On Syntactic Forgetting Under Uniform Equivalence

Ricardo Gonçalves[1], Tomi Janhunen[2], Matthias Knorr[1(✉)], and João Leite[1]

[1] Universidade Nova de Lisboa, Caparica, Portugal
{rjrg,mkn,jleite}@fct.unl.pt
[2] Tampere University, Tampere, Finland
tomi.janhunen@tuni.fi

Abstract. Forgetting in Answer Set Programming (ASP) aims at reducing the language of a logic program without affecting the consequences over the remaining language. It has recently gained interest in the context of modular ASP where it allows simplifying a program of a module, making it more declarative, by omitting auxiliary atoms or hiding certain atoms/parts of the program not to be disclosed. Unlike for arbitrary programs, it has been shown that forgetting for modular ASP can always be applied, for input, output and hidden atoms, and preserve all dependencies over the remaining language (in line with uniform equivalence). However, the definition of the result is based solely on a semantic characterization in terms of HT-models. Thus, computing an actual result is a complicated process and the result commonly bears no resemblance to the original program, i.e., we are lacking a corresponding syntactic operator. In this paper, we show that there is no forgetting operator that preserves uniform equivalence (modulo the forgotten atoms) between the given program and its forgetting result by only manipulating the rules of the original program that contain the atoms to be forgotten. We then present a forgetting operator that preserves uniform equivalence and is syntactic whenever this is suitable. We also introduce a special class of programs, where syntactic forgetting is always possible, and as a complementary result, establish it as the largest known class where forgetting while preserving all dependencies is always possible.

Keywords: Answer Set Programming · Forgetting · Uniform equivalence

1 Introduction

Forgetting, also known as variable elimination, aims at reducing the language of a knowledge base while preserving all direct and indirect relationships over the remaining language. First studied in the context of classical logic [5,14,27,32,33,39], it gained considerable interest in a wide variety of formalisms (cf. the recent survey [12]) and found applications in, e.g., cognitive robotics [30,31,35],

W. Faber et al. (Eds.): JELIA 2021, LNAI 12678, pp. 297–312, 2021.
https://doi.org/10.1007/978-3-030-75775-5_20

conflict resolution [13,27,28,41], and ontology abstraction and comparison [24–26,38]. In more general terms, its usefulness stems from the fact that auxiliary variables can be eliminated, resulting in a more declarative representation of (certain parts of) a knowledge base, as well as that certain pieces of information can be omitted/hidden for reasons of privacy or legal requirements.

In Answer Set Programming (ASP), forgetting has also been extensively studied, where its non-monotonic nature has created unique challenges resulting in a wide variety of different appproaches [4,10,13,15,18,23,36,37,40,41]. Among the many proposals of operators and desirable properties (cf. the survey on forgetting in ASP [16]), arguably, forgetting in ASP is best captured by *strong persistence* [23], a property which requires that the answer sets of a program and its forgetting result be in correspondence, even in the presence of additional rules over the remaining language. However, it is not always possible to forget and satisfy strong persistence [17,19].

Recently, forgetting has also gained interest in the context of modular ASP [2,9,20,22,34]. In general, modular programming is fundamental to facilitate the creation and reuse of large programs, and modular ASP allows the creation of answer set programs equipped with well-defined input-output interfaces whose semantics is compositional on the individual modules. For modules with input-output interfaces, strong persistence can be relaxed to *uniform persistence* that only varies additional sets of facts (the inputs), and it has been shown that forgetting for modular ASP can always be applied and preserves all dependencies over the remaining language [15].

Uniform persistence is closely related to uniform equivalence, which in turn is closely connected to one of the central ideas of ASP: a problem is specified as an abstract program, and varying instances, represented by sets of facts, are combined with it to obtain concrete solutions. Thus, arguably, uniform persistence seems the better alternative when considering forgetting in ASP in general, but its usage is hindered by the lack of practically usable forgetting operators: the definition of a result in [15] is based solely on an advanced semantic characterization in terms of HT-models, so computing an actual result is a complicated process and the result, though semantically correct w.r.t. uniform persistence, commonly bears no resemblance to the original program. What is missing is a syntactic operator that computes results of forgetting, ideally only by manipulating the rules of the original program that contain the atoms to be forgotten.

Concrete syntactic forgetting operators have been considered infrequently in the literature. Zhang and Foo [41] define two such operators in the form of strong and weak forgetting, but neither of them does even preserve the answer sets of the original program (modulo the forgotten atoms) [13]. Eiter and Wang [13] present a syntactic operator for their semantic forgetting, but it only preserves the answer sets themselves and does not satisfy uniform nor strong persistence. Knorr and Alferes [23] provide an operator that aims at aligning with strong persistence which is not possible in general. Thus, it is only defined for a non-standard class of programs, and cannot be iterated in general, as the operator is not closed for this non-standard class, nor does it satisfy uniform persistence.

Berthold et al. [4] introduce an operator that satisfies strong persistence whenever possible, but it does not satisfy uniform persistence, nor is it closed for the non-standard class defined in [23]. Finally, based on the idea of forks [1], a forgetting operator is provided [3] that introduces so-called anonymous cycles when forgetting in the sense of strong persistence is not possible. However, rather than reducing the language this operator does introduce new auxiliary atoms to remove existing ones, though only in a restricted way. Thus, no syntactic forgetting operator exists in the literature that satisfies uniform persistence.

In this paper, we research whether there exists such a syntactic forgetting operator that satisfies uniform persistence. Somewhat surprisingly, we answer this question negatively in the general case. This raises several questions:

- When is it possible/suitable to forget syntactically while preserving uniform persistence?
- Are there meaningful classes of programs where syntactic forgetting is always possible while preserving uniform persistence?
- Can such an operator be iterated, i.e., is it closed for the class of programs for which it is defined?
- Are there correspondences to existing operators in restricted settings (to clarify relations to related work)?

Our contributions can be summarized as follows:

- We show that there is no forgetting operator that preserves uniform equivalence (modulo the forgotten atoms) between the given program and its forgetting result, by only manipulating the rules of the original program that contain the atoms to be forgotten (as formalized in the property ($\mathbf{SI_u}$)).
- We argue that forgetting an atom for which there are rules of the form $p \leftarrow \mathit{not}\ \mathit{not}\ p$ is indeed not suitable in a syntactic manner, even for cases where such result could still be constructed based alone on the rules of the original program that contain the atoms to be forgotten.
- We present a forgetting operator that preserves uniform equivalence while forgetting, and is syntactic whenever this is suitable. We show that this operator can indeed be iterated in the general case.
- In addition, we present a special case of our operator for stratified programs, without disjunction and loops over (double) negation, and show that syntactic forgetting is always possible while preserving uniform equivalence.
- We also show that, for stratified programs, this operator corresponds to an existing forgetting operator that aims at preserving all dependencies whenever possible, and we establish this class of programs as the largest known class where forgetting while preserving all dependencies is always possible.

The remainder of our paper is structured as follows. We recall relevant notions and notations in Sect. 2. In Sect. 3, we first introduce our operator for stratified programs, before we establish our main impossibility result in the general case and present our general operator in Sect. 4. We conclude in Sect. 5.

2 Preliminaries

Let us first recall relevant notions on logic programs under answer set semantics and forgetting in Answer Set Programming (ASP).

An *(extended) rule* r is an expression of the form

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_l, not\, c_1, \ldots, not\, c_m, not\, not\, d_1, \ldots, not\, not\, d_n, \quad (1)$$

where $a_1, \ldots, a_k$, $b_1, \ldots, b_l, c_1, \ldots, c_m$, and $d_1, \ldots, d_n$ are atoms of a given propositional alphabet $\mathcal{A}$.[1] We also write such rules as

$$H(r) \leftarrow B^+(r), not\, B^-(r), not\, not\, B^{--}(r), \quad (2)$$

where $H(r) = \{a_1, \ldots, a_k\}$, $B^+(r) = \{b_1, \ldots, b_l\}$, $B^-(r) = \{c_1, \ldots, c_m\}$, and $B^{--}(r) = \{d_1, \ldots, d_n\}$, and we will use both forms interchangeably.[2] Given a rule r, $H(r)$ is called the *head* of r, and $B(r) = B^+(r) \cup not\, B^-(r) \cup not\, not\, B^{--}(r)$ the *body* of r, where, for a set L of literals (elements of the form a, $not\, a$, or $not\, not\, a$, for $a \in \mathcal{A}$), $not\, L = \{not\, \ell \mid \ell \in L\}$, where $not\, not\, not\, \ell$ systematically simplifies as $not\, \ell$. An *(extended) logic program* is a finite set of rules. By $\mathcal{A}(P)$ we denote the set of atoms appearing in P and by $\mathcal{C}_e$ the class of extended programs. We call r *disjunctive* if $B^{--}(r) = \emptyset$; *normal* if, additionally, $H(r)$ has at most one element; *Horn* if on top of that $B^-(r) = \emptyset$; and *fact* if also $B^+(r) = \emptyset$. The classes of *disjunctive*, *normal* and *Horn programs*, $\mathcal{C}_d$, $\mathcal{C}_n$, and $\mathcal{C}_H$, are defined as usual. Given a program P and an *interpretation* I, i.e., a set $I \subseteq \mathcal{A}$, the *reduct* P^I is defined as:

$$P^I = \{H(r) \leftarrow B^+(r) \mid r \text{ of the form (2) in } P, B^-(r) \cap I = \emptyset, B^{--}(r) \subseteq I\}.$$

An *HT-interpretation* is a pair $\langle X, Y\rangle$ s.t. $X \subseteq Y \subseteq \mathcal{A}$. Given a program P, an HT-interpretation $\langle X, Y\rangle$ is an *HT-model of* P if $Y \models P$ and $X \models P^Y$, where $\models$ stands for the standard satisfaction relation of classical logic. The set of *all HT-models of* P is denoted by $\mathcal{HT}(P)$, and we admit that the set of HT-models of a program P can be restricted to $\mathcal{A}(P)$ even if $\mathcal{A}(P) \subset \mathcal{A}$. Given a program P, a set of atoms $Y \subseteq \mathcal{A}(P)$ is an *answer set* of P if $\langle Y, Y\rangle \in \mathcal{HT}(P)$ and there is no $X \subset Y$ s.t. $\langle X, Y\rangle \in \mathcal{HT}(P)$. The set of all answer sets of P is denoted by $\mathcal{AS}(P)$. Given a set $V \subseteq \mathcal{A}$, the *V-exclusion* of a set of answer sets (a set of HT-interpretations) $\mathcal{M}$, denoted $\mathcal{M}_{\|V}$, is $\{X \backslash V \mid X \in \mathcal{M}\}$ $(\{\langle X \backslash V, Y \backslash V\rangle \mid \langle X, Y\rangle \in \mathcal{M}\})$. Two programs P_1 and P_2 are *equivalent*, denoted by $P_1 \equiv_n P_2$, if $\mathcal{AS}(P_1) = \mathcal{AS}(P_2)$, *strongly equivalent*, denoted by $P_1 \equiv P_2$, if $\mathcal{AS}(P_1 \cup R) = \mathcal{AS}(P_2 \cup R)$ for any $R \in \mathcal{C}_e$ (alternatively, if $\mathcal{HT}(P_1) = \mathcal{HT}(P_2)$ by [29]), and *uniformly equivalent*, denoted by $P_1 \equiv_u P_2$, if $\mathcal{AS}(P_1 \cup R) = \mathcal{AS}(P_2 \cup R)$, for any set of facts R.

Strongly or uniformly equivalent programs can be syntactically different, e.g., due to the occurrence of non-minimal or tautological rules, i.e., rules that if removed would not affect its HT-models in any way. To facilitate the presentation

[1] Note that double negation is standard in the context of forgetting in ASP.

[2] Thus, there cannot be any duplicates in any of the rule components.

in this paper, and in line with related work, we restrict our considerations to programs in normal form following the definition introduced in [4]. Formally, a rule r in P is *minimal* if there is no rule $r' \in P$ such that $H(r') \subseteq H(r) \wedge B(r') \subset B(r)$ or $H(r') \subset H(r) \wedge B(r') \subseteq B(r)$. We also recall that a rule r is *tautological* if $H(r) \cap B^+(r) \neq \emptyset$, or $B^+(r) \cap B^-(r) \neq \emptyset$, or $B^-(r) \cap B^{--}(r) \neq \emptyset$.

Definition 1. *A program P is in* normal form *if the following conditions hold:*

1. *for every $a \in \mathcal{A}(P)$ and $r \in P$, at most one of a, (not a) or (not not a) is in $B(r)$;*
2. *if $a \in H(r)$, then neither a, nor (not a) are in $B(r)$;*
3. *all rules in P are minimal.*

It is shown in [4], that, for a given program P, a strongly equivalent normal form $NF(P)$ can be obtained in polynomial time.

A *forgetting operator* over a class $\mathcal{C}$ of programs[3] over $\mathcal{A}$ is a partial function $\mathsf{f} : \mathcal{C} \times 2^{\mathcal{A}} \rightarrow \mathcal{C}$ s.t. the *result of forgetting about V from P*, $\mathsf{f}(P, V)$, is a program over $\mathcal{A}(P) \backslash V$, for each $P \in \mathcal{C}$ and $V \subseteq \mathcal{A}$. We denote the domain of f by $\mathcal{C}(\mathsf{f})$. The operator f is called *closed* for $\mathcal{C}' \subseteq \mathcal{C}(\mathsf{f})$ if $\mathsf{f}(P, V) \in \mathcal{C}'$, for every $P \in \mathcal{C}'$ and $V \subseteq \mathcal{A}$. A *class* F *of forgetting operators (over $\mathcal{C}$)* is a set of forgetting operators f s.t. $\mathcal{C}(\mathsf{f}) \subseteq \mathcal{C}$.

Among the many properties introduced for different classes of forgetting operators in ASP [16], *strong persistence* **(SP)** [23] is arguably the one that should intuitively hold, since it imposes the preservation of all original direct and indirect dependencies between atoms not to be forgotten. I.e., closely related to strong equivalence, the answer sets of $\mathsf{f}(P, V)$ correspond to those of P, no matter what programs R over $\mathcal{A} \backslash V$ we add to both. However, as shown in [17,19], there is no forgetting operator that satisfies **(SP)** and that is defined for all pairs $\langle P, V \rangle$, called *forgetting instances*, where P is a program and V is a set of atoms to be forgotten from P. Thus, a relaxation of property **(SP)** was introduced in [15], called *uniform persistence* **(UP)**, that only considers R consisting of facts. We recall both properties, where F is a class of forgetting operators.

(SP) F satisfies *Strong Persistence* if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}(\mathsf{f})$ and $V \subseteq \mathcal{A}$, we have $\mathcal{AS}(\mathsf{f}(P, V) \cup R) = \mathcal{AS}(P \cup R)_{\|V}$, for all programs $R \in \mathcal{C}(\mathsf{f})$ with $\mathcal{A}(R) \subseteq \mathcal{A} \backslash V$.

(UP) F satisfies *Uniform Persistence* if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}(\mathsf{f})$ and $V \subseteq \mathcal{A}$, we have $\mathcal{AS}(\mathsf{f}(P, V) \cup R) = \mathcal{AS}(P \cup R)_{\|V}$, for all sets of facts R with $\mathcal{A}(R) \subseteq \mathcal{A} \backslash V$.

A class of forgetting operators F_{UP} is defined in [15] based on semantic definition over HT-models, that is shown to satisfy this property **(UP)**, and it is shown that an operator exists for that class relying on the countermodels of the semantic characterization in terms of HT-models [7] – a construction previously used for computing concrete results of forgetting for classes of forgetting operators based on HT-models [19,36,37].

[3] In this paper, we only consider the very general class of programs introduced before, but, often, subclasses of it appear in the literature of ASP and forgetting in ASP.

In light of the general impossibility result for **(SP)** for arbitrary context programs R, and the fact that **(UP)** is satisfiable for sets of facts R, one may wonder whether it is possible to find a property that uses programs R in a class in between these two extremes such that it is possible to forget and preserve all dependencies. As it turns out, this is not possible.

Proposition 1. **(UP)** *is the strongest relaxation of* **(SP)** *w.r.t. the class of programs R such that there is a forgetting operator over $\mathcal{C} \supseteq \mathcal{C}_n$ that satisfies it.*

This novel result stresses the importance of **(UP)** and of finding syntactic operators that satisfy this property.

3 Uniform Forgetting from Stratified Programs

In this section, we introduce a syntactic operator for forgetting from a restricted class of programs that aligns with the ideas of uniform persistence by only manipulating the rules of the original program that contain the atoms to be forgotten. We focus first on this restricted class of programs with the intuition to ease the reading and to facilitate comparisons to the few existing syntactic forgetting operators in the literature. As usual, this operator will be defined for forgetting a single atom first. Several atoms can then be forgotten iteratively, after showing that the operator can indeed be iterated.

We start by formalizing our restricted forms of programs, called stratified programs that do not allow cycles over (double) negation nor disjunctions. For that purpose, we introduce notation to refer to all rules in P that include some specific atom in one of its components. Namely, P_p^H refers to the rules r in P with $p \in H(r)$, P_p^+ refers to the rules r in P with $p \in B^+(r)$, P_p^- refers to the rules r in P with $p \in B^-(r)$, and P_p^{--} refers to the rules r in P with $p \in B^{--}(r)$.

Definition 2. *Let P be a logic program. We call P* stratified *if:*

1. *all of its rules of the form (1) are s.t. $k \leq 1$;*
2. *P can be partitioned into disjoint P_i s.t., for each $p \in \mathcal{A}$,*
   (a) *all rules $r \in P_p^H$ occur in one P_i;*
   (b) *if $p \in B^+(r)$ with $r \in P_i$, then $P_p^H \subseteq P_j$ with $j \leq i$;*
   (c) *if $p \in (B^-(r) \cup B^{--}(r))$ with $r \in P_i$, then $P_p^H \subseteq P_j$ with $j < i$.*

This not only avoids cycles over negation between atoms, it also prohibits in the case of normal forms that any atom occurs in more than one part of a rule.

We now proceed towards introducing the new stratified uniform forgetting operator f_{su}. For that purpose, we require some further notation and we introduce it along with motivating examples that provide intuitions on how to obtain desired results. To ease the notation, we usually consider forgetting p from P.

Example 1. Consider P containing the following rules:

$$p \leftarrow s \qquad p \leftarrow not\, q, r \qquad t \leftarrow p \qquad v \leftarrow not\, p$$

If we add $s \leftarrow$ to P, then p becomes derivable, thus also t. So, when forgetting about p, we want to preserve that adding s makes t derivable, which can be achieved by introducing a rule $t \leftarrow s$, i.e. by replacing the body atom p with the body whose rule head is p, in a way quite similar to wGPPE [6]. For the same reason, a rule $t \leftarrow not\, q, r$ should appear in the result of forgetting, passing along that, if q is false and r is true, then t is true. This kind of replacement of p in some body does not directly transfer to the replacement of $not\, p$ in another body. In fact, v will be derivable if none of the bodies of rules with head p is true. For example, we want a rule $v \leftarrow not\, s, not\, not\, q$ in the result of forgetting to capture one such case. It can be verified in that case that $not\, p$ is true if one of the two conjuncts, $not\, s$ and $not\, not\, q$, is true, and false otherwise, and that adding further rules to capture the remaining combinations allows one to preserve the dependency between v and the cases in which $not\, p$ is true. □

To be able to capture these dependencies in the case of negated atoms, [4] extends the ideas of the as-dual from [23] inspired by [13]. The as-dual is a set of conjunctions of literals, each of which can be used to replace some negated atom, but preserves its truth value. Here, and in the following, we identify by $B^{\setminus p}(r) = B(r) \setminus \{p, not\, p, not\, not\, p\}$ the set of body literals of r after removing every occurrence of p.

Definition 3. *Let P be a logic program, $p \in \mathcal{A}(P)$, and $P_p^H = \{r_1, \ldots, r_n\}$. We define the* as-dual of P for p *as follows:*

$$\mathcal{D}_{as}^p(P) = \{not\, \{l_1, \ldots, l_n\} \mid l_i \in B^{\setminus p}(r_i), 1 \leq i \leq n\}.$$

Note that for a stratified program in normal form, $B(r_i)$ cannot contain any form of p, but we prefer to keep this definition general, so that it be applicable in the general case.

Example 2. Recall program P from Example 1. Among these rules, only two contain p in the head, and we obtain $\mathcal{D}_{as}^p(P) = \{\{not\, s, not\, r\}, \{not\, s, not\, not\, q\}\}$. We can verify that whenever all elements in one of these sets are true, p cannot be true in any answer set of P. □

Based on that, we can formalize our first operator.

Definition 4. *Let P be a stratified program over $\mathcal{A}$, $N = NF(P)$ the normal form of P, and $p \in \mathcal{A}$. The result of forgetting about p from P, $\mathsf{f}_{su}(P, p)$, is $NF(S)$ where S is obtained from N as follows:*

1. *replace $r \in N_p^+$ with rules $H(r) \leftarrow B^{\setminus p}(r) \cup B(r_1)$ for each $r_1 \in N_p^H$;*
2. *replace $r \in N_p^{--}$ with rules $H(r) \leftarrow B^{\setminus p}(r) \cup not\, not\, B(r_1)$ for each $r_1 \in N_p^H$;*
3. *replace $r \in N_p^-$ with rules $H(r) \leftarrow B^{\setminus p}(r) \cup D$ s.t. $D \in \mathcal{D}_{as}^p(N_p^H)$;*
4. *omit N_p^H.*

Example 3. Consider P, a slight variation of the program in Example 1.

$$p \leftarrow s \qquad p \leftarrow not\, q, r \qquad t \leftarrow p \qquad v \leftarrow not\, p \qquad w \leftarrow not\, not\, p$$

We have $\mathcal{D}^{p}_{as}(P) = \{\{not\, s, not\, r\}, \{not\, s, not\, not\, q\}\}$ from Example 2 and the following result of forgetting:

$$\begin{array}{lll} t \leftarrow s & w \leftarrow not\, not\, s & v \leftarrow not\, s, not\, not\, q \\ t \leftarrow not\, q, r & w \leftarrow not\, q, not\, not\, r & v \leftarrow not\, s, not\, r \end{array}$$

We can verify that no matter which set of facts over the remaining $\mathcal{A}$ is added to the forgetting result, the induced answer sets coincide with those of the original program modulo the forgotten atoms. □

Note that the normal form of a stratified program is also a stratified program.

Lemma 1. *Given a stratified program P, $NF(P)$ is a stratified program.*

This allows us to show that forgetting an atom p from P using f_{su}, results in a stratified program not mentioning the atom to be forgotten.

Proposition 2. *Let P be a stratified program over signature $\mathcal{A}$ and $p \in \mathcal{A}$. Then $\mathsf{f}_{su}(P, p)$ is a stratified program over $\mathcal{A} \setminus \{p\}$.*

This result is important as it allows us to iterate the operator, which can be used to iteratively forget a set of atoms. For that purpose, we define how such iteration can be achieved for any operator defined for forgetting a single atom.

Definition 5. *Let P be a logic program over Σ, $V = \{v_1, v_2, \ldots, v_n\} \subseteq \mathcal{A}$ an ordered sequence of atoms, and f an operator defined for forgetting a single atom. Then, we define $\mathsf{f}(P, V)$ inductively as follows:*

- $\mathsf{f}(P, \{v_1\}) = \mathsf{f}(P, v_1)$;
- $\mathsf{f}(P, \{v_1, v_2, \ldots, v_n\}) = \mathsf{f}(\mathsf{f}(P, \{v_1\}), \{v_2, \ldots, v_n\})$.

We need to fix an order on the set of atoms to be forgotten (lexicographic for example) to ensure that the result of forgetting is indeed a unique program. This raises the question as to whether the order in which we forget the elements of such a set matters. To answer this question for f_{su}, we first relate to existing work in the literature. In fact, due to the restriction to stratified programs, for many of the syntactic operators presented in the literature, we can show that our operator does coincide with them, though on different levels.

Theorem 1. *Let P be a stratified program and $p \in \mathcal{A}$. We have:*

1. $\mathsf{f}_{su}(P,p) = \mathsf{f}_{SP}(P,p)$ *for* f_{SP} *defined in [4];*
2. $\mathsf{f}_{su}(P,p) \equiv \mathsf{f}_{Sas}(P,p)$ *for* f_{Sas} *defined in [23];*
3. $\mathsf{f}_{su}(P,p) \equiv_n \mathsf{forget}_3(P,p)$ *for* forget_3 *defined in [13] if P does not contain double negation.*

Thus, for stratified programs, the syntactic operator that aims at satisfying **(SP)** whenever possible, f_{SP}, coincides with our operator; the one that aims at satisfying **(SP)** in a rather restricted non-standard setting, f_{Sas}, provides strongly equivalent forgetting results, and the one defined for preserving answer sets, forget_3, only provides equivalent results, and is only defined for disjunctive programs originally, though it is shown in [13] that for N-acyclic programs, i.e., programs that satisfy conditions (a) and (c) for $r \in P_p^{--}$ of Definition 1, double negation can simply be omitted.[4] In the latter two cases, the reason why the correspondence is not stronger lies mainly in the preprocessing applied: the normal form in [23] does not eliminate non-minimal rules, and the transformations applied in [13] to simplify the program based on its answer sets before forgetting do not preserve strong equivalence (e.g. Positive Reduction). There are further syntactic forgetting operators [41], but these do not even preserve equivalence [13], thus no correspondence result exists.

Among the results of Theorem 1, the first one is of particular interest, as it relates to an operator that aims at preserving **(SP)** whenever possible. In the case of stratified programs, we can improve on that premise and show that it is always possible to forget while preserving **(SP)**.

Theorem 2. *The operator* f_{su} *satisfies* **(SP)**.

Since **(SP)** is stronger than **(UP)**, the following corollary is straightforward.

Corollary 1. *The operator* f_{su} *satisfies* **(UP)**.

As shown in [17,19] no class of forgetting operators can satisfy **(SP)** on any class of programs including normal programs. Thus, Theorem 2 is an interesting result in its own right as it presents the first operator for which it has been shown that it can be iterated and satisfies **(SP)** on a class of programs beyond Horn programs, unlike previous work in [4,13,23].

The relevance of this class of programs is further witnessed by the following complementary result, that shows that three classes of forgetting operators [18, 19] do coincide on stratified programs.

Proposition 3. *For stratified programs, the classes* $\mathsf{F_{SP}}$, $\mathsf{F_R}$ *and* $\mathsf{F_M}$ *do coincide.*

This is interesting, as these classes have been shown to satisfy different minimal relaxations of **(SP)** and were assumed to be different in a more general setting, as such coincidence was only known for Horn programs.

Finally, Theorem 2 also helps determine that the order of iteration does not affect the final result w.r.t. strong equivalence.

[4] The semantics in [13] then only considers the minimal models of the resulting program, but for the sake of the comparison of the operators as such, this is irrelevant.

Proposition 4. *Let P be a stratified program over $\mathcal{A}$ and $V_1, V_2 \subseteq \mathcal{A}$. Then,*

$$\mathsf{f}_{su}(\mathsf{f}_{su}(P, V_1), V_2) \equiv \mathsf{f}_{su}(\mathsf{f}_{su}(P, V_2), V_1).$$

Hence, for f_{su}, indeed any order for atoms can be chosen in the sense of Definition 5.

4 Uniform Forgetting in General

In this section, we present the general impossibility result of a syntactic operator that satisfies **(UP)**, and then define an operator that does satisfy **(UP)** and is syntactic whenever suitable. To lift our ideas presented for stratified programs, according to Definition 2, we need to consider in addition how to deal with disjunction and with cycles involving negation. In the following, we first discuss the actual challenges resulting from admitting these and we start with disjunction.

Example 4. Consider forgetting about p from program P just consisting of a single rule $p \vee q \leftarrow$. We have $\mathcal{AS}(P) = \{\{p\}, \{q\}\}$, and thus, according to **(UP)** the answer sets of the forgetting result must be $\{\}$ and $\{q\}$. A program over q with these answer sets is the program containing a single rule $q \leftarrow not\, not\, q$. Though it is not immediately clear how to obtain this program in a syntactic manner, it helps to consider the following program P' consisting of two rules, $p \leftarrow not\, q$ and $q \leftarrow not\, p$. Both programs have the same answer sets, and though it is well-known that they are not strongly equivalent, they are uniformly equivalent. Moreover, the result of forgetting p is exactly the same, and in the case of P', Definition 4 could be applied to obtain precisely that result. □

Building on the notion of semi-shifting [13], we formalize the ideas presented in the previous example to remove rules containing disjunctions including a particular atom, replacing them with rules without disjunction.

Definition 6. *Let P be a logic program and p an atom in $\mathcal{A}(P)$. The result of semi-shifting P w.r.t. p, $SH(P, p)$, is defined as replacing any rule $r \in P$ s.t. $H(r) = p \vee a_1 \vee \cdots \vee a_k$ and $k \geq 1$, by the two rules $p \leftarrow not\, a_1, \ldots, not\, a_k, B(r)$ and $a_1 \vee \cdots \vee a_k \leftarrow not\, p, B(r)$.*

Inspecting Definition 4, we note that if the normal form of P, $N = NF(P)$, is such that $N_p^H \cap N_p^{--} = \emptyset$, then we can apply f_{su} to $SH(P, p)$ for forgetting p. To make this precise, and to make this operator applicable to non-stratified programs, we define a new operator as follows.

Definition 7. *Let P be a program over $\mathcal{A}$, $N = NF(P)$ the normal form of P and $p \in \mathcal{A}$ s.t. $N_p^H \cap N_p^{--} = \emptyset$. We extend f_{su} to this class of programs by:*

$$\mathsf{f}_{du}(P, p) = \mathsf{f}_{su}(SH(P, p), p).$$

We can prove that f_{du} defined over this broader class of programs still satisfies **(UP)** when forgetting a single atom.

Proposition 5. *Let P be a program over $\mathcal{A}$, $N = NF(P)$ its normal form, and $p \in \mathcal{A}$ s.t. $N_p^H \cap N_p^{--} = \emptyset$. Then, for all sets of facts R with $\mathcal{A}(R) \subseteq \mathcal{A}\backslash\{p\}$:*

$$\mathcal{AS}(\mathsf{f}_{du}(P,p) \cup R) = \mathcal{AS}(P \cup R)_{\|\{p\}}.$$

This result shows that we can forget an atom p from a program P in normal form by syntactically manipulating its rules, even if P is not stratified, as long as $P_p^H \cap P_p^{--} = \emptyset$. In that regard, note that restricting to normal or disjunctive programs does not help as f_{du} may introduce double negation.

This brings us to the question of what happens if we want to forget p from a program that contains such rules. The difficulty resides in the fact that a rule $p \leftarrow not\ not\ p$ admits two models, $\{p\}$ and $\{\}$, and so forgetting about p has to correctly propagate this choice to all occurrences of p in rule bodies.

Example 5. Consider forgetting about p from program P containing the rules

$$p \leftarrow not\ not\ p, B(r_1) \qquad r \leftarrow p, B(r_2) \qquad s \leftarrow p, B(r_3) \qquad q \leftarrow not\ p, B(r_4)$$

where the $B(r_i)$ represent the remaining rule bodies. If all these $B(r_i)$ are empty, then, by **(UP)**, the answer sets of P, i.e., $\{p, r, s\}$ and $\{q\}$, should be preserved (modulo p). Moreover, adding, e.g., $\{r \leftarrow\}$ should still admit two answer sets, while adding, e.g., $\{r \leftarrow; s \leftarrow; q \leftarrow\}$ should admit precisely one answer set, which means that no simple syntactic transformation as used so far can achieve this. Instead, we have to look at which rule heads depend on p (r and s) and which on $not\ p$ (q). This information can first be used to create rules to represent the models of the resulting program of forgetting, by combining these opposing rule heads in all possible ways, i.e., either the elements supported by p are true, or those supported by $not\ p$, but not both.

$$r \leftarrow not\ q \qquad s \leftarrow not\ q \qquad q \leftarrow not\ r \qquad q \leftarrow not\ s \tag{3}$$

However, this does not suffice, since we still need to guarantee that the answer sets are preserved in the presence of an additional set R of facts not containing p.

This can be remedied by adding the following rules:

$$r \leftarrow not\ not\ r, not\ not\ s, q \qquad s \leftarrow not\ not\ s, not\ not\ r, q \qquad q \leftarrow not\ not\ q, r, s \tag{4}$$

Now, whenever q is derivable (independently), r and s may both either be simultaneously true or false. This can be generalized to rules with non-empty $B(r_i)$ by adding $\bigcup_i B(r_i)$ for the involved i to each rule mentioned in (3) and (4), i.e., such model generators only apply if the remaining bodies are true. E.g., we obtain $r \leftarrow not\ not\ r, not\ not\ s, q, B(r_1), B(r_2)$ in the case of $r \leftarrow not\ not\ r, not\ not\ s, q$. However, if, e.g., $B(r_3)$ is false, then so is s, and therefore r and s will no longer be simultaneously true, namely, we need to add a rule $r \leftarrow not\ not\ r, q, B(r_1), B(r_2), not\ B(r_3)$, and similarly for the other cases. I.e., in general we have to create rules matching the different possible combinations of true and false rule bodies over the rules containing p in the body. □

This example indicates that we cannot forget p from P in a syntactic manner if $P_p^H \cap P_p^{--}$ is not empty: rather than replacing (possibly negated) occurrences of p in the body with (parts of) the bodies of the rules with head p, a set of rules is created that rebuilds the semantic relations based on that choice between p and *not* p. This hardly resembles the original program in general and the possibly resulting combinatorial representation is not desirable. One could argue that, presumably, at least this problem is restricted to the rules mentioning the atom to be forgotten. I.e., implicitly we have used so far the following property that characterizes the fact that, when forgetting, we can focus just on the rules that contain the atom to be forgotten.

($\mathbf{SI_u}$). A class F of forgetting operators satisfies *Strong Invariance with respect to uniform equivalence* if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}(\mathsf{f})$ and $V \subseteq \mathcal{A}$, we have $\mathsf{f}(P, V) \cup R \equiv_u \mathsf{f}(P \cup R, V)$, for all programs $R \in \mathcal{C}(\mathsf{f})$ with $\mathcal{A}(R) \subseteq \mathcal{A} \backslash V$.

This property is a relaxation of the property strong invariance **(SI)** (see, e.g., [16]), referring to uniform equivalence rather than strong equivalence.

Unfortunately, it turns out that **(UP)** and ($\mathbf{SI_u}$) are in general incompatible.

Theorem 3. *There is no forgetting operator over $\mathcal{C}_e$ that satisfies both* **(UP)** *and* ($\mathbf{SI_u}$).

This shows that the problems observed in Example 5 are in fact a consequence of a more general incompatibility between the two properties: it is in general not possible to have a completely syntactic operator that satisfies **(UP)**.

In light of this result, the rather convoluted hinted construction of a possible result in Example 5, and the fact that forgetting p can be obtained nicely if $P_p^H \cap P_p^{--}$ is empty, we argue that it is not suitable to forget syntactically if $P_p^H \cap P_p^{--}$ is not empty, even in corner cases where such result exists, despite Theorem 3. We thus propose an operator that combines our syntactic approach whenever this is suitable, and use a semantic operator only for the remaining cases.

For this purpose, let f_{UP} be the semantic operator sketched in [15] (based on the countermodel construction [7]), which satisfies **(UP)**.

Definition 8. *Let P be a program over $\mathcal{A}$, $N = NF(P)$ the normal form of P, and $p \in \mathcal{A}$. The result of forgetting about p from P, $\mathsf{f}_u(P, p)$, is $NF(S)$ where:*

$$S = \begin{cases} \mathsf{f}_{du}(P, p) & \text{if } N_p^H \cap N_p^{--} = \emptyset \\ \mathsf{f}_{UP}(P, p) & \text{otherwise} \end{cases}$$

Example 6. As an example for f_{UP}, consider P from Example 5 with all $B(r_i)$ empty. The HT-models of the forgetting result coincide with the HT-models obtained for f_{UP} [15]. The countermodel construction provides 15 rules, including one of those in (4) and the other two with one double negation omitted. Also, variants of the rules in (3) appear together with constraints to obtain the

desired HT-models. This result can be further simplified following work on minimal programs [8], and that, in all, the resulting program is rather similar to ours supporting our stance that in such a case the forgetting result is not truly syntactic. □

The resulting operator is defined for every program P and any atom p to be forgotten, and forgetting p from P according to f_u provides program without p.

Proposition 6. *Let P be a program over signature $\mathcal{A}$ and $p \in \mathcal{A}$. Then $\mathsf{f}_u(P, p)$ is a program over $\mathcal{A} \setminus \{p\}$.*

This result naturally allows the extension of f_u to sets of atoms using Definition 5. We are able to prove that f_u indeed satisfies **(UP)**.

Theorem 4. *The operator f_u satisfies* **(UP)**.

We have thus defined the first general operator that can be iterated and that satisfies **(UP)** and that, whenever this is possible and suitable, produces a result of forgetting that corresponds to a syntactic manipulation of the rules of the original program that contain the atoms to be forgotten.

In addition, we can show that the order of iteration does not affect the final result w.r.t. uniform equivalence.

Proposition 7. *Let P be a program over $\mathcal{A}$ and $V_1, V_2 \subseteq \mathcal{A}$. Then,*

$$\mathsf{f}_u(\mathsf{f}_u(P, V_1), V_2) \equiv_u \mathsf{f}_u(\mathsf{f}_u(P, V_2), V_1).$$

Thus, we can forget a set of atoms in any order, which may allow us to prioritize atoms where syntactic forgetting is possible and suitable, resulting in a uniformly equivalent program, but possibly syntactically closer to the original program.

Finally, it is not surprising that computing such forgetting results is worst-case exponential in the size of the input program, due to the as-dual in the case of f_{du} and the computation of the countermodels in the case of f_{UP} (for f_{du} alone exponential in the size of rules mentioning the atom to be forgotten).

5 Conclusions

In this paper, we have investigated syntactic forgetting under uniform equivalence in modular ASP. We have studied this problem first for stratified programs and shown that even strong equivalence is preserved while forgetting, establishing interesting results to existing operators of forgetting and novel results as to when these coincide. We then considered the general case and showed that it is not always possible to syntactically forget using only the rules that mention the atom(s) to be forgotten while preserving uniform persistence. This can be traced back to rules of the form $p \leftarrow \mathit{not}\, \mathit{not}\, p$ which are known to break the antichain property of answer sets. We argue that syntactic forgetting in such cases is not suitable, as it would result in a semantic reconstruction of possible

ways of assigning truth to the involved atoms. We thus establish an operator that is syntactic whenever this is possible and suitable, and we show that this operator can be iterated and preserves strong persistence.

To add to the discussion of related work in the introduction, we note that [4], which is closest in spirit to our work, provides an operator that is syntactic, as the class for which it is defined satisfies strong invariance, i.e., it is amenable to restrict forgetting only to the rules that mention the atom(s) to be forgotten. Still, it has been observed that the construction is particularly complicated whenever there are rules of the form $p \leftarrow not\ not\ p$, often with rules in the forgetting result that are not easily associated to the rules in the original program.

Possible future work includes investigating the precise relationship of **(UP)** to the notion of relativized uniform equivalence [11], to gain further insights into semantic operators, study in more detail minimization of logic programs [8] for simplifying the results of semantic operators, and investigate other impossibility results in the context of ASP for similarities, such as embedding ASP into propositional logic [21], where knowing individual rules does not suffice and a more holistic view is required.

Acknowledgments. We thank the anonymous reviewers for their helpful comments. Authors R. Gonçalves, M. Knorr, and J. Leite were partially supported by FCT project FORGET (PTDC/CCI-INF/32219/2017) and by FCT project NOVA LINCS (UIDB/04516/2020). T. Janhunen was partially supported by the Academy of Finland projects ETAIROS (251170) and AI-ROT (335718).

References

1. Aguado, F., Cabalar, P., Fandinno, J., Pearce, D., Pérez, G., Vidal, C.: Forgetting auxiliary atoms in forks. Artif. Intell. **275**, 575–601 (2019)
2. Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls and use of ensembles in modular answer set programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 376–390. Springer, Heidelberg (2006). https://doi.org/10.1007/11799573_28
3. Berthold, M., Gonçalves, R., Knorr, M., Leite, J.: Forgetting in answer set programming with anonymous cycles. In: Moura Oliveira, P., Novais, P., Reis, L.P. (eds.) EPIA 2019. LNCS (LNAI), vol. 11805, pp. 552–565. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30244-3_46
4. Berthold, M., Gonçalves, R., Knorr, M., Leite, J.: A syntactic operator for forgetting that satisfies strong persistence. Theory Pract. Log. Program. **19**(5–6), 1038–1055 (2019)
5. Bledsoe, W.W., Hines, L.M.: Variable elimination and chaining in a resolution-based prover for inequalities. In: Bibel, W., Kowalski, R. (eds.) CADE 1980. LNCS, vol. 87, pp. 70–87. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10009-1_7
6. Brass, S., Dix, J.: Semantics of (disjunctive) logic programs based on partial evaluation. J. Log. Program. **40**(1), 1–46 (1999)
7. Cabalar, P., Ferraris, P.: Propositional theories are strongly equivalent to logic programs. TPLP **7**(6), 745–759 (2007)

8. Cabalar, P., Pearce, D., Valverde, A.: Minimal logic programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 104–118. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74610-2_8
9. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Modular nonmonotonic logic programming revisited. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 145–159. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02846-5_16
10. Delgrande, J.P., Wang, K.: A syntax-independent approach to forgetting in disjunctive logic programs. In: Bonet, B., Koenig, S. (eds.) Proceedings of AAAI, pp. 1482–1488. AAAI Press (2015)
11. Eiter, T., Fink, M., Woltran, S.: Semantical characterizations and complexity of equivalences in answer set programming. ACM Trans. Comput. Log. **8**(3) (2007)
12. Eiter, T., Kern-Isberner, G.: A brief survey on forgetting from a knowledge representation and reasoning perspective. Künstliche Intell. **33**(1), 9–33 (2019)
13. Eiter, T., Wang, K.: Semantic forgetting in answer set programming. Artif. Intell. **172**(14), 1644–1672 (2008)
14. Gabbay, D.M., Schmidt, R.A., Szalas, A.: Second Order Quantifier Elimination: Foundations, Computational Aspects and Applications. College Publications (2008)
15. Gonçalves, R., Janhunen, T., Knorr, M., Leite, J., Woltran, S.: Forgetting in modular answer set programming. In: AAAI, pp. 2843–2850. AAAI Press (2019)
16. Goncalves, R., Knorr, M., Leite, J.: The ultimate guide to forgetting in answer set programming. In: Baral, C., Delgrande, J., Wolter, F. (eds.) Proceedings of KR, pp. 135–144. AAAI Press (2016)
17. Gonçalves, R., Knorr, M., Leite, J.: You can't always forget what you want: on the limits of forgetting in answer set programming. In: Fox, M.S., Kaminka, G.A. (eds.) Proceedings of ECAI, pp. 957–965. IOS Press (2016)
18. Gonçalves, R., Knorr, M., Leite, J., Woltran, S.: When you must forget: beyond strong persistence when forgetting in answer set programming. TPLP **17**(5–6), 837–854 (2017)
19. Gonçalves, R., Knorr, M., Leite, J., Woltran, S.: On the limits of forgetting in answer set programming. Artif. Intell. **286**, 103307 (2020)
20. Harrison, A., Lierler, Y.: First-order modular logic programs and their conservative extensions. TPLP **16**(5–6), 755–770 (2016)
21. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. J. Appl. Non Class. Logics **16**(1–2), 35–86 (2006)
22. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. J. Artif. Intell. Res. (JAIR) **35**, 813–857 (2009)
23. Knorr, M., Alferes, J.J.: Preserving strong equivalence while forgetting. In: Fermé, E., Leite, J. (eds.) JELIA 2014. LNCS (LNAI), vol. 8761, pp. 412–425. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11558-0_29
24. Konev, B., Ludwig, M., Walther, D., Wolter, F.: The logical difference for the lightweight description logic EL. J. Artif. Intell. Res. (JAIR) **44**, 633–708 (2012)
25. Konev, B., Lutz, C., Walther, D., Wolter, F.: Model-theoretic inseparability and modularity of description logic ontologies. Artif. Intell. **203**, 66–103 (2013)
26. 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)
27. Lang, J., Liberatore, P., Marquis, P.: Propositional independence: formula-variable independence and forgetting. J. Artif. Intell. Res. (JAIR) **18**, 391–443 (2003)

28. Lang, J., Marquis, P.: Reasoning under inconsistency: a forgetting-based approach. Artif. Intell. **174**(12–13), 799–823 (2010)
29. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Trans. Comput. Log. **2**(4), 526–541 (2001)
30. Lin, F., Reiter, R.: How to progress a database. Artif. Intell. **92**(1–2), 131–167 (1997)
31. Liu, Y., Wen, X.: On the progression of knowledge in the situation calculus. In: Walsh, T. (ed.) Proceedings of IJCAI, pp. 976–982. IJCAI/AAAI (2011)
32. Middeldorp, A., Okui, S., Ida, T.: Lazy narrowing: strong completeness and eager variable elimination. Theoret. Comput. Sci. **167**(1&2), 95–130 (1996)
33. Moinard, Y.: Forgetting literals with varying propositional symbols. J. Log. Comput. **17**(5), 955–982 (2007)
34. Oikarinen, E., Janhunen, T.: Achieving compositionality of the stable model semantics for Smodels programs. TPLP **8**(5–6), 717–761 (2008)
35. Rajaratnam, D., Levesque, H.J., Pagnucco, M., Thielscher, M.: Forgetting in action. In: Baral, C., Giacomo, G.D., Eiter, T. (eds.) Proceedings of KR. AAAI Press (2014)
36. Wang, Y., Wang, K., Zhang, M.: Forgetting for answer set programs revisited. In: Rossi, F. (ed.) Proceedings of IJCAI, pp. 1162–1168. IJCAI/AAAI (2013)
37. Wang, Y., Zhang, Y., Zhou, Y., Zhang, M.: Knowledge forgetting in answer set programming. J. Artif. Intell. Res. (JAIR) **50**, 31–70 (2014)
38. 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)
39. Weber, A.: Updating propositional formulas. In: Expert Database Conference, pp. 487–500 (1986)
40. Wong, K.S.: Forgetting in logic programs. Ph.D. thesis, The University of New South Wales (2009)
41. Zhang, Y., Foo, N.Y.: Solving logic program conflict through strong and weak forgettings. Artif. Intell. **170**(8–9), 739–778 (2006)