TPLP **19** (5–6): 1038–1055, 2019. © Cambridge University Press 2019
doi:10.1017/S1471068419000346

A Syntactic Operator for Forgetting that Satisfies Strong Persistence

MATTI BERTHOLD
Technische Universität Dresden and Universität Leipzig, Germany

RICARDO GONÇALVES, MATTHIAS KNORR and JOÃO LEITE
NOVA LINCS, Departamento de Informática, Universidade Nova de Lisboa, Portugal

submitted 30 July 2019; accepted 31 July 2019

Abstract

Whereas the operation of *forgetting* has recently seen a considerable amount of attention in the context of Answer Set Programming (ASP), most of it has focused on theoretical aspects, leaving the practical issues largely untouched. Recent studies include results about what sets of properties operators should satisfy, as well as the abstract characterization of several operators and their theoretical limits. However, no concrete operators have been investigated.

In this paper, we address this issue by presenting the first concrete operator that satisfies *strong persistence* – a property that seems to best capture the essence of forgetting in the context of ASP – whenever this is possible, and many other important properties. The operator is syntactic, limiting the computation of the forgetting result to manipulating the rules in which the atoms to be forgotten occur, naturally yielding a forgetting result that is close to the original program.

KEYWORDS: Answer Set Programming, Forgetting, Strong Persistence, Syntactic Operator

1 Introduction

Unlike belief change operations such as *revision*, *update* and *contraction*, which have deserved ample attention for over three decades now, only recently has the Knowledge Representation and Reasoning research community recognized the operation of *forgetting* or *variable elimination* – i.e., removing from a knowledge base information that is no longer needed – as a critical operation, as witnessed by the amount of work developed for different logical formalisms (cf. the survey (Eiter and Kern-Isberner 2019)).

The operation of *forgetting* is most useful when we wish to eliminate (temporary) variables introduced to represent auxiliary concepts, with the goal of restoring the declarative nature of some knowledge base, or just to simplify it. Furthermore, it is becoming increasingly necessary to properly deal with legal and privacy issues, including, for example, to enforce the new EU General Data Protection Regulation which includes the *Right to be Forgotten*. The operation of *forgetting* has been applied in cognitive robotics (Lin and Reiter 1997; Liu and Wen 2011; Rajaratnam et al. 2014), resolving conflicts (Lang et al. 2003; Zhang and Foo 2006; Eiter and Wang 2008; Lang and Marquis 2010), ontology abstraction and comparison (Wang et al. 2010; Kontchakov et al. 2010), among others (Zhang and Zhou 2009; Alferes et al. 2013), which further witnesses its importance.

This has also triggered an increasing interest in the development of forgetting operators for Logic Programs (LP) in particular Answer Set Programming (ASP)

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

(Gelfond and Lifschitz 1991), see, e.g., (Zhang and Foo 2006; Eiter and Wang 2008; Wang et al. 2013; Knorr and Alferes 2014; Wang et al. 2014; Delgrande and Wang 2015; Gonçalves et al. 2016b; Gonçalves et al. 2017; Gonçalves et al. 2019) or (Gonçalves et al. 2016a; Leite 2017) for a recent survey.

Despite the significant amount of interest in *forgetting* in the context of ASP, most research has focused on theoretical aspects, such as semantically characterizing various operators and investigating their properties. We now have a rather good understanding of the landscape of properties and operators, and their theoretical limits. Notably, it is now well established that *strong persistence* (Knorr and Alferes 2014) – a property inspired by *strong equivalence*, which requires that there be a correspondence between the answer sets of a program before and after forgetting a set of atoms, and that such correspondence be preserved in the presence of additional rules not containing the atoms to be forgotten – best captures the essence of forgetting in the context of ASP, even if it is not always possible to forget some atom from a program while obeying such property (Gonçalves et al. 2016b). Also, there exists a semantic characterisation of the class of forgetting operators that satisfy *strong persistence* whenever that is possible.

Most of these theoretical results, however, have not been accompanied by investigations into the practical aspects of these operators. The characterization of the operators found in the literature is usually provided through the set of HT-models of the result, usually appealing to some general method to generate a concrete program from those models, such as the method that relies on the notion of counter-models in HT (Cabalar and Ferraris 2007) or one of its variations (Wang et al. 2013; Wang et al. 2014). Whereas relying on such methods to produce a concrete program is important in the sense of being a proof that such a program exists, it suffers from several severe drawbacks when used in practice:

- In general, it produces programs with a very large number of rules. For example, if we use the counter-models method variation in (Wang et al. 2013; Wang et al. 2014) to determine the result of forgetting about atom q from the following program

$$d \leftarrow not\, c \qquad a \leftarrow q \qquad q \leftarrow b$$

  while satisfying *strong persistence*, the result would have 20 rules, even if strongly equivalent to the much simpler program

$$d \leftarrow not\, c \qquad a \leftarrow b$$

- Even if we replace the resulting program with some equivalent canonical one at the expense of the considerable additional computational cost (cf., (Inoue and Sakama 1998; Inoue and Sakama 2004; Cabalar et al. 2007; Slota and Leite 2011)), its syntactic form would most likely bear no real connection to the syntactic form of the original program, compromising its declarative nature. Even the syntactic form of rules that are not related to the atoms to be forgotten would likely be compromised by such method.
- It requires the computation of HT-models, which does not come without a significant cost.

This naturally suggests the need to investigate *syntactic* forgetting operators, i.e., forgetting operators that produce the result through a syntactic manipulation of the original program, thus not requiring the computation of HT-models, with the aim to keep the result as close as possible to the original program.

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

Despite the fact that most research on forgetting has focused on its semantic characterisation, there are some exceptions in the literature, notably the syntactic forgetting operators described in (Zhang and Foo 2006; Eiter and Wang 2008; Knorr and Alferes 2014). However, for different reasons, the early approaches are too simplistic (Zhang and Foo 2006; Eiter and Wang 2008), in that they hardly satisfy any of the desirable properties of forgetting in ASP (Gonçalves et al. 2016a), while the recent one has a very restricted applicability (Knorr and Alferes 2014) (cf. Sect. 4 for a detailed discussion). For example, the syntactic operators in (Zhang and Foo 2006) lack the semantic underpinning and are too syntax-sensitive, not even preserving existing answer sets while forgetting. The purely syntactic operator in (Eiter and Wang 2008), dubbed forget_3, designed to preserve the answer sets, falls short of adequately distinguishing programs that, though having the same answer sets, are not strongly equivalent, which prevents it from satisfying *strong persistence* even in very simple cases. Additionally, forget_3 is defined for disjunctive logic programs, but may produce programs with double negation, hence preventing its iterative use. Even from a syntactic perspective, forget_3 performs unnecessary changes to the initial program, which may even affect rules that do not contain atom(s) to be forgotten. Finally, the operator in (Knorr and Alferes 2014) is only applicable to a very particular narrow non-standard subclass of programs, which excludes many cases where it is possible to forget while preserving *strong persistence*, and, in general, cannot be iterated either.

In this paper, we address these issues and present a concrete forgetting operator, defined for the entire class of extended logic programs (disjunctive logic programs with double negation), that allows one to forget a single atom and produces a program in the same class. The operator is syntactic, limiting the computation of the result of forgetting to manipulating the rules in which the atom to be forgotten occurs, thus naturally yielding a resulting program that is close to the original program, while not requiring expensive computation of HT-models. Crucially, the operator satisfies *strong persistence* whenever this is possible, making it the first such operator.

Whereas one might argue that we still need to compute the HT-models in order to determine whether it is possible to satisfy *strong persistence*, hence eliminating one of the advantages mentioned above, this is significantly mitigated by the fact that the operator can always be used, even in cases where *strong persistence* cannot be satisfied, while still exhibiting desirable properties. In other words, if you *must* forget, this operator will ensure the "*perfect*" result whenever *strong persistence* is possible, and a "*good*" result when "*perfection*" is not possible. In this paper, we investigate the operators' properties not only when *strong persistence* is possible, but also when that is not the case. In addition, we characterise a class of programs – dubbed *q-forgettable* – from which it is always possible to forget (atom q) while satisfying *strong persistence*, whose membership can be checked in linear time.

We proceed as follows: in Sec. 2, we recall extended logic programs,and briefly recap important notions related to forgetting in ASP; then, we begin Sec. 3 by drawing some general considerations on syntactic forgetting and introduce some necessary notions in Sec. 3.1, introduce our new operator together with several illustrative examples in Sec. 3.2, followed by a discussion of its properties in Sec. 3.3; in Sec. 4 we discuss related work; and in Sec. 5, we conclude. All proofs of the established results and a detailed comparison with relevant syntactic operators in the literature can be found in the extended version (Berthold et al. 2019).

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

2 Preliminaries

In this section, we recall necessary notions on answer set programming and forgetting.

We assume a *propositional signature* Σ. A *logic program* P over Σ is a finite set of *rules* of the form $a_1 \vee \ldots \vee a_k \leftarrow b_1, ..., b_l, not\, c_1, ..., not\, c_m, not\, not\, d_1, ..., not\, not\, d_n$, where all $a_1, \ldots, a_k$, $b_1, \ldots, b_l, c_1, \ldots, c_m$, and $d_1, \ldots, d_n$ are atoms of Σ. Such rules r are also written more succinctly as $H(r) \leftarrow B^+(r), not\, B^-(r), not\, not\, B^{--}(r)$, 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. 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 A of atoms, $not\, A = \{not\, q \colon q \in A\}$ and $not\, not\, A = \{not\, not\, q : q \in A\}$. We term the elements in $B(r)$ *(body) literals*. $\Sigma(P)$ and $\Sigma(r)$ denote the set of atoms appearing in P and r, respectively. The general class of logic programs we consider[1] includes a number of special kinds of rules r: if $n = 0$, then we call r *disjunctive*; if, in addition, $k \leq 1$, then r is *normal*; if on top of that $m = 0$, then we call r *Horn*, and *fact* if also $l = 0$. The classes of *disjunctive*, *normal* and *Horn programs* are defined resp. as a finite set of disjunctive, normal, and Horn rules. Given a program P and an *interpretation*, i.e., a set $I \subseteq \Sigma$ of atoms, the *reduct* of P given I, is defined as $P^I = \{H(r) \leftarrow B^+(r) : r \in P \text{ such that } B^-(r) \cap I = \emptyset \text{ and } B^{--}(r) \subseteq I\}$.

An *HT-interpretation* is a pair $\langle X, Y \rangle$ s.t. $X \subseteq Y \subseteq \Sigma$. 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 $\models$ denotes the standard consequence relation for classical logic. We admit that the set of HT-models of a program P is restricted to $\Sigma(P)$ even if $\Sigma(P) \subset \Sigma$. We denote by $\mathcal{HT}(P)$ the set of *all HT-models of* P. A set of atoms Y is an *answer set* of P if $\langle Y, Y \rangle \in \mathcal{HT}(P)$, but there is no $X \subset Y$ such that $\langle X, Y \rangle \in \mathcal{HT}(P)$. The set of all answer sets of P is denoted by $\mathcal{AS}(P)$. We say that two programs P_1, P_2 are *equivalent* if $\mathcal{AS}(P_1) = \mathcal{AS}(P_2)$ and *strongly equivalent*, denoted by $P_1 \equiv P_2$, if $\mathcal{AS}(P_1 \cup R) = \mathcal{AS}(P_2 \cup R)$ for any program R. It is well-known that $P_1 \equiv P_2$ exactly when $\mathcal{HT}(P_1) = \mathcal{HT}(P_2)$ (Lifschitz et al. 2001). Given a set $V \subseteq \Sigma$, the *V-exclusion* of a set of answer sets (a set of HT-interpretations) $\mathcal{M}$, denoted $\mathcal{M}_{\| V}$, is $\{X \backslash V \mid X \in \mathcal{M}\}$ ($\{\langle X \backslash V, Y \backslash V \rangle \mid \langle X, Y \rangle \in \mathcal{M}\}$).

A *forgetting operator* over a class $\mathcal{C}$ of programs[2] over Σ is a partial function $\mathsf{f} : \mathcal{C} \times 2^{\Sigma} \to \mathcal{C}$ s.t. the *result of forgetting about V from P*, $\mathsf{f}(P, V)$, is a program over $\Sigma(P) \backslash V$, for each $P \in \mathcal{C}$ and $V \subseteq \Sigma$. 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 $V \subseteq \Sigma$. A *class* F *of forgetting operators (over $\mathcal{C}$)* is a set of forgetting operators f s.t. $\mathcal{C}(\mathsf{f}) \subseteq \mathcal{C}$.

Arguably, among the many properties introduced for different classes of forgetting operators in ASP (Gonçalves et al. 2016a), *strong persistence* (Knorr and Alferes 2014) is the one that should intuitively hold, since it imposes the preservation of all original direct and indirect dependencies between atoms not to be forgotten. In the following, F is a class of forgetting operators.

[1] Note that the use of double negation is very common in the literature of forgetting for ASP (cf. the recent survey (Gonçalves et al. 2016a)), and in fact necessary as argued in (Eiter and Wang 2008; Wang et al. 2013; Knorr and Alferes 2014; Wang et al. 2014). This does not pose a problem as double negation can be used to represent choice rules, and tools such as clingo support the syntax of double negation.

[2] 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.

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

(SP) F satisfies *Strong Persistence* if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}(\mathsf{f})$ and $V \subseteq \Sigma$, 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 $\Sigma(R) \subseteq \Sigma \backslash V$.

Thus, **(SP)** requires that the answer sets of $\mathsf{f}(P,V)$ correspond to those of P, no matter what programs R over $\Sigma \backslash V$ we add to both, which is closely related to the concept of strong equivalence. Among the many properties implied by **(SP)** (Gonçalves et al. 2016a), **(SI)** indicates that rules not mentioning atoms to be forgotten can be added before or after forgetting.

(SI) F satisfies *Strong (addition) Invariance* if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}$ and $V \subseteq \Sigma$, we have $\mathsf{f}(P,V) \cup R \equiv \mathsf{f}(P \cup R, V)$ for all programs $R \in \mathcal{C}$ with $\Sigma(R) \subseteq \Sigma \backslash V$.

However, it was shown in (Gonçalves et al. 2016b) that 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. Given this general impossibility result, **(SP)** was defined for concrete forgetting instances.

A forgetting operator f over $\mathcal{C}$ satisfies $\mathbf{(SP)}_{\langle P,V\rangle}$, for $\langle P, V\rangle$ a forgetting instance over $\mathcal{C}$, if $\mathcal{AS}(\mathsf{f}(P,V) \cup R) = \mathcal{AS}(P \cup R)_{\|V}$, for all programs $R \in \mathcal{C}$ with $\Sigma(R) \subseteq \Sigma \backslash V$.

A sound and complete criterion Ω was presented to characterize when it is not possible to forget while satisfying $\mathbf{(SP)}_{\langle P,V\rangle}$. An instance $\langle P, V\rangle$ *satisfies criterion* Ω if there is $Y \subseteq \Sigma \backslash V$ such that the set of sets $\mathcal{R}^{Y}_{\langle P,V\rangle} = \{R^{Y,A}_{\langle P,V\rangle} \mid A \in Rel^{Y}_{\langle P,V\rangle}\}$ is non-empty and has no least element, where

$$R^{Y,A}_{\langle P,V\rangle} = \{X \backslash V \mid \langle X, Y \cup A\rangle \in \mathcal{HT}(P)\}$$
$$Rel^{Y}_{\langle P,V\rangle} = \{A \subseteq V \mid \langle Y \cup A, Y \cup A\rangle \in \mathcal{HT}(P) \text{ and } \nexists A' \subset A \text{ s.t. } \langle Y \cup A', Y \cup A\rangle \in \mathcal{HT}(P)\}.$$

This technical criterion was shown to be sound and complete, i.e., it is not possible to forget about a set of atoms V from a program P exactly when $\langle P, V\rangle$ satisfies Ω. A corresponding class of forgetting operators, F_{SP}, was introduced (Gonçalves et al. 2016b).

$$\mathsf{F}_{\mathsf{SP}} = \{\mathsf{f} \mid \mathcal{HT}(\mathsf{f}(P,V)) = \{\langle X, Y\rangle \mid Y \subseteq \Sigma(P) \backslash V \wedge X \in \bigcap \mathcal{R}^{Y}_{\langle P,V\rangle}\}, \text{ for all } P \in \mathcal{C}(\mathsf{f}) \text{ and } V \subseteq \Sigma\}.$$

It was shown that every operator in F_{SP} satisfies $\mathbf{(SP)}_{\langle P,V\rangle}$ for instances $\langle P, V\rangle$ that do not satisfy Ω. Moreover, in the detailed study of the case where Ω is satisfied (Gonçalves et al. 2017), it was shown that the operators in F_{SP} preserve all answer sets, even in the presence of an additional program without the atoms to be forgotten. This makes F_{SP} an ideal choice to forgetting while satisfying **(SP)** whenever possible. Whereas F_{SP} is only defined semantically, i.e., it only specifies the HT-models that a result of forgetting a set of atoms from program should have, a specific program could be obtained from that set of HT-models based on its counter-models (Cabalar and Ferraris 2007) – a construction previously adapted for computing concrete results of forgetting for classes of forgetting operators based on HT-models (Wang et al. 2013; Wang et al. 2014).

3 A Syntactic Operator

In this section, we present a syntactic forgetting operator f_{SP} that satisfies **(SP)** whenever possible. We start with some general considerations on syntactic forgetting and introduce some necessary notions in Sec. 3.1. Then, we introduce our new operator f_{SP} together with explanatory examples in Sec. 3.2, followed by a discussion of its properties in Sec. 3.3.

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

3.1 On Syntactic Forgetting in ASP

One fundamental idea in syntactic forgetting (in ASP) is to replace the occurrences of an atom to be forgotten in the body of some rule with the body of a rule whose head is the atom to be forgotten. This can be rooted in the weak generalized principle of partial evaluation wGPPE (Brass and Dix 1999) and aligns with the objective to preserve answer sets, no matter which rules over the remaining atoms are to be added.

Example 1
Consider program $P = \{t \leftarrow q; v \leftarrow not\, q; q \leftarrow s; q \leftarrow w\}$ from which we want to forget about q. We claim that the following should be a result of forgetting about q.

$$t \leftarrow s \qquad\qquad t \leftarrow w \qquad\qquad v \leftarrow not\, s, not\, w$$

Since t depends on q, and q depends on s as well as on w, we want to preserve these dependencies without mentioning q, thus creating two rules in which q is replaced by s and w, respectively. This way, whenever a set R of rules allow us to derive s (or w), then t must occur in all answer sets of $P \cup R$ as well as of $\mathsf{f}(P, \{q\}) \cup R$. At the same time, since v depends on $not\, q$, and q would be false whenever both s and w are false, we capture this in a rule that represents this dependency.

As we will see later, these natural ideas provide the foundation for the syntactic operator we are going to define, even though several adjustments have to be made to be applicable to all programs, and to ensure that the operator satisfies **(SP)** whenever possible. Still, even in such simplified examples, the occurrence of certain rules that can be considered redundant would complicate this idea of syntactic forgetting. If $q \leftarrow q$ occurred in program P, then we would certainly not want to replace q by q and add $t \leftarrow q$. To avoid the problems caused by redundant rules and redundant parts in rules, we simplify the program upfront, and similarly to (Knorr and Alferes 2014) and previous related work Inoue and Sakama 1998; Inoue and Sakama 2004; Cabalar et al. 2007; Slota and Leite 2011), we adopt a normal form that avoids such redundancies, but is otherwise syntactically identical to the original program. There are two essential differences to the normal form considered in (Knorr and Alferes 2014). First, contrarily to (Knorr and Alferes 2014), our normal form applies to programs with disjunctive heads. Moreover, we eliminate non-minimal rules (Brass and Dix 1999), which further strengthens the benefits of using normal forms.

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
Let P be a program. We say that P is in *normal form* if the following conditions hold:

- for every $a \in \Sigma(P)$ and $r \in P$, at most one of a, *not* a or *not not* a is in $B(r)$;
- if $a \in H(r)$, then neither a, nor *not* a are in $B(r)$;
- all rules in P are minimal.

The next definition shows how to transform any program into one in normal form.

Definition 2
Let P be a program. The normal form $NF(P)$ is obtained from P by:

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

1. removing all tautological rules;
2. removing all atoms a from $B^{--}(r)$ in the remaining rules r, whenever $a \in B^{+}(r)$;
3. removing all atoms a from $H(r)$ in the remaining rules r, whenever $a \in B^{-}(r)$;
4. removing from the resulting program all rules that are not minimal.

Note that the first item of Def. 1 is ensured by conditions 1 and 2 of Def. 2, the second by 1 and 3, and the third by condition 4. We can show that the construction of $NF(P)$ is correct.

Proposition 1
Let P be a program. Then, $NF(P)$ is in normal form and is strongly equivalent to P.

In addition, $NF(P)$ can be computed in at most quadratic time in terms of the number of rules in P (as ensuring minimality requires comparing all n rules with each other).

Proposition 2
Let P be a program. Then, the normal form $NF(P)$ can be computed in PTime.

Thus for the remainder of the paper, we only consider programs in normal form, as this can be efficiently computed and is syntactically equal to the original program apart from redundancies.

Coming back to Ex. 1, the way we replaced $not\, q$ becomes more complicated when the rules with head q have more than one body literal or head atoms other than q.

Example 2
Consider program $P = \{v \leftarrow not\, q; q \leftarrow s, t; q \vee u \leftarrow w\}$, a variation of the program in Ex. 1. We observe that the following rules for v would desirably be in the result of forgetting q.

$$v \leftarrow not\, s, not\, w \qquad v \leftarrow not\, t, not\, w \qquad v \leftarrow not\, s, not\, not\, u \qquad v \leftarrow not\, t, not\, not\, u$$

These four rules correspond to the four possible ways to guarantee that q is false, i.e., that q is not derived by any of the rules with q in the head. For each rule, this means that either one body literal is false or one head atom different from q is true. The first two rules correspond to the cases where one body atom in each rule with head q is false. The last two rules correspond to the case where u, the other disjunct in the head of the third rule, is true and one of s and t is false. Intuitively, u in the head corresponds to $not\, u$ in the body, which is why $not\, not\, u$ appears.

Besides the rules for v, we should also have the following rules for deriving u.

$$u \leftarrow w, not\, s, not\, not\, u \qquad\qquad u \leftarrow w, not\, t, not\, not\, u$$

These rules are necessary to guarantee that, similarly as in P, u can be either true or false (which is why $not\, not\, u$ appears) whenever w (the body of the original rule with head u) is true.

Together, these six rules guarantee the preservation of the existing implicit dependencies of P.

In (Eiter and Wang 2008; Knorr and Alferes 2014), sets of conjunctions of literals are collected with the aim of replacing some $not\, q$ while preserving its truth value (though not considering disjunction nor double negation (Eiter and Wang 2008)). We now introduce the notion of *as-dual* $\mathcal{D}^{q}_{as}(P)$, as generalized from (Knorr and Alferes 2014), which collects

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

the possible ways to satisfy all rules of P independently of q. The intuitive idea is that, when applied to the set of rules that contain q in the head, the as-dual contains sets of literals, each representing a possible way to guarantee that q is false (since every rule with q in the head is satisfied independently of q).

For this purpose, we need to introduce further notation. For a set S of literals, $not\,(S) = \{not\, s : s \in S\}$ and $not\, not\,(S) = \{not\, not\, s : s \in S\}$, where, for $p \in \Sigma$, we assume the simplification $not\, not\, not\, p = not\, p$ and $not\, not\, not\, not\, p = not\, not\, p$. The sets $B^{\setminus q}(r)$ and $H^{\setminus q}(r)$ respectively denote the set of body and head literals after removing every occurrence of q, i.e., $B^{\setminus q}(r) = B(r) \setminus \{q, not\, q, not\, not\, q\}$ and $H^{\setminus q}(r) = H(r) \setminus \{q\}$. We define $\mathcal{D}^q_{as}(P)$ as follows.

$$\mathcal{D}^q_{as}(P) = \{not\,(\{l_1, \ldots, l_m\}) \cup not\, not\,(\{l_{m+1}, \ldots, l_n\}) : l_i \in B^{\setminus q}(r_i), 1 \leq i \leq m,$$
$$l_j \in H^{\setminus q}(r_j), m+1 \leq j \leq n, \langle \{r_1, \ldots, r_m\}, \{r_{m+1}, \ldots, r_n\} \rangle \text{ is a partition of } P\}$$

The as-dual construction considers the possible partitions of P, and for each partition $\langle F, T \rangle$ collects the negation of exactly one element (except q) from the body of each rule of F, thus guaranteeing that the body of every rule of F is not satisfied, together with the double negation of exactly one head atom (except q) from each rule of T, thus guaranteeing that the head of every rule of T is satisfied, together guaranteeing that all rules in P are satisfied independently of q.

When applied to the set of rules that contain q in the head, the as-dual represents the possible ways to replace the occurrences of $not\, q$ in the body of a rule when forgetting about q from P. It is important to note that, in this case, each set in $\mathcal{D}^q_{as}(P)$ contains one literal for each rule that contains q in the head (those that allow q to be derived), since only in this way we can guarantee that q is false. This definition covers two interesting corner cases. First, if there is no rule with q in its head, i.e., the input program P is empty, then $\mathcal{D}^q_{as}(P) = \{\emptyset\}$, meaning that q is false, and therefore we do not need to impose conditions on other atoms. Finally, if P contains q as a fact, then $\mathcal{D}^q_{as}(P) = \emptyset$, since in this case it is impossible to make q false.

Example 3
Consider the program of Ex. 1. Applying the as-dual to the two rules with q in the head, we obtain $\{\{not\, s, not\, w\}\}$, whose unique element corresponds to the only way to make q false.

If we now consider the program of Ex. 2, which has two rules with q in the head, the as-dual construction renders $\{\{not\, s, not\, not\, u\}, \{not\, t, not\, not\, u\}, \{not\, s, not\, w\}, \{not\, t, not\, w\}\}$, which correspond to the four possible ways to guarantee that q is false.

3.2 A Novel Forgetting Operator

We are now ready to present the formal definition of the operator f_{SP}. As this definition is technically elaborate, we will first present the new operator itself that allows forgetting a single atom from a given program, and subsequently explain and illustrate its definition.

Definition 3 (*Forgetting about q from P*)
Let P be a program over Σ, and $q \in \Sigma$. Let $P' = NF(P)$ be the normal form of P. Consider the following sets, each representing a possible way q can appear in rules of P':

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

$$R := \{r \in P' \mid q \notin \Sigma(r)\} \qquad R_2 := \{r \in P' \mid not\ not\ q \in B(r), q \notin H(r)\}$$
$$R_0 := \{r \in P' \mid q \in B(r)\} \qquad R_3 := \{r \in P' \mid not\ not\ q \in B(r), q \in H(r)\}$$
$$R_1 := \{r \in P' \mid not\ q \in B(r)\} \qquad R_4 := \{r \in P' \mid not\ not\ q \notin B(r), q \in H(r)\}$$

The result of forgetting about q from P, $\mathsf{f}_{SP}(P, q)$, is $NF(P'')$, where P'' is as follows:

- each $r \in R$
- for each $r_0 \in R_0$
  - **1a** for each $r_4 \in R_4$
    $H(r_0) \cup H^{\setminus q}(r_4) \leftarrow B^{\setminus q}(r_0) \cup B(r_4)$
  - **2a** for each $r_3 \in R_3$, $r' \in R_1 \cup R_4$
    $H(r_0) \cup H^{\setminus q}(r_3) \leftarrow B^{\setminus q}(r_0) \cup B^{\setminus q}(r_3) \cup not\,(H^{\setminus q}(r')) \cup not\,not\,(B^{\setminus q}(r'))$
  - **3a** for each $r_3 \in R_3$, $h(r_0) \in H(r_0)$, $D \in \mathcal{D}^q_{as}(R_0 \cup R_2 \setminus \{r_0\})$
    $H(r_0) \leftarrow B^{\setminus q}(r_0) \cup \{not\ not\ h(r_0))\} \cup D \cup B^{\setminus q}(r_3) \cup not\,(H^{\setminus q}(r_3))$
- for each $r_2 \in R_2$
  - **1b** for each $r_4 \in R_4$
    $H(r_2) \leftarrow B^{\setminus q}(r_2) \cup not\,(H^{\setminus q}(r_4)) \cup not\,not\,(B(r_4))$
  - **2b** for each $r_3 \in R_3$, $r' \in R_1 \cup R_4$
    $H(r_2) \leftarrow B^{\setminus q}(r_2) \cup not\,(H^{\setminus q}(r_3) \cup H^{\setminus q}(r')) \cup not\,not\,(B^{\setminus q}(r_3) \cup B^{\setminus q}(r'))$
  - **3b** for each $r_3 \in R_3$, $h(r_2) \in H(r_2)$, $D \in \mathcal{D}^q_{as}(R_0 \cup R_2 \setminus \{r_2\})$
    $H(r_2) \leftarrow B^{\setminus q}(r_2) \cup not\,(H^{\setminus q}(r_3)) \cup not\,not\,(B^{\setminus q}(r_3) \cup \{h(r_2)\}) \cup D$
- for each $r' \in R_1 \cup R_4$
  - **4** for each $D \in \mathcal{D}^q_{as}(R_3 \cup R_4)$ such that $D \cap not\,B^{\setminus q}(r') = \emptyset$
    $H^{\setminus q}(r') \leftarrow B^{\setminus q}(r') \cup D$
  - **5** for each $r_3 \in R_3$, $r \in R_0 \cup R_2$, $D \in \mathcal{D}^q_{as}(R_4)$ such that $D \cap not\,B^{\setminus q}(r') = \emptyset$
    $H^{\setminus q}(r') \leftarrow B^{\setminus q}(r') \cup not\,(H(r) \cup H^{\setminus q}(r_3)) \cup not\,not\,(B^{\setminus q}(r) \cup B^{\setminus q}(r_3)) \cup D$
  - **6** for each $r_3 \in R_3$, $h(r') \in H(r')$, $D \in \mathcal{D}^q_{as}(R_1 \cup R_4 \setminus \{r'\})$
    $H^{\setminus q}(r') \leftarrow B^{\setminus q}(r') \cup not\,(H^{\setminus q}(r_3)) \cup not\,not\,(B^{\setminus q}(r_3) \cup \{h(r')\}) \cup D$
- for each $r_0 \in R_0$
  - **7** for each $r_3, r'_3 \in R_3$ with $r_3 \neq r'_3$, $D \in \mathcal{D}^q_{as}(R_0 \cup R_2 \setminus \{r_0\}), h(r_0) \in H(r_0)$
    $H(r_0) \cup H^{\setminus q}(r_3) \leftarrow B^{\setminus q}(r_0) \cup B^{\setminus q}(r_3) \cup not\,(H^{\setminus q}(r'_3)) \cup not\,not\,(B^{\setminus q}(r'_3) \cup \{h(r_0)\}) \cup D$

Using the normal form P' of P, five sets of rules, R_0, R_1, R_2, R_3, and R_4, are defined over P', in each of which q appears in the rules in a different form. In addition, R contains all rules from P' that do not mention q. These appear unchanged in the final result of forgetting.

The construction is divided in two cases: one for the rules which contain q or $not\,not\,q$ in the body (R_0 or R_2), and one for the rules that contain $not\,q$ in the body or q in the head (R_1 or R_4).

For rules r in $R_0 \cup R_2$, **1a** and **1b** generate the rules obtained by substituting the occurrences of q or $not\,not\,q$ by the body of those rules whose head is q and do not contain q in the body (those in R_4). Also, **2a** and **2b** create the model-generating rules, one for each rule r' supported by $not\,q$ (those in $R_1 \cup R_4$), and **3a** and **3b** create rules of the form $H(r) \leftarrow not\,not\,H(r), B(r)$, one such rule for every possible partition of the bodies of rules in $R_0 \cup R_2$, which can be obtained using the as-dual construction. Each such rule also contains $B^{\setminus q}(r)$, $B^{\setminus q}(r_3)$ and $not\,(H^{\setminus q}(r_3))$ in the body, because the original rule $H(r) \leftarrow q$ is triggered if $B^{\setminus q}(r)$ and $B^{\setminus q}(r_3)$ are true and no other head atom but q in the generating rule r_3 is true.

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

Now let us consider the other case (for rules r_1 in R_1 and r_4 in R_4). Since $\mathit{not}\, q$ appears in the body of r_1, the case **4** is based on the as-dual as discussed in Ex. 2. The idea is that $H(r_1)$ can be concluded if $B^{\setminus q}(r_1)$ is true and no body of a rule with head q (those in R_3 and R_4) is true. Likewise, $H^{\setminus q}(r_4)$ can be concluded if $B(r_4)$ is true and there is no further evidence that q is true, which again can be given by the as-dual construction. The cases **5** and **6** are similar to those of **2a**, **2b** and **3a**, **3b**. If there is cyclic support for q, and each head of $R_1 \cup R_4$ is true, we can justify cyclic support for the rule. For each of these rules that has no true head, we require a false body atom as evidence that the head is not true, because its body is not satisfied. Lastly, if there are multiple self-cycles on the atom that is to be forgotten, the case **7** connects them so that they are handled correctly.

Once all these rules are computed, a final normalization step is applied to remove any tautologies or irrelevant atoms in the resulting rules.

Example 4
Recall P from Ex. 1. The sets R_2 and R_3 are empty and, therefore, the result of $\mathsf{f}_{SP}(P, q)$ is

$$t \leftarrow s \qquad\qquad t \leftarrow w \qquad\qquad v \leftarrow \mathit{not}\, s, \mathit{not}\, w$$

where the first two rules are produced by **1a**, s.t. q in the body of $t \leftarrow q$ is replaced by s and w, resp. The third rule is obtained by **4**, which replaces $\mathit{not}\, q$ in the body of $v \leftarrow \mathit{not}\, q$ by $\mathit{not}\, s, \mathit{not}\, w$ since $\mathcal{D}^q_{as}(R_3 \cup R_4) = \{\{\mathit{not}\, s, \mathit{not}\, w\}\}$. The result corresponds exactly to the one given in Ex. 1.

Example 5
Recall program P of Ex. 2 which also uses disjunction. Since $\mathcal{D}^q_{as}(R_3 \cup R_4) = \{\{\mathit{not}\, s, \mathit{not}\, \mathit{not}\, u\}, \{\mathit{not}\, t, \mathit{not}\, \mathit{not}\, u\}, \{\mathit{not}\, s, \mathit{not}\, w\}, \{\mathit{not}\, t, \mathit{not}\, w\}\}$, **4** replaces q in the head of $q \vee u \leftarrow w$, and $\mathit{not}\, q$ in the body of $v \leftarrow \mathit{not}\, q$ with such elements, yielding

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

the result of $\mathsf{f}_{SP}(P, q)$. Note that by **4**, $\{\mathit{not}\, s, \mathit{not}\, w\}$ and $\{\mathit{not}\, t, \mathit{not}\, w\}$ are not used with $q \vee u \leftarrow w$.

In the previous examples, no cyclic dependencies on the atom to be forgotten existed. Next, we discuss two examples that deal with this more sophisticated case.

Example 6
Consider forgetting about q from the following program $P = \{q \leftarrow \mathit{not}\, \mathit{not}\, q; a \leftarrow q\}$, an example which cannot be handled by any syntactic forgetting operator in the literature. In this case, since R_1, R_2 and R_4 are empty, the only applicable rule is **3a**. Since there is a cyclic dependency on q, i.e., $R_3 \neq \emptyset$, the application of **3a** renders the result of $\mathsf{f}_{SP}(P, q)$ to be:

$$a \leftarrow \mathit{not}\, \mathit{not}\, a$$

Thus, when forgetting about q from P, the cyclic dependency on q is transferred to a.

If there are several dependencies on q though, such self-cycles also create an implicit dependency between the elements supported by q (the heads of rules in $R_0 \cup R_2$) and those

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

supported by $not\, q$ (the heads of rules in $R_1 \cup R_4$). When forgetting q, such dependencies must be taken into account.

Example 7
Now, consider program $P = \{q \leftarrow not\, not\, q; u \leftarrow q; s \leftarrow q; t \leftarrow not\, q\}$. The answer sets $\{q, u, s\}$ and $\{t\}$ of P should be preserved (for all but q) when forgetting about q, even if we add to P a set of rules not containing q. By distinguishing which rule heads depend on q (those of $R_0 \cup R_2$) and which on $not\, q$ (those of $R_1 \cup R_4$), **2a** and **5** create the following model-generating rules, which guarantee that the atoms that depend on q should be true whenever the atoms that depend on $not\, q$ are false, and vice versa:

$$u \leftarrow not\, t \qquad s \leftarrow not\, t \qquad t \leftarrow not\, u \qquad t \leftarrow not\, s$$

These rules alone have the two desired answer sets $\{u, s\}$ and $\{t\}$, since they basically state that either the elements supported by q are true, or the elements supported by $not\, q$ are, but not both. In addition, **3a** and **6** add the following rules:

$$u \leftarrow not\, not\, u, not\, not\, s \qquad s \leftarrow not\, not\, s, not\, not\, u \qquad t \leftarrow not\, not\, t$$

The first two rules guarantee that, whenever t is derivable (independently, e.g., by the existence of $t \leftarrow$), then u and s may both either be simultaneously true or false. Similarly, the third rule ensures that t may still vary between true and false if we, e.g., add both $u \leftarrow$ and $s \leftarrow$.

3.3 Properties

In this section, we present several properties that our syntactic operator f_{SP} satisfies.

First of all, we show that f_{SP} is in fact a forgetting operator, i.e., $\mathsf{f}_{SP}(P, q)$ does not contain q.

Proposition 3
Let P be a program over Σ and $q \in \Sigma$. Then $\mathsf{f}_{SP}(P, q)$ is a program over $\Sigma \backslash \{q\}$.

In order to show that f_{SP} satisfies **(SP)** whenever this is possible, we start by showing that f_{SP} semantically coincides with any operator of the class F_{SP} (cf. Def. 4 of (Gonçalves et al. 2016b)).

Theorem 1
Let P be a program over Σ and $q \in \Sigma$. For any $\mathsf{f} \in F_{SP}$, we have $\mathcal{HT}(\mathsf{f}_{SP}(P, q)) = \mathcal{HT}(\mathsf{f}(P, \{q\}))$.

Since, as mentioned in Sec. 2, F_{SP} was shown to correspond to the class of forgetting operators that satisfy **(SP)** for all those cases when this is possible (cf. Thm. 4 of (Gonçalves et al. 2016b)), Thm. 1 allows us to conclude the main result of the paper, i.e., that f_{SP} satisfies **(SP)** for all those cases when this is possible. In fact, it is the first such syntactic forgetting operator.

Theorem 2
Let P be a program over Σ and $q \in \Sigma$. If $\langle P, \{q\} \rangle$ does not satisfy Ω, then f_{SP} satisfies $\mathbf{(SP)}_{\langle P, \{q\} \rangle}$.

This result guarantees that f_{SP} provides the ideal result whenever Ω is not true.

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

When Ω is true, such as in Ex. 7, although we are not in an ideal situation, it might be the case that we must forget (Gonçalves et al. 2017). In this case, we can use f_{SP}, as it is defined for every extended program, while still satisfying several desirable properties.

Property **(SI)** guarantees that all rules of a program P not mentioning q be (semantically) preserved when forgetting about q from P. Notably, although **(SI)** is a desirable property, several classes of forgetting operators in the literature fail to satisfy it. Interestingly, f_{SP} preserves all rules of P not mentioning q, thus satisfying a strong version of **(SI)**.

Proposition 4
Let P be a program over Σ and $q \in \Sigma$. Then, $\mathsf{f}_{SP}(P \cup R, q) = \mathsf{f}_{SP}(P, q) \cup R$, for all programs R over $\Sigma \backslash \{q\}$.

An important property that f_{SP} satisfies in general is the preservation of all answer sets of P (modulo q), even in the presence of an additional program not containing q.

Theorem 3
Let P be a program over Σ and $q \in \Sigma$. Then, $\mathcal{AS}(P \cup R)_{\|\{q\}} \subseteq \mathcal{AS}(\mathsf{f}_{SP}(P, q) \cup R)$, for all programs R over $\Sigma \backslash \{q\}$.

Obviously, whenever f_{SP} satisfies $\mathbf{(SP)}_{\langle P, \{q\} \rangle}$, then we obtain equality of the two sets of answer sets in the previous result. However, when Ω is satisfied, then we may obtain a strict superset. Take Ex. 7, where Ω is satisfied. We can observe that the provided result does admit a third answer set $\{s, t, u\}$ besides the two P itself has. But, no matter which program R we add, existing answer sets are preserved. In other words, even if Ω is satisfied, the construction of f_{SP} ensures that no existing answer sets (modulo the forgotten atom) are lost while forgetting.

The operator f_{SP} also preserves strong equivalence, i.e., forgetting the same atom from two strongly equivalent programs yields strongly equivalent results.

Proposition 5
Let P and P' be programs over Σ and $q \in \Sigma$. If $P \equiv P'$ then $\mathsf{f}_{SP}(P, q) \equiv \mathsf{f}_{SP}(P', q)$.

These positive results suggest that, in cases when we *must* forget, f_{SP} can be used without first checking Ω, in line with the observations on the usage of F_{SP} in (Gonçalves et al. 2017).

Still, we may want to find a broad class of instances $\langle P, \{q\} \rangle$ for which we can guarantee that $\mathbf{(SP)}_{\langle P, \{q\} \rangle}$ is satisfied without having to check criterion Ω. We have seen in Ex. 7 that self-cycles on the atom to be forgotten are relevant for forgetting while satisfying **(SP)**, a fact already observed in (Knorr and Alferes 2014). We now extend the notion of q-forgettable given in (Knorr and Alferes 2014) to programs with disjunction in the head of the rules.

Definition 4
Let P be a program in normal form over Σ and $q \in \Sigma$. Then, we say that P is *q-forgettable* if at least one of the following conditions holds:

- all occurrences of q in P are within self-cycles, rules with $q \in H(r)$ and $q \in B^{--}(r)$
- P contains the fact $q \leftarrow$
- P contains no self-cycle on q, a rule with $q \in H(r)$ and $q \in B^{--}(r)$

We can show that, when restricted to q-forgettable programs, Ω is not satisfied.

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

Theorem 4
Let P be a program over Σ, and $q \in \Sigma$. If P is q-forgettable, then $\langle P, \{q\}\rangle$ does not satisfy Ω.

An immediate consequence of this theorem is that, for q-forgettable programs, f_{SP} satisfies $\mathbf{(SP)}_{\langle P,\{q\}\rangle}$. In the case of the programs of Ex. 1 and Ex. 2, which are q-forgettable, we can forget q from both programs while satisfying $\mathbf{(SP)}_{\langle P,\{q\}\rangle}$, and we can use f_{SP} to obtain the desired result.

The converse of Thm. 4 does not hold in general. Program P of Ex. 6 is a simple counter-example, since it is not q-forgettable, and $\langle P, \{q\}\rangle$ does not satisfy Ω. This is not surprising given that deciding Ω is Σ_3^P-complete (cf. Thm. 7 of (Gonçalves et al. 2017)), whereas deciding if a program is q-forgettable requires to check each rule once.

Proposition 6
Let P be a program over Σ, and $q \in \Sigma$. Deciding if P is q-forgettable can be done in linear time.

Besides the simplicity of checking if a program is q-forgettable, the restriction to these programs implies that f_{SP} can be constructed using only a small subset of the derivation rules.

Theorem 5
Let P be a program over Σ, and $q \in \Sigma$. If P is q-forgettable, then $\mathsf{f}_{SP}(P, q)$ is constructed using only the derivation rules **1a**, **1b** and **4**.

We can now present the complexity result of our forgetting operator f_{SP}.

Theorem 6
Let P be a program over Σ and $q \in \Sigma$. Then, computing $\mathsf{f}_{SP}(P, q)$ is in EXPTIME in the number of rules containing occurrences of q and linear in the remaining rules.

This result is not surprising taking into account the arguments in (Eiter and Kern-Isberner 2019) showing that the result of forgetting is necessarily exponential. Nevertheless, in the case of f_{SP}, this exponential behavior is limited to the rules mentioning the atom to be forgotten.

Besides the properties already mentioned, a fundamental characteristic of the operator f_{SP} is its syntactic nature, in the sense that the result of forgetting is obtained by a manipulation of the rules of the original program. As a consequence, the result of forgetting according to f_{SP} is somehow close to the original program. In order to formalize this idea, we define a distance function between programs. It builds on a distance measure between rules, which counts the number of literals only appearing in one of the rules.

Definition 5
Let r and r' be two rules over Σ. The distance between r and r' is $d(r, r') = |H(r) \ominus H(r')| + |B(r) \ominus B(r')|$ where $A \ominus B = (A \backslash B) \cup (B \backslash A)$ is the usual symmetric difference. The size of a rule r is defined as $|r| = |H(r)| + |B(r)|$.

To extend this distance to a distance between programs, we use mappings between programs.

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

Definition 6
Let P_1 and P_2 be programs over Σ. The distance between P_1 and P_2 is $dist(P_1, P_2) = Min\{dist_m(P_1, P_2) : m$ is a mapping between P_1 and $P_2\}$, where a mapping between P_1 and P_2 is a partial injective function $m : P_1 \to P_2$ and $dist_m(P_1, P_2) = Sum\{d(r, m(r)) : m(r) \in P_2\} + Sum\{|r| : r \in (P_1 \backslash m^{-1}[P_2])\} + Sum\{|r| : r \in (P_2 \backslash m[P_1])\}$.

The distance between P_1 and P_2 induced by a mapping m is the sum of the distances of those rules associated by m, and the sizes of the remaining rules. The distance between P_1 and P_2 is then the minimal value for the possible mappings of P_1 and P_2. Intuitively, this distance corresponds to the minimal number of literals we need to add to both programs to make them equal, noting that we may need to add new rules.

Example 8
Let $P_1 = \{a \leftarrow b, not\, c\}$ and $P_2 = \{a \leftarrow not\, c,\ b \leftarrow d\}$ be two programs. Then, $dist(P_1, P_2)$=3, which corresponds to the sum of the distance between the first rule of P_2 and the rule of P_1, which is 1, with the size of $b \leftarrow d$, which is 2. Intuitively, this corresponds to adding b to the body of the first rule of P_2 and adding the entire rule $b \leftarrow d$ to P_1. We could consider adding P_1 to P_2 and P_2 to P_1, corresponding to the mapping that is undefined for each rule of P_1, but this would induce a distance of 7, the sum of the size of every rule in the programs. In fact, this is always an upper bound to the distance between two programs.

Let us now consider the distance function in the case of forgetting. We compare our syntactic operator f_{SP} with a semantic operator $\mathsf{f}_{Sem} \in F_{SP}$, which is defined using the counter-model construction (based on (Wang et al. 2013; Wang et al. 2014).

First, we prove a result regarding the rules generated by the semantic operator.

Proposition 7
Let P be a program over Σ and $q \in \Sigma$. Then, for each $r \in \mathsf{f}_{Sem}(P, q)$, we have that $|r| \geq |\Sigma|$.

The following result presents an upper bound for the syntactic operator and a lower bound for the semantic one.

Proposition 8
Let P be a program over Σ and $q \in \Sigma$. Then,

- $dist(P, \mathsf{f}_{SP}(P, q)) \leq (|\mathsf{f}_{Sem}(P, q)| + |P|) \times 2|\Sigma|$;
- $dist(P, \mathsf{f}_{Sem}(P, q)) \geq (|\mathsf{f}_{Sem}(P, q)| - |P|) \times |\Sigma|$.

The upper and lower bounds of the previous result depend heavily on the size of the result of forgetting for each operator. In general the result of forgetting using the semantic operator has many more rules than the result using the syntactic one.

Proposition 9
Let P be a program over Σ and $q \in \Sigma$. For each rule $r \in \mathsf{f}_{SP}(P, q)$ there are at least 2^D rules in $\mathsf{f}_{Sem}(P, q)$, with $D = Min(|H(r)|, |\Sigma \backslash \Sigma(r)|)$.

Example 9
For $P = \{q \leftarrow s; q \vee u \leftarrow r; t \leftarrow q; v \leftarrow not\, q\}$, we obtain $\mathsf{f}_{SP}(P, q)$ as follows:

$$t \leftarrow s \qquad t \vee u \leftarrow r \qquad u \leftarrow r, not\, s, not\, not\, u$$
$$v \leftarrow not\, s, not\, not\, u \qquad v \leftarrow not\, s, not\, r$$

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

The distance between $\mathsf{f}_{SP}(P, q)$ and P is $dist(P, \mathsf{f}_{SP}(P, q)) = 16$. The result of the forgetting according to f_{Sem} has 73 rules. The distance to the original program P is $dist(P, \mathsf{f}_{Sem}(P, q)) = 486$, more than thirty times the distance in the case of f_{SP}.

The result of forgetting according to f_{SP} is clearly closer to the original program than that obtained by the f_{Sem} using the counter-models construction of (Wang et al. 2013; Wang et al. 2014). Although this construction improves on that of (Cabalar and Ferraris 2007) by doing better than just giving one rule for each counter-model, we could have considered the construction in (Cabalar et al. 2007), which is also based on counter-models, but improves on the former by guaranteeing a minimal program as a result. However, the extra complexity to obtain a minimal program as result is not quantified (Cabalar et al. 2007). More importantly, being also a semantic construction, the resulting program will most likely not resemble the original program at all.

4 Related Work

Related work can be divided into two groups: one of early approaches that do not consider the notion of strong equivalence and the related HT-models for their definition, and those that do.

The first approach, due to Zhang and Foo (2006), belongs to the former group and consists of a pair of purely syntactic operators, strong and weak forgetting, defined for normal logic programs. As argued by Eiter and Wang (2008), both lack the semantic underpinnings and are syntax-sensitive, i.e., even answer sets are not preserved while forgetting, and this necessarily generalizes to preserving HT-models (modulo the forgotten atoms). Thus **(SP)** cannot be preserved even in very simple cases. For example, strong forgetting about q from $P = \{p \leftarrow not\, q;\ p \leftarrow not\, p\}$ results in $\{p \leftarrow not\, p\}$, an inconsistent program, and weak forgetting about q from $P = \{q \leftarrow;\ p \leftarrow not\, q\}$ results in $\{p \leftarrow\}$, a counter-intuitive result (Eiter and Wang 2008).

Semantic Forgetting (Eiter and Wang 2008) aims at preserving the answer sets while forgetting to overcome the problems of these two operators. It is defined for disjunctive programs, but, among the three forgetting operators presented, only forget_3 is syntactic (the other two create canonical representations based on the computed answer sets). While being preferable over strong and weak forgetting for its semantic foundation, it is not sufficient since it focuses on *equivalence* instead of *strong equivalence*. Hence, even for simple examples, **(SP)** cannot be satisfied. Consider forgetting about q from $P' = \{p \leftarrow q;\ q \leftarrow not\, c\}$: we would expect the result to be $\{p \leftarrow not\, c\}$, since we want to preserve the dependency between c and p, however, forget_3 returns $\{p \leftarrow\}$. Both programs have the same answer sets – hence are *equivalent* – but do not have the same logical meaning, witnessed by the fact that they are not *strong equivalent*. In addition, while the operator is defined for disjunctive programs, its output may contain double negation, which means that the operator cannot be iterated. To partially circumvent this, a restriction on programs is introduced under which it can be iterated (Eiter and Wang 2008). Yet, this restriction excludes very basic examples such as forgetting about q from $P = \{c \leftarrow not\, p; p \leftarrow not\, q; q \leftarrow not\, p\}$. Moreover, an initial transformation is applied in forget_3 to obtain a negative normal form, i.e., a program without positive atoms in rule bodies. While facilitating the presentation of the algorithm, it considerably

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

reduces the similarity between the input program and its forgetting result, unnecessarily increasing the number of rules in the resulting program, even affecting rules that do not contain the atom(s) to be forgotten.

The only syntactic HT-models based operator is strong as-forgetting (Knorr and Alferes 2014) – all other HT-models based approaches in the literature are limited to a semantic characterization of the result, not presenting a concrete forgetting operator, or only one which is based on the notion of counter-models in HT (Cabalar and Ferraris 2007), with the drawbacks discussed in Sec. 1. While strong as-forgetting provides the expected results in certain cases, such as P' above, its applicability is severely limited to certain programs within a non-standard class with double negation, but without disjunction. Additionally, its result often does not belong to the class of programs for which it is defined, preventing its iterative use. f_{SP} overcomes all the shortcomings of strong as-forgetting and satisfies **(SP)** whenever this is possible.

5 Conclusions

We proposed a concrete forgetting operator for the class of extended logic programs which satisfies several important properties, notably **(SP)** whenever this is possible. Equally important is the fact that the operator is defined in a syntactic way, thus not requiring the calculation of models, and producing a program that is somehow close to the original program, hence keeping its declarative character. In particular, the rules that do not mention the atom to be forgotten remain intact. Furthermore, while the remaining rules may grow exponentially – which is unsurprising – even if only on the number of rules that mention the atom being forgotten, often the resulting program is the closest to the initial program, as would be the case in the example described in Sect. 1. Overall, this is a substantial improvement over the existing operators, which either did not obey most desirable properties (Zhang and Foo 2006; Eiter and Wang 2008), or were only defined for very restricted classes of programs (Knorr and Alferes 2014), or were based on generic semantic methods that required the computation of models and produced large programs that were syntactically very different from the original ones. Furthermore, we characterised a class of programs – *q-forgettable* – whose membership can be checked in linear time, and for which we can guarantee **(SP)**. For cases where **(SP)** cannot be ensured, our operator can nevertheless still be employed with desirable properties, such as guaranteeing that all answer sets are preserved, along the lines of one of the alternatives semantically investigated in (Gonçalves et al. 2017). In other words, our operator can be used without having to perform the computationally expensive semantic check (Ω) for *strong persistence*, ensuring the “*perfect*” result whenever **(SP)** is possible, and a “*good*” result when “*perfection*” is not possible.

Future work includes dealing with (first-order) variables, and following the other alternatives proposed in (Gonçalves et al. 2017) to deal with cases where it is not possible to guarantee **(SP)**.

Acknowledgments

M. Berthold was partially supported by the International MSc Program in Computational Logic (MCL). R. Gonçalves, M. Knorr, and J. Leite were partially sup-

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

ported by FCT projects FORGET (PTDC/CCI-INF/32219/2017) and NOVA LINCS (UID/CEC/04516/2019).

References

Alferes, J. J., Knorr, M., and Wang, K. 2013. Forgetting under the well-founded semantics. In *Procs. of LPNMR*. LNCS, vol. 8148. Springer, 36–41.

Berthold, M., Gonçalves, R., Knorr, M., and Leite, J. 2019. A syntactic operator for forgetting that satisfies strong persistence. *CoRR abs/1907.12501*.

Brass, S. and Dix, J. 1999. Semantics of (disjunctive) logic programs based on partial evaluation. *J. Log. Program. 40,* 1, 1–46.

Cabalar, P. and Ferraris, P. 2007. Propositional theories are strongly equivalent to logic programs. *TPLP 7,* 6, 745–759.

Cabalar, P., Pearce, D., and Valverde, A. 2007. Minimal logic programs. In *Procs. of ICLP*. Springer.

Delgrande, J. P. and Wang, K. 2015. A syntax-independent approach to forgetting in disjunctive logic programs. In *Procs. of AAAI.* AAAI Press, 1482–1488.

Eiter, T. and Kern-Isberner, G. 2019. A brief survey on forgetting from a knowledge representation and reasoning perspective. *KI - Künstliche Intelligenz 33,* 1, 9–33.

Eiter, T. and Wang, K. 2008. Semantic forgetting in answer set programming. *Artif. Intell. 172,* 14, 1644–1672.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Comput. 9,* 3-4, 365–385.

Gonçalves, R., Janhunen, T., Knorr, M., Leite, J., and Woltran, S. 2019. Forgetting in modular answer set programming. In *Procs. of AAAI.* AAAI Press.

Gonçalves, R., Knorr, M., and Leite, J. 2016a. The ultimate guide to forgetting in answer set programming. In *Procs. of KR.* AAAI Press, 135–144.

Gonçalves, R., Knorr, M., and Leite, J. 2016b. You can't always forget what you want: on the limits of forgetting in answer set programming. In *Procs. of ECAI.* IOS Press, 957–965.

Gonçalves, R., Knorr, M., and Leite, J. 2017. Iterative variable elimination in ASP. In *Procs. of EPIA*. LNCS, vol. 10423. Springer, 643–656.

Gonçalves, R., Knorr, M., Leite, J., and Woltran, S. 2017. When you must forget: Beyond strong persistence when forgetting in answer set programming. *TPLP 17,* 5-6, 837–854.

Inoue, K. and Sakama, C. 1998. Negation as failure in the head. *J. Log. Program. 35,* 1, 39–78.

Inoue, K. and Sakama, C. 2004. Equivalence of logic programs under updates. In *Procs. of JELIA*. Springer.

Knorr, M. and Alferes, J. J. 2014. Preserving strong equivalence while forgetting. In *Procs. of JELIA*. LNCS, vol. 8761. Springer, 412–425.

Kontchakov, R., Wolter, F., and Zakharyaschev, M. 2010. Logic-based ontology comparison and module extraction, with an application to dl-lite. *Artif. Intell. 174,* 15, 1093–1141.

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

Lang, J. and Marquis, P. 2010. Reasoning under inconsistency: A forgetting-based approach. *Artif. Intell. 174,* 12-13, 799–823.

Leite, J. 2017. A bird's-eye view of forgetting in answer-set programming. In *Procs. of LPNMR.* LNCS, vol. 10377. Springer, 10–22.

Lifschitz, V., Pearce, D., and Valverde, A. 2001. Strongly equivalent logic programs. *ACM Trans. Comput. Log. 2,* 4, 526–541.

Lin, F. and Reiter, R. 1997. How to progress a database. *Artif. Intell. 92,* 1-2, 131–167.

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346

Liu, Y. and Wen, X. 2011. On the progression of knowledge in the situation calculus. In *Procs. of IJCAI*. IJCAI/AAAI, 976–982.

Rajaratnam, D., Levesque, H. J., Pagnucco, M., and Thielscher, M. 2014. Forgetting in action. In *Procs. of KR*. AAAI Press.

Slota, M. and Leite, J. 2011. Back and forth between rules and se-models. In *Procs. of LPNMR*. Springer.

Wang, Y., Wang, K., and Zhang, M. 2013. Forgetting for answer set programs revisited. In *Procs. of IJCAI*. IJCAI/AAAI, 1162–1168.

Wang, Y., Zhang, Y., Zhou, Y., and Zhang, M. 2014. Knowledge forgetting in answer set programming. *J. Artif. Intell. Res. (JAIR) 50*, 31–70.

Wang, Z., Wang, K., Topor, R. W., and Pan, J. Z. 2010. Forgetting for knowledge bases in DL-Lite. *Ann. Math. Artif. Intell. 58,* 1-2, 117–151.

Zhang, Y. and Foo, N. Y. 2006. Solving logic program conflict through strong and weak forgettings. *Artif. Intell. 170,* 8-9, 739–778.

Zhang, Y. and Zhou, Y. 2009. Knowledge forgetting: Properties and applications. *Artif. Intell. 173,* 16-17, 1525–1537.

Downloaded from https://www.cambridge.org/core. IP address: 3.226.254.115, on 18 Nov 2019 at 01:29:46, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S1471068419000346