal. If no negative cycle of DG_P contains l, then* $\mathsf{forget}_3(P, l)$ *outputs an N-acyclic program.*

Proof The result is a consequence of the fact that for a program P as described, also in the dependency graph DG_{N_0} for the program N_0 from Step 1 of $\mathsf{forget}_3(P, l)$ no negative cycle will contain l. This is because each of the transformations in $\mathbf{T}^*$ preserves the property that if a literal l does not occur on a negative cycle of DG_{P_0} of the original program P_0, then it does not occur on a negative cycle of DG_{P_1} in the transformed program P_1. Indeed, each transformation except Unfolding only removes edges; Unfolding can add only some positive edges from l to l' such that positive edges from l to l'' and from l'' to l' exist, for some literal l'', and some negative edges from l to l' such that a positive edge from l to l'' and a negative edge from l'' to l' exist, for some literal l''. Thus, occurrence of l in a negative cycle of DG_{P_1} implies occurrence of l in a negative cycle of DG_{P_0}.

As a consequence, in Step 3 (where $N = N_0$) no replacements of $not\ l$ by D_i will be performed that can violate the condition (ii) of N-acyclicity; condition (i) is vacuously true. □

Theorem 9 *Given a consistent (disjunctive) logic 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.*

Proof Deciding whether $X \notin \mathcal{AS}_l(P)$ can be done in NP time using an NP oracle: we must show that either (1) $X \notin \mathcal{AS}(P)$ (which is in co-NP, cf. [22]), or that (2) there exists some $X' \in \mathcal{AS}(P)$ such that $X' \subset_l X$; such an X' can be guessed and checked using an NP-oracle in polynomial time. Consequently, l-answer set checking is in co-$\mathrm{NP}^{\mathrm{NP}}$ = Π_2^p.

The hardness result is shown by a reduction from deciding whether a given logic program P (without strong negations) has no answer set, which is Π_2^p-complete [22].

In fact, given a (disjunctive) logic program P, construct a logic program $P' = \{head(r) \leftarrow p, body(r) \mid r \in P\} \cup \{q \leftarrow not\ p.\ p \leftarrow not\ q\} \cup \{a \leftarrow not\ p \mid a$ 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. □

Theorem 10 *Given a consistent normal logic 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.*

Proof Similar to the proof of Theorem 9, in order to show that $X \notin \mathcal{AS}_l(P)$, we must show that either (1) $X \notin \mathcal{AS}(P)$ (which can be tested in polynomial time), or that (2) there exists some $X' \in \mathcal{AS}(P)$ such that $X' \subset_l X$. Such an X' can be guessed and checked in polynomial time. Hence, deciding $X \notin \mathcal{AS}_l(P)$ is in NP, which implies that l-answer set checking is in co-NP.

The hardness result is shown by the reduction in Theorem 9. In fact, note that if we consider only normal programs P (without strong negation), then P' is also a normal program. The problem of deciding whether P has no answer set, which is co-NP-complete, is thus reduced to deciding whether X_0 is a p-answer set of P'. □

Theorem 11 *Given a consistent negative 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.*

Proof The co-NP membership follows from that fact that for any given set of literals X and negative program P, deciding whether $X \in \mathcal{AS}(P)$ is polynomial. (Indeed, $X \in \mathcal{AS}(P)$ iff $X \models P^X$ and $X \setminus \{l\} \not\models P^X$, for every $l \in X$.) Thus, testing that (1) $X \in \mathcal{AS}(P)$ and that (2) there is no $X' \in \mathcal{AS}(P)$ such that $X' \subset_l X$ is feasible in co-NP.

As for co-NP-hardness, let $C = C_1 \wedge \cdots \wedge C_k$ be a propositional CNF over atoms $y_1, \ldots, y_m$, where each C_j is non-empty. Define

$$N = \bigcup_{i=1}^{m} (N_i \cup \{ \leftarrow not\ C'_i\}) \cup \{l \leftarrow not\ y_1.\ l \leftarrow not\ y'_1\},$$

where

$$\begin{aligned} N_i &= \{y_i \leftarrow not\ y'_i.\ y'_i \leftarrow not\ y_i.\ y_i \leftarrow not\ l.\ y'_i \leftarrow not\ l\}, \quad 1 \leq i \leq m, \\ C'_j &= \{y_i \mid y_i \in C_j\} \cup \{y'_i \mid \neg y_i \in C_j\}, \quad 1 \leq j \leq k. \end{aligned}$$

Clearly, the satisfying assignments of C correspond one-to-one to the answer sets of N containing l. Furthermore, the set $X = \{y_i, y'_i \mid 1 \leq i \leq m\}$ is an answer set of N. It holds that X is also an l-answer set of N iff C is unsatisfiable, which establishes the co-NP-hardness. □

Theorem 12 *Given a consistent (disjunctive) logic program P and literals l and l', deciding whether* $\mathsf{forget}(P, l) \models_c l'$ *is Σ_3^p-complete.*

Proof Given a logic program P and two literals l and l', $\mathsf{forget}(P, l) \models_c l'$ holds iff there exists some $S \in \mathcal{AS}_l(P)$ such that $l' \in S$. Since deciding $S \in \mathcal{AS}_l(P)$ is in Π_2^p, the problem thus is in Σ_3^p.

The Σ_3^p-hardness can be shown by an encoding of quantified Boolean formulas (QBFs) of the form $\exists Z \forall X \exists Y C$. It is well-known that the problem of deciding whether a disjunctive logic program has no answer sets is Π_2^p-complete [22]. As shown there, for every QBF $F = \forall X \exists Y C$ there exists a polynomial-time constructible logic program P_F which has no answer set iff F is true. We extend this program to an encoding of a QBF $F' = \exists Z \forall X \exists Y C$ into credulous inference under forgetting as follows.

First, P_F can be easily extended to a program $P_{F[Z]}$ encoding a QBF $F[Z] = \forall X \exists Y C[Z]$ with free variables (i.e., parameters) Z, where all new atoms in $P_{F[Z]}$ are from Z and do not occur in rule heads, such that for every truth assignment τ to Z, the program $P_{F[Z/\tau(Z)]} = P_{F[Z]} \cup \{z \leftarrow . \mid z \in Z, \tau(Z) = true\}$ has no answer set iff the QBF $F[Z/\tau(Z)]$ evaluates to true, where $F[Z/\tau(Z)]$ results from $F[Z]$ by replacing every $z \in Z$ with T if $\tau(z) = true$ and with F if $\tau(z) = false$.

More in detail, suppose without loss of generality that $C[Z] = C_1 \wedge \cdots \wedge C_r$ where each $C_i = \ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}$ is a disjunction of literals $\ell_{i,j}$ over atoms $X \cup Y \cup Z$. Let $X' = \{x' \mid x \in X\}$, $Y' = \{y' \mid y \in Y\}$, and w be new atoms. Then $P_{F[Z]}$ consists of the rules

$$
\begin{array}{lll}
x \vee x' \leftarrow . & & \text{for each } x \in X, \\
y \vee y' \leftarrow . \quad y \leftarrow w. \quad y' \leftarrow w. \quad w \leftarrow y, y'. & & \text{for each } y \in Y, \\
w \leftarrow \sigma(\neg.l_{k,1}), \sigma(\neg.l_{k,2}), \sigma(\neg.l_{k,3}). & & \text{for each } k = 1, \ldots, r, \\
w \leftarrow not\ w. & &
\end{array}
$$

where σ maps classical literals ℓ to classical and not literals as follows:

$$
\sigma(\ell) = \begin{cases} x' & \text{if } \ell = \neg x \text{ for some } x \in X \\ y' & \text{if } \ell = \neg y \text{ for some } y \in Y \\ not\ z & \text{if } \ell = \neg z \text{ for some } z \in Z \\ \ell & \text{otherwise} \end{cases}
$$

Notice that the only not literals occurring in $P_{F[Z]}$ are $not\ w$ in the last rule and the literals $not\ z$ in rule bodies; furthermore, atoms from Z occur only in rule bodies. For void Z, the program $P_{F[Z]}$ amounts to the program P_F in [22, Proof of Theorem 3] (after converting the $\forall\exists$-QBF into the $\exists\forall$-QBF used there). A simple extension of the proof of Theorem 3 in [22] gives the following lemma.

Lemma 7 *For each truth assignment τ to Z, $\forall X \exists Y C[Z/\tau(Z)]$ evaluates to true iff the program $P_{F[Z/\tau(Z)]}$ has no answer set.*

Let $Z' = \{z' \mid z \in Z\}$, l, and l', be fresh atoms, and let P be the logic program obtained from $P_{F[Z]}$ by adding l to the body of each rule and the following further rules:

(1) $l \leftarrow not\ l'.$ $\quad l' \leftarrow not\ l.$,
(2) $z \leftarrow not\ z'.$ $\quad z' \leftarrow not\ z,$ for each $z \in Z$,
(3) $x \leftarrow not\ l.$ $\quad x' \leftarrow not\ l.$ $\quad y \leftarrow not\ l.$ $\quad y' \leftarrow not\ l,$ for each $x \in X, y \in Y$, and
(4) the rule $w \leftarrow not\ l.$

Informally, the rules in (1) select one of l and l', and the rules in (2) select a truth assignment τ to Z. If l is selected, then the program $P_{F[Z]}$ is activated and evaluated for the selected τ, while the rules (3) and (4) are discarded. The evaluation will lead to some answer set S^l_τ, iff the program $P_{F[Z/\tau(Z)]}$ has some answer set; note that every such S contains l. On the other hand, if l' is selected, then only the rules in (2)-(4) are active. The truth assignment τ selected in (2) will be complemented with $X \cup X' \cup Y \cup Y' \cup \{w\}$ to a unique answer set $S^{l'}_\tau$ of the program P.

Now $S^l_\tau \subset_l S^{l'}_\tau$ holds for every S^l_τ. Furthermore, answer sets S and S' corresponding to different truth assignments τ and τ', respectively, are always incomparable w.r.t. $\subseteq$ on $Z \cup Z'$, and thus also w.r.t. $\subseteq_l$. Therefore, for every τ, $S^{l'}_\tau$ is an l-answer set of P iff no answer set S^l_τ exists, i.e., $P_{F[Z/\tau(Z)]}$ has no answer set, which by Lemma 7 equals that $\forall X \exists Y C[Z/\tau(Z)]$ evaluates to true. Hence, for some τ, $S^{l'}_\tau$ is an l-answer set of P iff $F' = \exists Z \forall X \exists Y C$ evaluates to true. Since every l-answer set of P that contains l' is of the form $S^{l'}_\tau$ for τ, it follows from Proposition 6 that $\mathsf{forget}(P, l) \models_c l'$ iff F' evaluates to true. Since P is constructible in polynomial time from F', the Σ^p_3-hardness is proved. $\square$

Theorem 13 *Given a consistent negative program P and literals l and l', deciding whether* $\mathsf{forget}(N, l) \models_c l'$ *is Σ^p_2-complete.*

Proof By Theorem 11, a guess for some l-answer set X of P such that $l' \in X$ can be verified with an NP oracle in polynomial time. Hence, deciding $\mathsf{forget}(P, l) \models_c l'$ is in Σ^p_2.

As for Σ^p_2-hardness, take a QBF $\exists X \forall Z E$, where $E = \bigvee_{i=1}^k D_i$ is a DNF on $X \cup Z$ such that without loss of generality in each disjunct D_i some variable from Z occurs. Construct the same program as above in Theorem 11 for $C = \neg E$ and where $Y = X \cup Z$ and y_1 is an arbitrary variable from Z, but (1) omit the clauses $x_i \leftarrow not\ l$ and $x'_i \leftarrow not\ l$, and (2) add a clause $l' \leftarrow not\ l$, where l' is a fresh literal. Then for each set $X' \subseteq X$, the set

$$S_{X'} = X' \cup \{x'_i \mid x_i \in X \setminus X'\} \cup Z \cup \{z'_j \mid z_j \in Z\} \cup \{l'\}$$

is an answer set of N. The sets $S_{X'}$ are also all answer sets of N that contain l' (and do not contain l). Furthermore, $S_{X'}$ is an l-answer set of$N$ iff there exists no

satisfying truth assignment for C (= $\neg E$) which corresponds on X to $S_{X'}$ in the obvious way. In summary, this means that N has l-answer set in which l' is true, i.e., $\mathsf{forget}(N, l) \models_c l'$, iff the formula $\exists X \forall Z E$ evaluates to true. □

Theorem 14 *Given a consistent normal program N and literals l and l', deciding whether* $\mathsf{forget}(N, l) \models_c l'$ *is Σ_2^p-complete.*

Proof It is sufficient to note that the program N constructed in the proof of Theorem 13 is normal. □

Theorem 15 *Given a consistent logic program P and literals l and l', deciding whether* $\mathsf{forget}(P, l) \models_s l'$ *is (i) Π_2^p-complete for arbitrary disjunctive logic programs P, and (ii) co-NP-complete for normal logic programs and for negative logic programs P.*

Proof By Theorem 7, to decide $\mathsf{forget}(P, l) \models_s l'$, we need only to decide $P \models_s l'$. The latter is Π_2^p-complete for logic programs [22] and co-NP complete for normal/negative programs; for both cases, membership in co-NP follows since testing $X \in \mathcal{AS}(P)$ is polynomial (cf. proof of Theorem 11 for negative programs), and co-NP hardness from the results in [52] (incorporating the consistency requirement is easy). □

Proposition 13 *Let P be a logic program and let F be a consistent set of literals. Suppose that (1) no literal in F occurs in a rule body in P, and (2) for each rule r, either no or every literal in $head(r)$ is in F. Then* $\mathsf{forget}(P, F) = P \setminus R(F)$.

We first provide two lemmas, which are straightforward corollaries of the well-known, more general Splitting Set Theorem in [47] and elementary properties of answer set semantics. To avoid introducing the necessary notions for that result, we provide for self-containedness simple genuine proofs.

Lemma 8 *Let P, F and $Q = P \setminus R(F)$ be given as in Proposition 13. If S' is an answer set of Q, then there exists a subset Y' of F such that $S' \cup Y'$ is an answer set of P.*

Since we have a very special case here, Lemma 8 also allows a simple proof.

Proof Suppose that S' is an answer set of Q. Denote $D' = \{head(r) \mid r \in R(F), S' \models body(r)\}$. Then every literal in D' must be in F. Let Y' be a minimal model of D' and $S = S' \cup Y'$. By the assumption of F, $S' \cap Y' = \emptyset$ and thus S is consistent. We show that S is an answer set of P.

First, since $P^S = P^{S'} = Q^{S'} \cup (R(F))^{S'}$, we have $S \models P^S$. Next, suppose $X \subseteq S$ and $X \models P^S$. Take $X' = X \setminus F$; then $X' \subseteq S'$. We can see that $X' \models Q^{S'}$ and thus

$X' = S'$. Notice that $Y' = S \cap F$ is a minimal model of D', and that $X \cap F \models D'$ implies $X \cap F = S \cap F$. Thus, $X = S$. $\square$

Lemma 9 *Let P, F and $Q = P \setminus R(F)$ be given as in Proposition 13. If S is an answer set of P, then $S \setminus F$ is an answer set of Q.*

Proof By $P^S = P^{S'}$ and $Q \subseteq P$, it is easy to see that $S' \models Q^{S'}$ where $S' = S \setminus F$.

Suppose that $Y' \subseteq S'$ and $Y' \models Q^{S'}$. Then $Y \models P^{S'} = P^S$ where $Y = Y' \cup (S \cap F)$. Thus $Y = S$. Since $Y' = Y \cap \bar{F}$ and $S' = S \cap \bar{F}$, we have $Y' = S'$. Here $\bar{F} = Lit_P \setminus F$.

Therefore, $S \setminus F$ is an answer set of Q. $\square$

Proof of Proposition 13 Denote $Q = P \setminus R(F)$. Suppose that S' is an answer set of Q. By Lemma 8, there exists a subset Y' of F such that $S = S' \cup Y'$ is an answer set of P.

To show that S' is an answer set of $\mathsf{forget}(P, F)$, it suffices to prove that S is an F-answer set of P. In fact, if Z is an answer set of P such that $Z \subseteq_F S$, then $Z' \subseteq S'$ where $Z' = Z \setminus F$. By Lemma 9, Z' is an answer set of Q. Thus $S' = Z'$, which means $Z \sim_F S$. That is, S is an F-answer set of P.

On the other hand, if S' is an answer set of $\mathsf{forget}(P, F)$, then there exists an F-answer set S of P such that $S' = S \setminus F$. By Lemma 9, S' is also an answer set of Q. $\square$

We are now ready to prove Theorem 16.

Theorem 16 *Let $(P, <)$ be an inheritance program and let S be a set of literals. Then S is an inheritance answer set of $(P, <)$ iff S is an answer set of $\mathsf{forget}(P', F)$ where P' is obtained as above.*

Proof S is an inheritance answer set of $(P, <)$ iff S is a minimal model of $(P, <)^S$ iff S is an answer set of $P \setminus R(F)$ iff S is an answer set of $\mathsf{forget}(P', F)$, by Proposition 13. $\square$

References

[1] J. Alferes, J. Leite, L. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic logic programming. In A. Cohn, L. Schubert, and S. Shapiro, editors, *Proceedings of the Sixth International Conference on the Principles of Knowledge Representation and Reasoning*, pages 98–109. Morgan Kaufmann Publishers, 1998.

[2] J. Alferes, J. Leite, L. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic Programming*, 45(1-3):43–70, 2000.

[3] Asparagus homepage. `http://asparagus.cs.uni-potsdam.de/`, Since 2005.

[4] R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12(1-2):53–87, 1994.

[5] W. Bledsoe and L. Hines. Variable elimination and chaining in a resolution-based prover for inequalities. In W. Bibel and R. A. Kowalski, editors, *Proceedings of 5th Conference on Automated Deduction (CADE 1980)*, volume 87 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 1980.

[6] G. Boole. *The Mathematical Analysis of Logic*. London: G. Bell, 1847. (Reprinted by Philosophy Library, New York, 1948.)

[7] S. Brass and J. Dix. Characterizations of the disjunctive stable semantics by partial evaluation. *Journal of Logic Programming*, 32(3):207–228, 1997.

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

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

[10] F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In D. De Schreye, editor, *Proceedings of the International Conference on Logic Programming (ICLP'99)*, pages 79–93. The MIT Press, 1999.

[11] F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. *Theory and Practice of Logic Programming*, 2(3):293–321, 2002.

[12] M. Cadoli, F. Donini, P. Liberatore, and M. Schaerf. Space efficiency of propositional knowledge representation formalisms. *Journal of Artificial Intelligence Research*, 13:1–31, 2000.

[13] M. Cadoli, F. M. Donini, P. Liberatore, and M. Schaerf. Preprocessing of intractable problems. *Information and Computation*, 176(2):89–120, 2002.

[14] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

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

[16] A. de Waal and J. Gallagher. Logic program specialisation with deletion of useless clauses. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming (ILPS 1993)*, page 632. MIT Press, 1993.

[17] T. Eiter and M. Fink. Uniform equivalence of logic programs under the stable model semantics. In C. Palamidessi, editor, *Proceedings 19th International Conference on Logic Programming (ICLP 2003)*, number 2916 in LNCS, pages 224–238. Springer, 2003.

[18] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Considerations on updates of logic programs. In M. Ojeda-Aciego, I. P. de Guzmán, G. Brewka, L. M. Pereira, editors, *Proceedings of the Seventh European Workshop on Logics in Artificial Intelligence (JELIA'2000)*, volume 1919 of *Lecture Notes in Artificial Intelligence*, pages 2–20. Springer-Verlag, 2000.

[19] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of update sequences based on causal rejection. *Theory and Practice of Logic Programming*, 2(6):711–767, 2002.

[20] T. Eiter, M. Fink, and S. Woltran. Semantical characterizations and complexity of equivalences in answer set programming. *ACM Transactions on Computational Logic*, 8(3), Aug. 2007.

[21] T. Eiter and G. Gottlob. The complexity of logic-based abduction. *Journal of the ACM*, 42:3–42, 1995.

[22] T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3-4):289–323, 1995.

[23] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer set programming. In L. P. Kaelbling and A. Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 90–96. Professional Book Center, 2005.

[24] T. Eiter, G. Ianni, R. Schindlauer, H. Tompits, and K. Wang. Forgetting in managing rules and ontologies. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence (WI 2006), Hongkong, Dec. 2006*, pages 411–419. IEEE Computer Society, 2006. Preliminary version at ALPSWS 2006.

[25] T. Eiter, K. Makino, and G. Gottlob. Computational aspects of monotone dualization: A brief survey. *Discrete Applied Mathematics*, 2007. doi:10.1016/j.dam.2007.04.017. Preliminary version available as Tech. Rep. INFSYS RR-1843-06-01, Institute of Information Systems, TU Vienna.

[26] E. Erdem and V. Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3:499–518, 2003.

[27] M. Gelfond and V. Lifschitz. Compiling circumscriptive theories into logic programs. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI 1988)*, pages 455–449. AAAI Press / The MIT Press, 1988.

[28] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski, K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming (ICLP/SLP 1988)*, pages 1070–1080. The MIT Press, 1988.

[29] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In D. H. D. Warren, Péter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming (ICLP 1990)*, pages 579–597. The MIT Press, 1990.

[30] M. Gelfond and V. Lifschitz. Classical negation in logic programs and deductive databases. *New Generation Computing*, 9:365–385, 1991.

[31] J. Grant, J. Horty, J. Lobo, and J. Minker. View updates in stratified disjunctive databases. *Journal of Automated Reasoning*, 11(2):249–267, 1993.

[32] K. Inoue and C. Sakama. Negation as failure in the head. *Journal of Logic Programming*, 35(1):39–78, 1998.

[33] T. Janhunen, I. Niemelä, P. Simons, and J.-H. You. Partiality and disjunctions in stable model semantics. In A. G. Cohn, F. Giunchiglia, and B. Selman, editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2000), April 12-15, Breckenridge, Colorado, USA*, pages 411–419. Morgan Kaufmann Publishers, Inc., 2000.

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

[35] J. Lang and P. Marquis. Resolving inconsistencies by variable forgetting. In D. Fensel, F. Giunchiglia, D. L. McGuinness and M.-A. Williams, editors, *Proceedings of the Eighth International Conference on the Principles of Knowledge Representation and Reasoning*, pages 239–250. Morgan Kaufmann, 2002.

[36] J. Larrosa. Boosting search with variable elimination. In R. Dechter, editor, *Proceedings 6th International Conference on Principles and Practice of Constraint Programming (CP 2000)*, volume 1894 of *Lecture Notes in Computer Science*, pages 291–305. Springer, 2000.

[37] J. Larrosa, E. Morancho, and D. Niso. On the practical use of variable elimination in constraint optimization problems: 'still-life' as a case study. *Journal of Artificial Intelligence Research*, 23:421–440, 2005.

[38] C. Lassez, K. McAloon, and G. S. Port. Stratification and knowledge based management. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming (ICLP 87)*, pages 136–151. MIT Press, 1987.

[39] J. Lee and V. Lifschitz. Loop Formulas for Disjunctive Logic Programs. In C. Palamidessi, editor, *Proceedings of the Nineteenth International Conference on Logic Programming (ICLP-03)*, volume 2916 of *Lecture Notes in Computer Science*, pages 451–465. Springer, 2003.

[40] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[41] C. I. Lewis. *A Survey of Symbolic Logic*. University of California Press (Reprinted by Dover Pub's., Inc., New York, 1960. Chapter II, "The Classic, or Boole-Schroder Algebra of Logic"), 1918.

[42] V. Lifschitz. Computing circumscription. In A. K. Joshi, editor, *Proceedings 9th International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 121–127. Morgan Kaufmann, 1985.

[43] V. Lifschitz. Circumscription. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume III, pages 297–352. Clarendon Press, Oxford, 1994.

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

[45] V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):426–541, 2001.

[46] V. Lifschitz, L. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

[47] V. Lifschitz and H. Turner. Splitting a logic program. In P. Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming (ICLP 1994)*, pages 23–37. MIT Press, 1994.

[48] F. Lin. On strongest necessary and weakest sufficient conditions. *Artificial Intelligence*, 128(1-2):143–159, 2001.

[49] F. Lin and R. Reiter. Forget it. In R. Greiner and D. Subramanian, program cochairs, *Proceedings of the AAAI Fall Symposium on Relevance*, pages 154–159. The AAAI Press, Technical Report FS-94-02, 1994.

[50] F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 112–117. The AAAI Press, 2002.

[51] F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157:115–137, 2004.

[52] W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38(3):588–619, 1991.

[53] W. Marek and M. Truszczyński. Revision programming. *Theoretical Computer Science*, 190:241–277, 1998.

[54] J. McCarthy. Circumscription — a form of nonmonotonic reasoning. *Artificial Intelligence*, 13(1-2):27–39, 1980.

[55] A. Middeldorp, S. Okui, and T. Ida. Lazy narrowing: Strong completeness and eager variable elimination. *Theoretical Computer Science*, 167(1&2):95–130, 1996.

[56] Y. Moinard. Forgetting literals with varying propositional symbols. *Journal of Logic and Computation*, 17(5):955–982, 2007.

[57] R. Moore. Semantical considerations on nonmonotonic logics. *Artificial Intelligence*, 25:75–94, 1985.

[58] N. F. Noy and H. Stuckenschmidt. Ontology alignment: An annotated bibliography. In *Semantic Interoperability and Integration*, volume 04391 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2005.

[59] D. Pearce, H. Tompits, and S. Woltran. Encodings for equilibrium logic and logic programs with nested expressions. In P. Brazdil and A. Jorge, editors, *10th Portuguese Conference on Artificial Intelligence (EPIA 2001)*, volume 2258 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2001.

[60] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[61] D. Sacca and C. Zaniolo. Partial models and three-valued stable models in logic programs with negation. In A. Nerode, W. Marek and V.S. Subrahmanian, editors, *Proceedings of the Workshop on Nonmonotonic Reasoning and Logic Programming*, pages 87–101. MIT Press, 1991.

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

[63] D. Subramanian, R. Greiner, and J. Pearl, editors. *Artificial Intelligence Journal: Special Issue on Relevance*. Elsevier, 1997. Volume 97, Numbers 1-2.

[64] C. Tessier, L. Chaudron, and H. Müller. *Conflicting Agents - Conflict Management in Multi-Agent Systems*. Kluwer Academic Publishers, Cambridge, 2001.

[65] K. Wang, A. Sattar, and K. Su. A theory of forgetting in logic programming. In *Proceedings of the 20th National Conference on Artificial Intelligence*, pages 682–687. AAAI Press, 2005.

[66] K. Wang and L. Zhou. Comparisons and computation of well-founded semantics for disjunctive logic programs. *ACM Transactions on Computational Logic*, 6(2):295–327, 2005.

[67] A. Weber. Updating propositional formulas. In *Proceedings of the First Conference on Expert Database Systems (EDS 1986)*, Larry Kerschberg, editor, pages 487–500. Benjamin Cummings, 1987.

[68] Y. Zhang. Logic program-based updates. *ACM Transactions on Computational Logic*, 7(3):421–472, 2006.

[69] Y. Zhang and N. Foo. Towards generalized rule-based updates. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI '97)*, pages 82–87. Morgan Kaufmann, 1997.

[70] Y. Zhang and N. Foo. Solving logic program conflict through strong and weak forgettings. *Artificial Intelligence*, 170(8-9):739–778, 2006.

[71] Y. Zhang, N. Foo, and K. Wang. Solving logic program conflicts through strong and weak forgettings. In L. P. Kaelbling and A. Saffiotti, editors, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 627–632. AAAI Press, 2005.