Preserving Strong Equivalence while Forgetting

Matthias Knorr and José J. Alferes

CENTRIA & Departamento de Informática, Faculdade Ciências e Tecnologia,
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal

Abstract. A variety of proposals for forgetting in logic programs under different semantics have emerged that satisfy differing sets of properties considered desirable. Despite the achieved progress in devising approaches that capture an increasing number of these properties, the idea that the result of forgetting should preserve the meaning of the initial program for the remaining, non-forgotten, atoms, has not yet been captured. In particular, the existing proposals may not preserve dependency relations between such atoms that are given by the structure of the program. In logic programs, these relations are captured by strong equivalence, but, preserving strong equivalence of two different programs while forgetting does not suffice. Rather, strong equivalence relativized to the remaining atoms should be preserved between the original program and the one that results from forgetting. In this paper, we overcome this deficiency by formalizing the property that captures this maintenance of relations while forgetting, and at the same time a general semantic definition for such a forgetting for arbitrary logic programs. Then, we study forgetting for normal programs under the well-founded semantics, and for programs with double negation under the answer set semantics. In both cases, we focus on efficient syntax-based algorithms that only manipulate the rules in which changes are effectively necessary.

1 Introduction

Removing or hiding information that is no longer needed in a knowledge base, also known as *forgetting* or *variable elimination* [12], is important in Knowledge Representation and Reasoning (KRR). This is witnessed by the recent amount of work developed for different logical formalisms [20, 9, 16, 13, 21, 2] and for Logic Programming (LP) in particular [6, 15, 14, 1], and has been applied, e.g., in cognitive robotics, ontologies, and resolving conflicts.

For LP, these approaches are commonly introduced together with a number of desirable properties that justify design rationales and allow comparisons between different approaches. Yet, the property that the result of forgetting should preserve all the semantic dependencies contained in the original program, for all but the atom(s) to be forgotten, has not been considered.

Example 1. Consider a part of a taxonomy including professors, university staff, and persons with properties assigned to them, represented in rules:[1]

$$person(X) \leftarrow ustaff(X) \qquad ustaff(X) \leftarrow professor(X)$$

[1] As usual, rules with variables stand for the set of ground rules obtained by replacing the variables by constants in all possible ways.

Consider that $professor(mary)$ is part of the program. Then, clearly, $person(mary)$ is derivable. Now suppose that we want to forget about the class university staff, e.g., because there are no longer specific properties attached to it. In this case, it should still be derivable that every professor is also a person, i.e., in the result of forgetting (all ground instances of) $ustaff(X)$ from this program, $person(mary)$ should still be derivable. This is indeed the case for most existing approaches of forgetting.

Now consider that university staff that are not professors must use the punch clock, and staff that do not have to use the punch clock have flexible schedules:

$$flexible(X) \leftarrow ustaff(X), not\, punchClock(X)$$
$$punchClock(X) \leftarrow ustaff(X), not\, professor(X)$$

Suppose that $professor(mary)$, $ustaff(peter)$ and $person(john)$ is the only information about these three individuals contained in the program. Then, we expect to derive that $flexible(mary)$ and $punchClock(peter)$ hold. Now suppose that we want to forget about $punchClock(X)$ from the program, then the derivation that professors have flexible schedules should not be lost. If we learn later that $john$ is a professor, then we would expect to be able to derive that he also has a flexible schedule, while $peter$ still does not. However, none of the existing proposals for forgetting in logic programs satisfies the desired behavior (see related work in Section 6).

Strong equivalence [10] has been introduced in LP to semantically capture the dependency relations between atoms expressed in logic programs, and has been used, e.g., in program optimization. Two programs P and Q are strongly equivalent if, for all programs R, $P \cup R$ and $Q \cup R$ are equivalent, i.e., they have the same models. However, preserving strong equivalence between programs to which the same forgetting is applied is not sufficient as it does not say much about how similar the output is to the original program [6, 1]. Neither would considering strong equivalence between a program and its result of forgetting work, simply because, in general, this does not hold for all programs R: consider P with two rules $a \leftarrow not\, b$ and $b \leftarrow not\, c$, then adding just $c \leftarrow$ to P allows us to derive a, but adding $c \leftarrow$ and $b \leftarrow$ does not, and there is no program Q over a and c (and without b) that allows us to obtain the same result.

Instead, strong equivalence restricted to programs R over the remaining, non-forgotten, atoms should be preserved between the original program and the one that results from forgetting. Relativized strong equivalence (RSE) [5, 17] was introduced to relax strong equivalence when certain internal atoms are no longer allowed to be part of R and thus captures our idea, yet no related notion of forgetting exists.

Most approaches for forgetting in LP also provide methods on how to obtain the result of forgetting, and often these methods rely on computing models and then determining a representation of the result of forgetting. Thus, the complexity of computing such a result usually corresponds to that of computing models in the considered class of logic programs, but additionally, the resulting program may be exponential in the size of the original program (see [14], and the syntactic transformation in [6]). We argue that computing the result of forgetting should in general not need to change any rules other than those containing the atoms to be forgotten. Therefore, we focus on syntax-based algorithms that manipulate precisely these rules without the need to compute any models. As argued in [6], such kind of algorithms are also of benefit for applications.

In this paper, we introduce a new property that formalizes the idea of preserving relativized strong equivalence while forgetting in LP and a new general definition of forgetting for logic programs that, besides our new property, also satisfies a number of other desired properties. We then study concrete cases of this new notion under the two most-widely used semantics for LP. Namely, we consider forgetting from normal logic programs under the well-founded semantics [7], and forgetting from programs with double negation under the answer set semantics [8]. It turns out, that such forgetting is not always possible in the latter case, so we subsequently study adequate restrictions under which our approach can still be applied. The construction of the resulting program, in both cases, is then achieved by applying syntactic transformations which do not require the computation of models. We can show that computing the resulting program is exponential in the number of rules that contain atoms to be forgotten and linear in the remaining. We argue that on average this is at least competitive when compared to computing models in P or NP (over the entire program), and certainly better than an algorithm that, in addition to computing models, creates a program exponential in the size of the entire given program.

2 Logic Programs

We start by recalling notions and notation of LPs. More precisely, we consider logic programs with double negation, a subset of extended logic programs [11].

A *logic program* P, is a finite set of rules r of the form

$$a \leftarrow b_1, ..., b_l, not\, c_1, ..., not\, c_m, not\, not\, d_1, ..., not\, not\, d_n$$

where a and all b_h, c_i, and d_j, for $1 \leq h \leq l$, $1 \leq i \leq m$, and $1 \leq j \leq n$, are propositional atoms over a signature Σ. Alternatively, a may be the special logical constant $\perp$ representing an empty head. Given such a rule r, we distinguish the *head* of r as $H(r) = a$, and the *body* of r, $B(r) = B^+(r) \cup not\, B^-(r) \cup not\, not\, B^{--}(r)$, where $B^+(r) = \{b_1, \ldots, b_l\}$, $B^-(r) = \{c_1, \ldots, c_m\}$, $B^{--}(r) = \{d_1, \ldots, d_n\}$, and, for a set A of atoms, $not\, A = \{not\, q\colon q \in A\}$ and $not\, not\, A = \{not\, not\, q\colon q \in A\}$.

Logic programs of this general form include a number of special kinds of rules: if $m = n = 0$, then we call r *positive*; if $l = m = n = 0$, then r is a *fact*; if $a = \perp$, then r is a *constraint*; and if $a \neq \perp$ and $n = 0$, we say r is *normal*. The classes of *positive* and *normal programs* are defined as a finite set of positive and normal rules, respectively.

We now recall the *answer set semantics* [8] by first defining the least model of positive programs and then relying on a generalization of the reduct to nested programs, a very general class of programs which admits double negation in the body [11].

Given a logic program P, B_P is the set of all atoms appearing in P. An *interpretation* for P is a set of atoms $I \in B_P$, and is meant to represent all the atoms considered true. A positive rule is *satisfied* in interpretation I if $B^+(r) \subseteq I$ implies $a \in I$. An interpretation I is a *model* of a positive program P if I satisfies all rules $r \in P$, and I is the *least model* of P if there is no model I' of P such that $I' \subset I$. The *reduct* of P w.r.t. an interpretation I is defined as $P^I = \{H(r) \leftarrow B^+(r) : r \in P, B^-(r) \cap I = \emptyset, B^{--}(r) \subseteq I\}$. Then, an interpretation I is an *answer set* of P iff I is the least model of P^I. A program is called *consistent* if it has (at least) one answer set, and the set of all answer sets of P is denoted by $M_{as}(P)$. Determining whether a (propositional) program has an answer set is NP-complete [4].[2]

We also consider the *well-founded semantics* [7] for normal programs. A *3-valued interpretation* I of a program P is defined as $I = I^+ \cup not\, I^-$ with $I^+ \cup I^- \subseteq B_P$ and $I^+ \cap I^- = \emptyset$; I^+ and I^- contain the atoms that are true and false in I, respectively; an atom p appearing neither in I^+ nor in I^- is undefined, and so is $not\, p$. The computation of the well-founded model requires a consequence operator T_P for three-valued interpretations that derives true information. For a normal program P and a three-valued interpretation I for P, we define $T_P(I) = \{H(r) : r \in P, B(r) \subseteq I\}$. The notion of unfounded set complements that by deriving false information. For a normal program P and a three-valued interpretation I for P, we say that $U \subseteq B_P$ is an *unfounded set* (of P) *w.r.t.* I if each atom $a \in U$ satisfies the following condition: for each rule $r \in P$ with $H(r) = a$ at least one of the following holds: (Ui) $not\, b_h \in I$ for some $b_h \in B^+(r)$, or $c_i \in I$ for some $c_i \in B^-(r)$; (Uii) $b_h \in U$ for some $b_h \in B^+(r)$. The *greatest unfounded set* $U_P(I)$ (of P) w.r.t. I always exists and leads to the definition of operator $W_P(I)$, by setting $W_P(I) = T_P(I) \cup not\, U_P(I)$. This operator W_P is monotonic, can be iterated by $W_P \uparrow 0 = \emptyset$, $W_P \uparrow (n+1) = W_P(W_P \uparrow n)$ for all n, and its least fixed point, which exactly corresponds to the *well-founded model* $M_{wf}(P)$, is obtained for some finite n for propositional normal programs as considered here. Determining $M_{wf}(P)$ of a (propositional) normal program is P-complete [4].

Finally, given a set of atoms V with $V \subseteq \Sigma$, two programs P_1 and P_2 are *strongly equivalent relative to* V under semantics S, denoted $P_1 \equiv_S^V P_2$, iff $M_S(P_1 \cup R) = M_S(P_2 \cup R)$ for all programs R over signature V. This notion is generalized from answer sets [17, 5] to arbitrary semantics S and captures as special cases that P_1 and P_2 are *equivalent* and *strongly equivalent*, denoted $P_1 \equiv_S P_2$ and $P_1 \equiv_S^s P_2$, by considering $V = \emptyset$ and $V = \Sigma$, respectively.

3 Forgetting with Strong Persistence

In previous work [6, 18, 15, 14, 1], a number of desirable properties for forgetting in logic programs has been investigated under both answer set and well-founded semantics. Before we introduce our new property, we generalize several properties presented in [14] to arbitrary classes of logic programs and arbitrary semantics.

For that purpose, we define that, given interpretation I under semantics S and a set of atoms V, $I_{\parallel V}$ represents the part of I without elements from V. E.g., for the answer set semantics, $I_{\parallel V}$ represents $I \setminus V$ and for the well-founded semantics $I \setminus (V \cup not\, V)$. For sets of interpretations $\mathcal{I}$, we also define $\mathcal{I}_{\parallel V} = \{I_{\parallel V} : I \in \mathcal{I}\}$.

Now, let $\mathcal{C}$ be a class of logic programs over a signature Σ, P and P' programs in $\mathcal{C}$, S a semantics for $\mathcal{C}$, $V \subseteq B_P$, and $\mathsf{f}(P, V)$ abstractly denote a program resulting from forgetting about V from P. Note that, in general $\mathsf{f}(P, V)$ does not determine a syntactically unique program but rather one representative of a class of (strongly) equivalent programs (depending on the considered notion of forgetting, e.g., in most notions, a result $\{q \leftarrow\}$ would also represent $\{q \leftarrow; q \leftarrow q\}$). Some properties about $\mathsf{f}(P, V)$ are:

[2] More precisely, this result coincides with the one first established for normal programs in [4].

(E) Existence w.r.t. $\mathcal{C}$: if P is in $\mathcal{C}$, then $\mathsf{f}(P, V)$ is expressible in $\mathcal{C}$.
(IR) Irrelevance: $\mathsf{f}(P, V) \equiv^s_S P'$ for some P' that does not contain any $v \in V$
(SE) Strong Equivalence: If $P \equiv^s_S P'$, then $\mathsf{f}(P, V) \equiv^s_S \mathsf{f}(P', V)$.
(CP) Consequence Persistence: $M_S(\mathsf{f}(P, V)) = M_S(P)_{\|V}$.

There are three further properties presented in [14], but it is shown that two of them conflict with the others in the case of answer set semantics. Moreover, all three require an additional entailment relation over logic programs, defined over HT logic for answer set semantics in [14], which is non-standard since entailment in LP is usually considered only for (sets of) atoms. Since the choice of this entailment relation for each semantics affects whether these properties hold or not, we leave such a study for future work.

As motivated in the introduction, none of the existing definitions of forgetting ensures that the result of forgetting really semantically resembles the original program if we ignore the atom(s) to be forgotten. This is why we introduce a new property that can be considered a generalization of **(CP)**, i.e., consequence persistence but under (relativized) strong equivalence, hence the name *strong persistence*.

(SP) Strong Persistence: $M_S(\mathsf{f}(P, V) \cup R) = M_S(P \cup R)_{\|V}$ for all programs R over signature $\Sigma \setminus V$.

The definition of **(SP)** strongly resembles that of relativized strong equivalence. The only essential technical difference is that we have to omit the elements in V from the models of $P \cup R$. This also clarifies that, even though both notions are strongly related, they are not identical nor is one a special case of the other.

Given that none of the existing approaches on forgetting for logic programs satisfy **(SP)**, we introduce a new general definition of forgetting in LP.

Definition 1. *Let $\mathcal{C}$ be a class of logic programs, and S a semantics for $\mathcal{C}$. A result of* strong S-forgetting *about $V \in B_P$ from $P \in \mathcal{C}$, denoted $F_S(P, V)$, is a program s.t.*

(1) all $v \in V$ do not appear in $F_S(P, V)$, and
(2) $M_S(F_S(P, V) \cup R) = M_S(P \cup R)_{\|V}$ for all programs R over signature $\Sigma \setminus V$.

Due to its generality, this notion of forgetting naturally satisfies **(SP)** for any class of programs and any semantics, but also several other of the previously introduced properties, namely **(IR)**, **(SE)**, and **(CP)**.

Proposition 1. *Let $\mathcal{C}$ be a class of logic programs, and S a semantics for $\mathcal{C}$. Then strong S-forgetting satisfies* **(IR)**, **(SE)**, **(CP)**, *and* **(SP)**.

Whether our definition of forgetting also satisfies **(E)** depends on the concrete class of programs and semantics and, in the following sections, we answer this question for the well-founded semantics of normal programs, and the answer set semantics of programs with double negation.

4 Strong WF-Forgetting for Normal Programs

We first consider *strong wf-forgetting*, in which the considered class of logic programs is normal programs and the semantics the well-founded semantics. We start by providing an algorithm (Alg. 1) that computes a result that satisfies Def. 1 for the simpler case of forgetting a single atom p from a normal program P, denoted $F_{wf}(P, p)$. Here and in the following, we abuse notation, and represent the singleton set $\{p\}$ simply by p.

Before we discuss Alg. 1, we need to introduce one additional notion, namely that of a wf-dual w.r.t. a program P and an atom p, that is useful when substituting $not\, p$ in rules in P while forgetting about p from P. For that, given a literal l, the *complementary literal*, $\bar{l}$, is defined as $\bar{p} = not\, p$ and $\overline{not\, p} = p$.

Definition 2. *Let P be a normal program, $p \in B_P$, and R all the n rules in P of the form $p \leftarrow l_{j1}, \ldots, l_{jm_j}$ where $n \geq 1$, $1 \leq j \leq n$, $m_j \geq 1$ for all j. The* wf-dual w.r.t. P and p*, denoted $\mathcal{D}_{wf}(P, p)$, is the set of all possible sets $\{\bar{l}_{1k_1}, \ldots, \bar{l}_{nk_n}\}$ with $1 \leq k_1 \leq m_1, \ldots, 1 \leq k_n \leq m_n$.*

The wf-dual w.r.t. P and p can be understood as a set of conjunctions that, building on the rules in P with head p and non-empty body, can be used to replace $not\, p$, but preserve its truth value. Consider P containing two rules $p \leftarrow s$ and $p \leftarrow not\, q, not\, r$. Then $\mathcal{D}_{wf}(P, p) = \{\{not\, s, q\}, \{not\, s, r\}\}$ and, e.g., $not\, p$ is true if one of the two conjuncts is true, false if both conjuncts are false, and undefined otherwise. This is what we apply in Alg. 1 whose details we explain next.

First, P' is initialized with P from which all rules whose head appears in the (positive) body are removed right away (line 1). This is known as elimination of tautologies TAUT [3]. Then, new rules are introduced by substituting occurrences of p in the bodies (of rules r) with the bodies of rules r_1 whose head is p, in a way similar to wGPPE [3] (lines 3-12). This includes a special case if $not\, p$ appears in the body of r_1 (lines 5-7). I.e., with such a rule alone, p would be undefined, which is why $not\, p$ is replaced with the negation of the rule head of r. Subsequently, all rules with p in the body can be removed (line 13). Next, new rules are introduced in which all $not\, p$ in rule bodies (apart from those with rule head p – line 14 – since those will be eliminated at the end) are substituted by the wf-duals (lines 14-28), unless one of the two special cases applies. Namely, either there is no rule with head p in which case $not\, p$ can simply be omitted in such a rule body (lines 15-16) or there is a fact for p, in which case none of the rules with $not\, p$ is considered any further for substitution (line 17). The application of the wf-duals again includes a special case to handle potential undefinedness due to the presence of $H(r)$ or p in the wf-dual (lines 20-22). Finally, rules containing p (in the head) or $not\, p$ in the body can be removed (line 29).

Example 2. Consider the following normal program P to illustrate Alg. 1:

$$r_1 : r \leftarrow p \qquad r_2 : q \leftarrow not\, p \qquad r_3 : p \leftarrow not\, p, t \qquad r_4 : p \leftarrow not\, s$$

The program $F_{wf}(P, p)$ returned by Alg. 1 is

$$r'_1 : r \leftarrow not\, r, t \qquad r'_2 : r \leftarrow not\, s \qquad r'_3 : q \leftarrow not\, q, s \qquad r'_4 : q \leftarrow not\, t, s$$

```
input  : Normal program $P$ and $p \in B_P$
output: Normal program $P' = F_{wf}(P, p)$
1  $P' := P \setminus \{r \in P : H(r) \cap B^+(r) \neq \emptyset\}$;
2  $R_1 := \{r \in P' : H(r) = p\}$;
3  for $r \in P'$ s.t. $p \in B(r)$ do
4      for $r_1 \in R_1$ do
5          if $not\, p \in B(r_1)$ then
6              $P' := P' \cup \{H(r) \leftarrow (B(r) \setminus \{p\}) \cup (\{not\, H(r)\} \cup (B(r_1) \setminus \{not\, p\}))\}$;
7          end
8          else
9              $P' := P' \cup \{H(r) \leftarrow (B(r) \setminus \{p\}) \cup B(r_1)\}$;
10         end
11     end
12 end
13 $P' := P' \setminus \{r \in P' : p \in B(r)\}$;
14 $R_2 := \{r \in P' : not\, p \in B(r), H(r) \neq p\}$;
15 if $R_1 = \emptyset$ then
16     $P' := P' \cup \{H(r) \leftarrow B(r)' : r \in R_2, B(r)' = B(r) \setminus \{not\, p\}\}$;
17 else if $\{p \leftarrow\} \not\subseteq R_1$ then
18     for $r \in R_2$ do
19         for $D \in \mathcal{D}_{wf}(P, p)$ do
20             if $(H(r) \in D)$ or $(p \in D)$ then
21                 $P' := P' \cup \{H(r) \leftarrow$
                   $(B(r) \setminus \{not\, p\}) \cup ((D \setminus \{H(r), p\}) \cup \{not\, H(r)\})\}$;
22             end
23             else
24                 $P' := P' \cup \{H(r) \leftarrow B(r) \setminus \{not\, p\} \cup D\}$;
25             end
26         end
27     end
28 end
29 $P' := P' \setminus (R_1 \cup R_2)$;
```

Algorithm 1: Strong wf-forgetting for a single atom p

where r_1' and r_2' are obtained from r_1 in combination with r_3 (by lines 5-7) and r_4 (by lines 8-10), respectively, while r_3' and r_4' are obtained from r_2, the duals over r_3 and r_4, and lines 20-22 and 22-25 respectively. It can be verified that $M_{wf}(P') = M_{wf}(P)_{\|\{p\}} = \{r, not\, q, not\, s, not\, t\}$, i.e., (2) of Def. 1 holds for $R = \emptyset$. In fact, it holds for arbitrary programs R over $\Sigma \setminus \{p\}$, e.g., for $R = \{s \leftarrow r\}$, we have $M_{wf}(P' \cup R) = M_{wf}(P \cup R)_{\|\{p\}} = \{not\, t\}$.

Example 3. Consider only the rules and facts explicitly given in Ex. 1 as P. The result of $F_{wf}(P, V)$ with $V = \{punchClock(X) \mid X \in \{mary, peter, john\}\}$ contains precisely three instances of $flexible(X) \leftarrow ustaff(X), professor(X)$. Thus, $flexible(mary)$ is derivable right away, and if $professor(john)$ is added later, then $flexible(john)$ becomes derivable as well.

We can show that Alg. 1 always returns a result P' that corresponds to $F_{wf}(P, p)$.

Theorem 1. *Given a normal program P and $p \in B_P$, Alg. 1 computes $F_{wf}(P, p)$.*

Alg. 1 can be generalized to arbitrary sets $V \in B_P$ using the following property applicable to strong S-forgetting for arbitrary classes $\mathcal{C}$ of programs and semantics S for $\mathcal{C}$.

Theorem 2. *Let $\mathcal{C}$ be a class of logic programs, S a semantics for $\mathcal{C}$, $P \in \mathcal{C}$, and $V_1, V_2 \subseteq B_P$. Then, for all $P' \in \mathcal{C}$, P' is $F_S(P, V_1 \cup V_2)$ iff P' is $F_S(F_S(P, V_1), V_2)$.*

Thus, Alg. 1 allows us to compute strong wf-forgetting about one atom, and Thm. 2 ensures that we can forget a set of atoms by simply forgetting each atom one after the other in any chosen order. This also guarantees that **(E)** holds for strong wf-forgetting.

Proposition 2. *Strong wf-forgetting for normal programs satisfies* **(E)**.

The computational complexity of Alg. 1 is as follows.

Theorem 3. *Given a normal program P and $p \in B_P$, computing $F_{wf}(P, p)$ is in EXPTIME in the number of rules containing occurrences of p and linear in the remaining rules.*

We would like to point out that this worst-case exponential is indeed limited to the wf-duals w.r.t. P and p, i.e., to the number of rules n_1 whose head is p and the number of body literals in these rules. In fact, any of the transformations in Alg. 1 (apart from the linear one in line 1) only affects rules in which p occurs. Since it is reasonable to assume that, in large programs, the atom to be forgotten does on average appear only in a small fraction of the rules, we argue that this considerably relativizes the high worst-case complexity, in the sense that an exponential on a small fraction of the input may be preferable to a polynomial over all rules as in [1].

5 Strong AS-Forgetting for Programs with Double Negation

We now present *strong as-forgetting* under the answer set semantics. Similar to [15, 14], strong as-forgetting for normal programs does not satisfy **(E)**. Consider $p \leftarrow \mathit{not}\, q$ and $q \leftarrow \mathit{not}\, p$ whose answer sets are $\{p\}$ and $\{q\}$. The result of strong as-forgetting about q should have two answer sets $\{\}$ and $\{p\}$, and there is no normal program where an answer set is a subset of another. That is why we consider logic programs with double negation where one single rule $p \leftarrow \mathit{not}\, \mathit{not}\, p$ suffices as such result of forgetting.[3]

Unfortunately, due to such rules, strong as-forgetting under the answer set semantics for programs with double negation is not always possible.

Example 4. Consider the following program P from which we want to forget about p:

$$p \leftarrow \mathit{not}\, \mathit{not}\, p \qquad\qquad q \leftarrow p \qquad\qquad r \leftarrow \mathit{not}\, p$$

[3] Applicability of ASP solvers to such programs is ensured by (linear) transformations cf. [6].

```
input  : Program $P$ with double negation
output: Program $P' = NF(P)$ in normal form
1  $P' := P \setminus \{r \in P : H(r) \cap B^+(r) \neq \emptyset\}$;
2  $P' := P' \setminus \{r \in P : B^+(r) \cap B^-(r) \neq \emptyset\}$;
3  $P' := P' \setminus \{r \in P : B^-(r) \cap B^{--}(r) \neq \emptyset\})$;
4  $R' := \{r \in P' : B^+(r) \cap B^{--}(r) \neq \emptyset\}$;
5  $P' := (P' \setminus R') \cup \{H(r) \leftarrow B(r)' : r \in R', B(r)' = B(r) \setminus \{not\,not\,q : q \in$
   $(B^+(r) \cap B^{--}(r))\}\}$;
6  $R'' := \{r \in P' : H(r) \cap B^-(r) \neq \emptyset\}$;
7  $P' := (P' \setminus R'') \cup \{\bot \leftarrow B(r) : r \in R''\}$;
8  $R''' := \{r \in P' : H(r) = B^{--}(r), B^+(r) \cup B^-(r) = \emptyset\}$;
9  for $r \in R'''$ do
10     if $\bot \leftarrow H(r)$ or $\bot \leftarrow not\,not\,H(r)$ then
11         $P' := P' \setminus r$;
12     end
13     if $\bot \leftarrow not\,H(r)$ then
14         $P' := (P' \setminus (r \cup \{\bot \leftarrow not\,H(r)\})) \cup \{H(r) \leftarrow\}$;
15     end
16 end
```

Algorithm 2: Computing a normal form of P

Strong as-forgetting requires to find a program over $\{q, r\}$ that satisfies condition (2) of Def. 1. Note first that P itself has two answer sets $\{p, q\}$ and $\{r\}$ and that adding either q or r as facts to P simply adds the atom to both answer sets, i.e., $P \cup \{q\}$ has two answer sets $\{p, q\}$ and $\{q, r\}$ and $P \cup \{r\}$ has two answer sets $\{p, q, r\}$ and $\{r\}$. We thus require that $P' = F_{as}(P, p)$ has two answer sets $\{q\}$ and $\{r\}$, and that $P' \cup \{q\}$ and $P' \cup \{r\}$ also both have two answer sets, namely $\{q\}$ and $\{q, r\}$, and $\{r\}$ and $\{q, r\}$ respectively. Such a program P' does not exist over $\{q, r\}$ since (a) it is required to be symmetric in q and r, (b) we have to ensure that precisely only one of q and r is true in each answer set of P', but (c) adding either of the two explicitly, must not avoid the existence of an answer set that contains the other and in which both atoms are true.

In the following, we investigate conditions under which strong as-forgetting can still be applied focusing again on syntactic transformations (as in the previous section), in the sense that computing the result of forgetting about $V \in B_P$ in P only uses the rules r with $H(r) \in V$ to replace (possibly negated) occurrences of V in the bodies of rules.

We start by introducing a normal form that simplifies the presentation and the cases to consider in such an algorithm, and, as a byproduct, reduces the size of the program. Formally, a logic program P with double negation is in *normal form* if: for every $p \in B_P$ and each rule $r \in P$, at most one of p, $not\,p$, and $not\,not\,p$ occurs in $B(r)$; if $H(r) = p$, then neither p nor $not\,p$ occur in $B(r)$ and; if $r = p \leftarrow not\,not\,p$, then no constraint containing only p, $not\,p$, or $not\,not\,p$ in its body occurs in P. This normal form $NF(P)$ can be computed using Alg. 2 in linear time. It first applies some general program transformations including TAUT and CONTRA [3] (lines 1-7), and then handles the case that simplifies $p \leftarrow not\,not\,p$ in combination with constraints (lines 8-16). The algorithm is correct, and $NF(P)$ is strongly equivalent to the original program:

Proposition 3. *Let P be a logic program with double negation. Alg. 2 computes $NF(P)$, and P and $NF(P)$ are strongly equivalent.*

With such a normal form in place, we proceed to introduce a notion that indicates whether a certain atom p can be forgotten syntactically from a given program P.

Definition 3. *Let P be a logic program with double negation in normal form and $p \in B_P$. We call P p-*forgettable *if a) there is no $r \in P$ with $H(r) \cap B^{--}(r) \neq \emptyset$ or b) $p \leftarrow$ in P, or c) there is no rule r with $H(r) \neq p$ and $p \in B^+(r) \cup B^-(r) \cup B^{--}(r)$.*

Case a) describes the kind of rule which in general conflicts with the existence of the result of strong as-forgetting (cf. also Ex. 4), while the cases b) and c) indicate exceptions under which rules described in a) are not problematic. Note that any normal logic program P is p-forgettable for any $p \in B_P$, and that rules matching case a) are allowed for all other atoms except the one to be forgotten.

Checking whether a program is p-forgettable is easy. So given a p-forgettable program P in normal form, we now introduce Alg. 3 for forgetting about a single atom p from P, denoted $F_{as}(P, p)$.

For that purpose, we introduce some further notation. First, we introduce a function N that applies a number of negation symbols to elements of a rule body by defining, for all $p \in B_P$ and for $x \in \{p, not\,p, not\,not\,p\}$, $N^0(x) = x$, $N^1(p) = not\,p$, $N^1(not\,p) = not\,not\,p$, and $N^1(not\,not\,p) = not\,p$, $N^2(p) = N^2(not\,not\,p) = not\,not\,p$ and $N^2(not\,p) = not\,p$, and $N^3 = N^1$. Also, for a rule body S, $N^i(S) = \{N^i(s) : s \in S\}$. We also adapt the notion of dual to strong as-forgetting.

Definition 4. *Let P be a logic program, $p \in B_P$, and R all $n \geq 1$ rules in P of the form $p \leftarrow l_{j1}, \ldots, l_{jm_j}$ where $1 \leq j \leq n$, $m_j \geq 1$ for all j. The* as-dual w.r.t. P and p*, denoted $\mathcal{D}_{as}(P, p)$, is the set of all possible sets $\{N^1(l_{1k_1}), ..., N^1(l_{nk_n})\}$, $1 \leq k_1 \leq m_1, ..., 1 \leq k_n \leq m_n$.*

Unlike the wf-dual, the as-dual contains only negated and double negated atoms.

We can now describe Alg. 3. First, P' and four disjoint sets of rules are initialized, in each of which p appears in the rules in a different form (lines 1-5). Then, the special case of existing a fact p is treated directly by introducing rules in which occurrences of p and $not\,not\,p$ in all rules in R_0 and R_2 (whose head is not p) are omitted (lines 6-9). Alternatively, if there is no fact p, then new rules are introduced by substituting all such occurrences of p and $not\,not\,p$ in a way similar to wGPPE [3] (lines 11-15), i.e., p and $not\,not\,p$ are adequately replaced by the rule bodies in R. Next, the replacement of $not\,p$ is treated, by simply canceling these if there is no rule with head p (lines 16-17) and using the as-dual otherwise (lines 18-24). Note that, unlike the previous section, no special case is necessary for handling potential occurrences of p of any form because P is p-forgettable and in normal form. Finally, all rules containing occurrences of p, $not\,p$, and $not\,not\,p$ are removed (line 26). Note that the steps introducing substitutions for p, $not\,p$, and $not\,not\,p$ in the bodies also handle constraints. Thus, it may happen that the result contains a rule $\bot \leftarrow$ which makes the resulting program permanently inconsistent and cannot be removed. This has similarly been observed in [14].

```
input : $p$-forgettable $P$ in normal form and $p \in B_P$
output: Program $P' = F_{as}(P, p)$
$R := \{r \in P : H(r) = p\}$;
$R_0 := \{r \in P : p \in B(r)\}$;
$R_1 := \{r \in P : \mathit{not}\, p \in B(r)\}$;
$R_2 := \{r \in P : \mathit{not}\,\mathit{not}\, p \in B(r), H(r) \neq p\}$;
$P' := P$;
if $\{p \leftarrow\} \subseteq R$ then
    for $r \in R_i$ s.t. $i = 0$ or $i = 2$ do
        $P' := P' \cup \{H(r) \leftarrow (B(r) \setminus \{N^i(p)\})\}$;
    end
else
    for $r \in R_i$ s.t. $i = 0$ or $i = 2$ do
        for $r_1 \in R$ do
            $P' := P' \cup \{H(r) \leftarrow (B(r) \setminus \{N^i(p)\}) \cup N^i(B(r_1))\}$;
        end
    end
    if $R = \emptyset$ then
        $P' := P' \cup \{H(r) \leftarrow B(r)' : r \in R_1, B(r)' = B(r) \setminus \{\mathit{not}\, p\}\}$;
    else
        for $r \in R_1$ do
            for $D_1 \in \mathcal{D}_{as}(P, p)$ do
                $P' := P' \cup \{H(r) \leftarrow B(r) \setminus \{\mathit{not}\, p\} \cup D_1\}$;
            end
        end
    end
end
$P' := P' \setminus (R \cup R_0 \cup R_1 \cup R_2)$;
```

Algorithm 3: Strong as-forgetting for a single atom p

Example 5. Consider the following program P to illustrate Algs. 2 and 3.

$$r_1 : q \leftarrow \mathit{not}\, p \qquad r_3 : p \leftarrow r, \mathit{not}\,\mathit{not}\, r \qquad r_5 : p \leftarrow \mathit{not}\,\mathit{not}\, p, \mathit{not}\, r, \mathit{not}\,\mathit{not}\, r$$
$$r_2 : p \leftarrow \mathit{not}\, t \qquad r_4 : s \leftarrow \mathit{not}\,\mathit{not}\, p$$

Since P is clearly not in the normal form (r_3, r_5), we first apply Alg. 2 and obtain P'.

$$r_1 : q \leftarrow \mathit{not}\, p \qquad r_2 : p \leftarrow \mathit{not}\, t \qquad r_3' : p \leftarrow r \qquad r_4 : s \leftarrow \mathit{not}\,\mathit{not}\, p$$

Rule r_3 is simplified to r_3' according to lines 4-5 (of Alg. 2), and rule r_5 can simply be omitted due to line 3 (Alg. 2). Note that the resulting program is not only in the normal form, but also p-forgettable. It can then be verified that Alg. 3 returns the following program P'' when forgetting about p from P' (and thus from P).

$$r_1'' : q \leftarrow \mathit{not}\,\mathit{not}\, t, \mathit{not}\, r \qquad r_2'' : s \leftarrow \mathit{not}\, t \qquad r_3'' : s \leftarrow \mathit{not}\,\mathit{not}\, r$$

The rule r_1'' can be obtained from r_1 and the as-dual over r_2 and r_3' (lines 18-24), while the rules r_2'' and r_3'' result from r_4 in combination with r_2 and r_3', respectively (lines 11-15). It can be verified that $M_{as}(P'' \cup R) = M_{as}(P \cup R)$ holds for all R over $\Sigma \setminus \{p\}$.

For example, for $R = \emptyset$, we obtain $M_{as}(P'') = M_{as}(P) = \{\{s\}\}$, and if we consider $R_1 = \{t \leftarrow not\, not\, t\}$, then $M_{as}(P \cup R_1) = M_{as}(P'' \cup R_1) = \{\{q, t\}, \{s\}\}$.

Example 6. Consider only the rules and facts explicitly given in Ex. 1 as P. The result of $F_{as}(P, V)$ with $V = \{punchClock(X) \mid X \in \{mary, peter, john\}\}$ contains precisely three instances of $flexible(X) \leftarrow ustaff(X), not\, not\, professor(X)$. Thus, again, $flexible(mary)$ is derivable right away, and if $professor(john)$ is added later, then $flexible(john)$ becomes derivable as well.

As expected, Alg. 3 always returns a result corresponding to $F_{as}(P, p)$.

Theorem 4. *Given a p-forgettable program P in normal form and $p \in B_P$, Alg. 3 computes $F_{as}(P, p)$.*

For the generalization to forgetting sets of atoms V, we can simply rely on Thm. 2 provided, of course, that the program is in normal form, which can easily be ensured by applying Alg. 2 after each step of forgetting, and that it is p-forgettable for each $p \in V$. If the latter is indeed the case, then we simply forget a set of atoms by forgetting one atom after the other. Note that Thm. 2 also allows us to forget atoms in any order. So, if one $p \in V$ is not p-forgettable immediately, then we may delay it and forget another atom q first which is q-forgettable at that time, thereby potentially modifying the program such that it becomes p-forgettable after q has been forgotten, by, e.g., reducing some rule to a fact for p or canceling a rule with head p and $p \in B^{--}(r)$. In this sense, whenever strong as-forgetting is applicable, then it ensures that **(E)** holds.

Proposition 4. *Strong as-forgetting for programs with double negation satisfies* **(E)**.

Under the same assumption, we can determine the complexity of strong as-forgetting.

Theorem 5. *Given a program P and $p \in B_P$, computing $F_{as}(P, p)$ is in EXPTIME in the number of rules containing occurrences of p and linear in the remaining rules.*

This result for computing $F_{as}(P, p)$ is identical to that of computing $F_{wf}(P, p)$ as obtained in Thm. 3. Indeed, the exponential can be traced to the as-duals, and an argument such as the one following Thm. 3 can be applied.

6 Related Work and Conclusions

We have proposed a new property for forgetting propositional variables, called strong persistence **(SP)**, that guarantees that the semantic dependencies between the extant propositional variables are kept. Since none of the existing approaches for forgetting in LP obeys this **(SP)**, we have introduced a new abstract definition of forgetting, which is closely related to **(SP)** and naturally satisfies a number of other properties previously studied in the literature. We have also studied this new notion of forgetting for the cases of well-founded semantics for normal programs and answer set semantics for programs with double negation, and focused on efficient syntax-based algorithms that effectively only touch the rules in which atoms to be forgotten appear.

Considering the related work, the only other forgetting for the well-founded semantics is [1] which does neither satisfy **(SP)** nor **(SE)**. Indeed, consider the following P which is a simplification of Ex. 1 (with obvious abbreviations):

$$\begin{array}{ccc} flx(m) \leftarrow not\, pC(m) & pC(m) \leftarrow not\, prof(m) & prof(m) \\ flx(j) \leftarrow not\, pC(j) & pC(j) \leftarrow not\, prof(j) & \end{array}$$

Even after forgetting about $\{pC(m), pC(j)\}$, $flx(m)$ should hold, and if $prof(j)$ is later added, then $flx(j)$ should hold as well. All three algorithms in [1] return only an instance of a rule with head $flx(X)$, viz., $flx(m) \leftarrow$. So, adding $prof(j)$ cannot yield the desired result. An advantage of [1] is that the size of the program always shrinks while forgetting. Also, computing the result is in PTIME, but over the entire program.

Regarding answer set semantics, several proposals exist, and we consider the same program P from above for comparison. The work in [19], which is based on syntactic transformations, returns only $prof(m)$ for strong forgetting, and, additionally, the facts $flx(m)$ and $flx(j)$ for weak forgetting. So neither of them satisfies **(SP)** nor actually **(SE)** or **(CP)**. The proposal in [6] satisfies **(CP)**, but not **(SE)**, nor **(SP)**, since in the considered example, $flx(m)$ would persist as a fact, but no rule with $flx(j)$ would be part of the result of forgetting, thus loosing the semantic relation that intuitively says that professors have flexible schedules. In [15], a correspondence based on strong equivalence via HT models [10] is established between a program and its result of forgetting, so **(SE)** is satisfied, but not **(CP)**. Also, the result is not always expressible as a logic program. The recent work in [14] further remedies that, thus satisfying both **(CP)** and **(SE)**, but not **(SP)**. In fact, since $flx(m)$ is part of the only answer set of P, its derivation persists, but, again, no rule with head $flx(j)$ is contained in the result of this forgetting. Finally, the three previous approaches have a worst case complexity of at least coNP on the entire program. But, as indicated for the syntactic transformation of [6] and for [14], the resulting program is in general of exponential size over the entire given program, whereas our approach is exponential only over the rules containing the atom(s) to be forgotten, and linear over the remainder, and does not require any additional model computation.

In terms of future work, we want to investigate the remaining properties presented in [14], including appropriate entailment relations for each semantics. The latter are most likely related to HT logics, which may also give rise to study the semantical relations, e.g., to the proposal in [14], which is based on this logic. Another topic to consider is the extension to other classes of programs and different semantics, e.g., disjunction in rules under answer set semantics, though we conjecture that efficient syntactic methods as investigated here cannot be used effectively for this class of programs and there is no clear counterpart for the well-founded semantics. Finally, we intend to implement our approach to allow testing its efficiency. In that regard, the normal programs resulting from forgetting under the well-founded semantics can be readily used, e.g., in XSB Prolog, while in the case of programs with double negation under answer set semantics, the applicability of ASP solvers to the results can be ensured based on (linear) transformations following ideas on N-acyclicity [6].

Acknowledgments Matthias Knorr and José J. Alferes were partially supported by Fundação para a Ciência e a Tecnologia under project “ERRO – Efficient Reasoning with Rules and Ontologies" (PTDC/EIA-CCO/121823/2010) and Matthias Knorr also by FCT grant SFRH/BPD/86970/2012.

References

1. Alferes, J.J., Knorr, M., Wang, K.: Forgetting under the well-founded semantics. In: Cabalar, P., Son, T.C. (eds.) Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8148, pp. 36–41. Springer (2013)
2. Antoniou, G., Eiter, T., Wang, K.: Forgetting for defeasible logic. In: Bjørner, N., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7180, pp. 77–91. Springer (2012)
3. Brass, S., Dix, J.: Semantics of (disjunctive) logic programs based on partial evaluation. J. Log. Program. 40(1), 1–46 (1999)
4. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Comput. Surv. 33(3), 374–425 (2001)
5. Eiter, T., Fink, M., Woltran, S.: Semantical characterizations and complexity of equivalences in answer set programming. ACM Trans. Comput. Log. 8(3) (2007)
6. Eiter, T., Wang, K.: Semantic forgetting in answer set programming. Artif. Intell. 172(14), 1644–1672 (2008)
7. Gelder, A.V., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM 38(3), 620–650 (1991)
8. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. 9(3-4), 365–385 (1991)
9. Kontchakov, R., Wolter, F., Zakharyaschev, M.: Logic-based ontology comparison and module extraction, with an application to DL-Lite. Artif. Intell. 174(15), 1093–1141 (2010)
10. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Trans. Comput. Log. 2(4), 526–541 (2001)
11. Lifschitz, V., Tang, L.R., Turner, H.: Nested expressions in logic programs. Ann. Math. Artif. Intell. 25(3-4), 369–389 (1999)
12. Lin, F., Reiter, R.: Forget it! In: In Proceedings of the AAAI Fall Symposium on Relevance. pp. 154–159 (1994)
13. Lutz, C., Wolter, F.: Foundations for uniform interpolation and forgetting in expressive description logics. In: Walsh, T. (ed.) IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011. pp. 989–995. IJCAI/AAAI (2011)
14. Wang, Y., Wang, K., Zhang, M.: Forgetting for answer set programs revisited. In: Rossi, F. (ed.) IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013. IJCAI/AAAI (2013)
15. Wang, Y., Zhang, Y., Zhou, Y., Zhang, M.: Forgetting in logic programs under strong equivalence. In: Brewka, G., Eiter, T., McIlraith, S.A. (eds.) Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012. pp. 643–647. AAAI Press (2012)
16. Wang, Z., Wang, K., Topor, R.W., Pan, J.Z.: Forgetting for knowledge bases in DL-Lite. Ann. Math. Artif. Intell. 58(1-2), 117–151 (2010)
17. Woltran, S.: Characterizations for relativized notions of equivalence in answer set programming. In: Alferes, J.J., Leite, J.A. (eds.) JELIA. Lecture Notes in Computer Science, vol. 3229, pp. 161–173. Springer (2004)

18. Wong, K.S.: Forgetting in Logic Programs. Ph.D. thesis, The University of New South Wales (2009)
19. Zhang, Y., Foo, N.Y.: Solving logic program conflict through strong and weak forgettings. Artif. Intell. 170(8-9), 739–778 (2006)
20. Zhang, Y., Zhou, Y.: Knowledge forgetting: Properties and applications. Artif. Intell. 173(16-17), 1525–1537 (2009)
21. Zhou, Y., Zhang, Y.: Bounded forgetting. In: Burgard, W., Roth, D. (eds.) Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011. AAAI Press (2011)