Forgetting in Answer Set Programming with Anonymous Cycles

Matti Berthold[1,2], Ricardo Gonçalves[3], Matthias Knorr[3], and João Leite[3]

[1] Technische Universität Dresden, Germany
[2] Institute of Computer Science, Leipzig University, Germany
[3] NOVA LINCS, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal

Abstract. It is now widely accepted that the operation of forgetting in the context of Answer Set Programming [10,18] is best characterized by the so-called *strong persistence*, a property that requires that all existing relations between the atoms not to be forgotten be preserved. However, it has been shown that strong persistence cannot always be satisfied. What happens if we must nevertheless forget? One possibility that has been explored before is to consider weaker versions of strong persistence, although not without a cost: some relations between the atoms not to be forgotten are broken in the process. A different alternative is to enhance the logical language so that all such relations can be maintained after the forgetting operation. In this paper, we borrow from the recently introduced notion of *fork* [1] – a conservative extension of Equilibrium Logic and its monotonic basis, the logic of Here-and-There – which has been shown to be sufficient to overcome the problems related to satisfying strong persistence. We map this notion into the language of logic programs, enhancing it with so-called anonymous cycles, and we introduce a concrete syntactical forgetting operator over this enhanced language that we show to always obey *strong persistence*.

1 Introduction

There has been a substantial interest in investigating the operation of *forgetting* in the context of Answer Set Programming (c.f. [10] for a recent survey). Intuitively, when we forget some atoms from a logic program, the goal is to come up with another program, written in a language that does not include the atoms to be forgotten, which preserves the meaning with respect to the remaining atoms.

Whereas different approaches over the years, e.g., [6,23,5,24,9,12,7,8], proposed different ways to semantically characterize this operation, it is now rather well accepted that *strong persistence* [16] best captures its essence. *Strong persistence* is a property that requires that all existing relations between the atoms not to be forgotten be preserved during the forgetting operation. However, it has been shown that strong persistence cannot always be satisfied [11]. There are cases where the atoms to be forgotten play such a pivotal role in the original program that one cannot represent its effects on the remaining atoms without them. These cases are usually associated with atoms involved in the so-called *even cycles through negation* (somehow equivalent to *choice rules*) that generate different stable models (or answer sets).

What if we are faced with a situation in which we must forget – e.g., because of a court order, or any other strong reason – but we cannot while obeying *strong persistence*? This problem was first tackled in [13], where the authors investigated different ways to weaken *strong persistence*. However, that does not come without a cost: some relations between the atoms not to be forgotten are broken in the process. This may result, for example, in the unwanted disappearance of existing stable models, or the appearance of new ones. What if we cannot afford to loose any relations between the non-forgotten atoms, but must proceed with the forgetting operation? One alternative is to enhance the logical language so that all such relations can be maintained after the forgetting operation. Recently, in [1], the authors introduced the notion of *fork* – a conservative extension of Equilibrium Logic and its monotonic basis, the logic of Here-and-There – which, in a nutshell, allows for the specification of formulas whose stable models are the union of the stable models of separate formulas. They also proved that *forks* are sufficient to overcome the problems related to satisfying strong persistence.

Inspired by the concept of *forks*, we begin this paper by enhancing the language of logic programs with so-called *anonymous cycles*, aiming at being able to specify concrete forgetting operators that satisfy strong persistence. Extending the language of logic programs with *anonymous cycles* essentially amounts to extending the alphabet with a set of anonymous atoms that can *only* be used in *anonymous cycles*, i.e., to generate and condition alternatives, which are then ignored when the stable models are considered. We then introduce a concrete forgetting operator over this enhanced language, which we show to obey *strong persistence*.

The new operator is syntactical in nature, i.e., it achieves the result by syntactically manipulating the rules of the input program – there is no need to compute any models – and it is the first concrete forgetting operator that obeys *strong persistence*, albeit for this enhanced language. One might argue, somehow cynically, that if we allow ourselves to extend the language and use new atoms, then why not simply keep the ones we were supposed to forget, perhaps renaming them to appear as if they are not the same. Even if this renaming would obey *strong persistence*, modulo these new atoms, it could hardly be classified as *forgetting*. By ensuring that all the non-forgotten atoms introduced by our forgetting operator are anonymous, thus used in a very constrained and fixed way, we ensure a clear syntactical distinction between these and the forgotten ones. Equally important is the fact that the operator is closed under the language of logic programs with anonymous cycles. This allows its iteration, admitting for any number of atoms to be forgotten from a program in any sequence, while still obeying *strong persistence*.

2 Preliminaries

In this section, we recall necessary notions on answer set programming. We assume a *propositional signature* Σ. A *logic program* P over Σ is a finite set of *rules* of the form

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

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

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

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. Given a rule r, $H(r)$ is called the *head* of r, and $B(r) = B^+(r) \cup not\, B^-(r) \cup not\, not\, B^{--}(r)$ is called the *body* of r, where, for a set A of atoms, $not\, A = \{not\, q\colon\ q \in A\}$ and $not\, not\, A = \{not\, not\, q\colon q \in A\}$. 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. 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 $P^I = \{H(r) \leftarrow B^+(r) : r \in P$ such that $B^-(r) \cap I = \emptyset$ 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 $\mathcal{AS}(P)$. Two programs P_1, P_2 are *equivalent* if $\mathcal{AS}(P_1) = \mathcal{AS}(P_2)$ and *strongly equivalent*, $P_1 \equiv P_2$, 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 $\mathcal{HT}(P_1) = \mathcal{HT}(P_2)$ [19]. 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 $\{X \backslash V \mid X \in \mathcal{M}\}$ ($\{\langle X \backslash V, Y \backslash V\rangle \mid \langle X, Y\rangle \in \mathcal{M}\}$).

3 Forgetting with Anonymous Cycles

Forgetting in answer set programming (ASP) aims at eliminating certain elements from the language Σ, without affecting the consequences inferable for the language elements that remain. However, it is not always possible to forget in ASP [11], intuitively, because some elements of the language are crucial to preserve certain dependencies between atoms in the program. A way to avoid this problem would thus be to not remove such atoms entirely, but rather preserve them in a localized form, such that later, under certain circumstances, they could potentially be removed, e.g., after all related atoms have been forgotten. This has been tackled by proposing an extension of ASP with a new construct, called forks [1]. Here, we map this notion into the language of logic programs, but instead of extending the language with the fork constructor, we enhance the language with a set of distinguished atoms, called anonymous atoms, that can only be used in a very restricted way, namely to generate so-called anonymous cycles.

Thus, in this section, we first introduce programs with anonymous cycles and, then, we reconcile the notions of forgetting in ASP with this extension of programs. We start by extending the signature to allow for an infinite number of anonymous atoms.

Definition 1. *An* anonymous signature *is a pair $\langle \Sigma, \Sigma_{an}\rangle$ where Σ is a signature and Σ_{an} is an infinite set of atoms such that $\Sigma \cap \Sigma_{an} = \emptyset$.*

We use Roman letters to denote elements of Σ and Greek letters to denote elements of Σ_{an}. In what follows, we assume a fixed anonymous signature $\langle \Sigma, \Sigma_{an}\rangle$. Based on this, we now define the class of programs with anonymous cycles.

Definition 2. *A* program with anonymous cycles *(over $\langle \Sigma, \Sigma_{an}\rangle$) is a program P such that, for each $\delta \in \Sigma_{an}$ and $r \in P$, exactly one of the following conditions is true:*

– δ *does not appear in* r*;*
– δ *belongs only to* $B^+(r)$*;*
– δ *belongs only to* $B^-(r)$*;*
– r *is of the form* $\delta \leftarrow not\ not\ \delta$*.*

We term a rule r *of the latter form an* anonymous cycle*.*

Therefore, in a program with anonymous cycles, the atoms of Σ_{an} can only appear in a very restricted way. Namely, they can only be used to generate cycles, in anonymous cycles,[4] and, in addition, appear in the positive or negative body of rules. This means that for an anonymous atom $\delta \in \Sigma_{an}$, the rule $\delta \leftarrow not\ not\ \delta$ is the only rule in which δ is allowed to appear in the head or in the double negated body.

Since the atoms of Σ_{an} are to be seen as auxiliary, the semantics of programs with anonymous cycles over an anonymous signature $\langle \Sigma, \Sigma_{an} \rangle$ is defined as the restriction to Σ of the semantics of the program when considered as program over $\Sigma \cup \Sigma_{an}$.

Definition 3. *Let* P *be a program with anonymous cycles over* $\langle \Sigma, \Sigma_{an} \rangle$*, and* P^* *the program over* $\Sigma \cup \Sigma_{an}$ *with the same rules as* P*. Then, the set of HT-models of* P *and the set of answer sets of* P *are defined, respectively, as*

$$\mathcal{HT}(P) = \mathcal{HT}(P^*)_{\|\Sigma_{an}} \qquad \mathcal{AS}(P) = \mathcal{AS}(P^*)_{\|\Sigma_{an}}.$$

Example 1. Consider the program with anonymous cycles P over $\langle \Sigma, \Sigma_{an} \rangle$:

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

where $a, b, c \in \Sigma$ and $\delta \in \Sigma_{an}$. If we consider program P^* over $\Sigma \cup \Sigma_{an}$ with precisely the same rules as P, we have that $\mathcal{AS}(P^*) = \{\{\delta\}, \{c\}\}$ and $\mathcal{AS}(P) = \{\emptyset, \{c\}\}$.

We are now ready to extend notions of forgetting in ASP from the literature to the class of programs with anonymous cycles.

A *forgetting operator* over a class $\mathcal{C}$ of programs with anonymous cycles[5] over $\langle \Sigma, \Sigma_{an} \rangle$ is a partial function $\mathsf{f} : \mathcal{C} \times 2^{\Sigma} \to \mathcal{C}$ s.t. the *result of forgetting about* V *from* P, denoted as $\mathsf{f}(P, V)$, is a program with anonymous cycles over $\langle \Sigma(P) \backslash V, \Sigma_{an} \rangle$, 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. $\mathcal{C}(\mathsf{f}) \subseteq \mathcal{C}$, commonly satisfying some definition of the class.

The notions in the remainder of the section are indeed very similar to the ones introduced for programs (without anonymous cycles). They can essentially be re-used here due to our definition of HT-models and answer sets for programs with anonymous cycles, and because we never forget anonymous atoms.

Arguably, among the many properties introduced for different classes of forgetting operators in ASP [10], *strong persistence* [16] 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.

[4] Note that this term is due to the fact that the answer sets of such an anonymous cycle are precisely $\{\delta\}$ and $\{\}$.

[5] 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.

(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)** [10], **(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 $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}(\mathsf{f})$ and $V \subseteq \Sigma$, we have $\mathsf{f}(P,V) \cup R \equiv \mathsf{f}(P \cup R, V)$ for all programs $R \in \mathcal{C}(\mathsf{f})$ with $\Sigma(R) \subseteq \Sigma \backslash V$.

Although **(SP)** is the central property one wants to ensure to hold when forgetting atoms from an answer set program, it was shown in [11] that this is not always possible, that is, 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. Moreover, a sound and complete criterion, Ω, was presented to characterize when exactly it is not possible to forget while satisfying **(SP)**. In addition, a corresponding class of forgetting operators, F_{SP}, was introduced. It was shown that every operator in F_{SP} satisfies **(SP)** for instances $\langle P, V \rangle$ that do not satisfy Ω, i.e., those instances for which it is possible to forget V from P while satisfying **(SP)**. This makes F_{SP} the ideal choice whenever forgetting is possible. Nevertheless, F_{SP} has two main problems: first, it is only defined semantically, i.e., it only specifies the HT-models that a result of forgetting a set of atoms V from program P should have; and second, for instances $\langle P, V \rangle$ that satisfy Ω, i.e., those instances for which we know that it is not possible to forget V from P while satisfying **(SP)**, the result $\mathsf{f}(P,V)$ necessarily does not have a strong connection with P as imposed by **(SP)**.

4 A Syntactic Operator

Our main result of this paper is that the impossibility result for forgetting in ASP can be overcome at the cost of introducing anonymous cycles (whenever necessary). Moreover, we do it in a syntactic way, by only manipulating the rules of the input program. Note again that this approach does not coincide with, e.g., renaming some atom, as in the end only one rule with the new anonymous atom in the head exists.

Thus, in this section, we introduce the operator f_{AC} which, by syntactical manipulation of the input, removes an atom from a program with anonymous cycles. As this operator produces in the worst case a program with anonymous cycles, we will then extend it in a straightforward way to forget any number of atoms iteratively and show that the order of doing so in fact has no effect on the correctness of the result.

To simplify the presentation and the cases that are considered in the construction, and also to reduce the size of the input, we reduce programs to a normal form, similar to [16] and previous related work [14,15,4,20]. There are two essential differences to the normal form considered in [16]. First of all, contrarily to [16], our normal form applies to programs with disjunctive heads. Moreover, we eliminate non-minimal rules [2], which further strengthens the benefits of using normal forms, since non-minimal rules do not have to be considered any longer. 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 $H(r') \subset H(r) \wedge B(r') \subseteq B(r)$.

Definition 4. *Let P be a logic program with anonymous cycles over $\langle \Sigma, \Sigma_{an} \rangle$. We say that P is in* normal form *if the following conditions hold:*

- *for every $a \in \Sigma$ 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.*

Note that though the restrictions on appearance of atoms in the definition of the normal form are only on non-anonymous atoms, they are met by the anonymous ones too, thanks to the restrictions within the definition of programs with anonymous cycles. The next definition shows how to transform any program into one in normal form.

Definition 5. *Let P be a logic program with anonymous cycles over $\langle \Sigma, \Sigma_{an} \rangle$. The normal form $NF(P)$ is obtained by:*

1. *removing from P all tautological rules r, i.e. rules with $H(r) \cap B^+(r) \neq \emptyset$, with $B^+(r) \cap B^-(r) \neq \emptyset$ or with $B^-(r) \cap B^{--}(r) \neq \emptyset$;*
2. *removing, from the remaining rules, occurrences of double negated atoms from the body, if the atoms appear in the positive body of the same rule;*
3. *removing, from the remaining rules r, the atoms from the head of r that also occur in the negated body of r;*
4. *finally, removing from the resulting program P' all rules r that are not minimal.*

Note that the above construction ensures that all items of Def. 4 are satisfied, namely the first item of Def. 4 is ensured by condition 1. and 2. of Def. 5, the second by conditions 1. and 3., and the third by conditions 1. and 4. Notably, not only we can show that the construction of $NF(P)$ is correct, i.e. that $NF(P)$ is in normal form, but, additionally, we can show that it is strongly equivalent to the original program P.

Proposition 1. *Let P be an logic program with anonymous cycles. 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 an logic program with anonymous cycles. 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 these can be efficiently computed and are syntactically equal to the original programs apart from redundancies in the rules.

Forgetting about an atom from a program while satisfying **(SP)** should imply the preservation of the implicit dependencies between the atoms that are not forgotten.

Example 2. Consider the following program P:

$$a \leftarrow q \qquad\qquad q \leftarrow c \qquad\qquad q \leftarrow d$$

Whenever c or d are true, a is indirectly implied via q. Therefore, when forgetting about q from P, the implicit relationship between a and c, and that between a and d should be preserved. This can be expressed using the following rules without q:

$$a \leftarrow c \qquad\qquad a \leftarrow d$$

These rules correspond to replacing the positive occurrences of q in a rule body with the body of the rules in which q appears in their head.

If q is not the only atom in the head of a rule, then we need to consider these additional atoms in the head of the resulting rule.

Example 3. Consider the following program P:

$$a \leftarrow q \qquad\qquad q \vee b \leftarrow c$$

When forgetting about q from P the implicit relation between c and a must be preserved. This can be expressed by the rule:

$$a \vee b \leftarrow c$$

But what happens if the atom to be forgotten appears in the negative body of a rule?

Example 4. Consider the following program P:

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

In this case, there is an implicit relationship between a and the atoms b, c, and d. When forgetting about q from P such relationship must be preserved. For the literal $not\ q$ in the body of the first rule to be true we must have, for each rule r in which q appears in the head, either the body r is false (c in the case of the second rule and d in the case of the third rule), or the other atoms that appear in head of r must not be false (b in the case of the second rule). This can be represented by the following two rules:

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

This problem has been tackled by collecting a set of conjunctions of literals [6,16], each of which can be used to replace $not\ q$, but preserves its truth value.

Accordingly, we now generalize the notion of as-dual from [16], for which we need to introduce some auxiliary functions first. Let N be the function that applies a number of negation symbols to literals. Formally, for all $p \in \Sigma$, $N^0(p) = p$, $N^0(not\, p) = not\, p$, $N^0(not\, not\, p) = not\, not\, p$, $N^1(p) = N^1(not\, not\, p) = not\, p$, $N^1(not\, p) = not\, not\, p$, $N^2(p) = N^2(not\, not\, p) = not\, not\, p$, $N^2(not\, p) = not\, p$. For a set of literals S, $N^i(S) = \{N^i(s) : s \in S\}$. 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 $H^{\setminus q}(r) = H(r) \setminus \{q\}$.

To make this notion concrete, we introduce the as-dual $\mathcal{D}^q_{as}(P)$ for forgetting about q from P that collects the set of conjunctions of literals, that can be used to replace $not\, q$, the negated occurrence of the atom to be forgotten.

$$\begin{aligned}\mathcal{D}^q_{as}(P) = \{\{N^1(l_1), \ldots, N^1(l_m)\} \cup \{N^2(l_{m+1}), \ldots, N^2(l_n)\} : \\ l_i \in B^{\setminus q}(r_i), 1 \leq i \leq m, l_j \in H^{\setminus q}(r_j), m+1 \leq j \leq n, \\ \langle \{r_1, \ldots, r_m\}, \{r_{m+1}, \ldots, r_n\} \rangle \text{ is a partition of } P\}\end{aligned}$$

The idea is to pass to the operator all the rules that have q in their head as an argument. Then, we consider the possible partitions $\langle F, T\rangle$ of P, and the sets obtained by collecting 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. This definition covers all possible cases to provide the set of all rules for a program P such that the considered q cannot be derived. In particular, there are two interesting corner cases: If there is no rule with q in its head, i.e. the input program P is empty, $\mathcal{D}^q_{as}(P) = \{\emptyset\}$, meaning that negating q requires no atom to have a particular truth value. Furthermore, if P contains q as a fact, $\mathcal{D}^q_{as}(P) = \emptyset$, because it is impossible to negate q.

In the examples above, when forgetting about q from a program P, we were able to capture the implicit relationships using rules only over the remaining atoms. As already mentioned, the impossibility results in [11] show that this is in general not possible. In fact, if the atom to be forgotten has self-cycles, it may be the case that we cannot faithfully represent the implicit relationships between the remaining atoms using only rules over these remaining atoms. In theses cases we consider the use of anonymous atoms within anonymous cycles.

Example 5. Consider the following program P:

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

When b is not true, the first rule does not allow us to conclude q, and therefore c must be true by the third rule. This implicit relationship between b and c can be captured by the rule $c \leftarrow not\ b$, which is obtained by the substitution pattern mentioned in the previous examples. Whenever b is true, the self-cycle on q of the first rule generates a choice between a and c. Such choice cannot be represented using only rules over a, b and c. We therefore use anonymous cycles to generate such choice. So, additionally to the rule $c \leftarrow not\ b$, the result of forgetting about q from P has also the rules:

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

where δ_q is a fresh anonymous atom from Σ_{an}. These rules faithfully capture the implicit relationship between a, b and c in P.

We are now ready to present the formal definition of the operator f_{AC}. As this definition is technically involved, we will first present the new operator itself that allows forgetting about a single atom from a given program and subsequently explain and illustrate its definition. Forgetting about a set of atoms iteratively is presented subsequently.

In order to guarantee the uniqueness of the construction of the operator f_{AC}, we assume a fixed enumeration $\delta_0, \delta_1, \ldots \delta_n, \ldots$ of the elements of Σ_{an}.

Definition 6. *Let P be a program with anonymous cycles over $\langle \Sigma, \Sigma_{an}\rangle$, and $q \in \Sigma$. Let $P_{nf} = NF(P)$ be the normal form of P and δ_q the anonymous atom with the lowest index that does not occur in P. Consider the sets*

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

The result of forgetting about q in P, $f_{AC}(P, q)$, is the normal form of the program composed of the following rules:

- *each* $r \in R$
- *for each* $r_4 \in R_4$
  - **1a** *for each* $r_0 \in R_0$
    $H(r_0) \cup H^{\setminus q}(r_4) \leftarrow B^{\setminus q}(r_0) \cup B(r_4)$
  - **1b** *for each* $r_2 \in R_2$
    $H(r_2) \leftarrow B^{\setminus q}(r_2) \cup N^1(H^{\setminus q}(r_4)) \cup N^2(B(r_4))$
- *for each* $r' \in R_1 \cup R_4$
  - **2** *for each* $D \in \mathcal{D}^q_{as}(R_3 \cup R_4 \setminus \{r'\})$
    $H^{\setminus q}(r') \leftarrow B^{\setminus q}(r') \cup D$
- *for each* $r_3 \in R_3$
  - **3a** *for each* $r_0 \in R_0$
    $H(r_0) \cup H^{\setminus q}(r_3) \leftarrow B^{\setminus q}(r_0) \cup B^{\setminus q}(r_3) \cup \{\delta_q\}$
  - **3b** *for each* $r_2 \in R_2$
    $H(r_2) \leftarrow B^{\setminus q}(r_2) \cup N^1(H^{\setminus q}(r_3)) \cup N^2(B^{\setminus q}(r_3)) \cup \{\delta_q\}$
- *if* $R_3 \neq \emptyset$
  - **4** *for each* $r' \in R_1 \cup R_4$, $D \in \mathcal{D}^q_{as}(R_4 \setminus \{r'\})$
    $H^{\setminus q}(r') \leftarrow B^{\setminus q}(r') \cup D \cup \{not\ \delta_q\}$
  - **AC** $\delta_q \leftarrow not\ not\ \delta_q$

The first step is to obtain the normal form P_{nf} of P using Def. 5. Then, five sets of rules, R_0, R_1, R_2, R_3, and R_4, are defined over P_{nf}, in each of which q appears in the rules in a different form. In addition, R contains all rules from P_{nf} that do not mention q. These latter rules are preserved in the final result of forgetting.

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

Derivation rules **1a** and **1b** connect rules in which q occurs positively in the body with non-cyclic supports of q. In the case of Ex. 2 and 3 the occurrences of q in rule bodies are replaced with the body literals of rules that have q in the head.

Derivation rule **2** replaces negative occurrences of q in the body of rules by a proof that q cannot be derived, i.e., an element of $\mathcal{D}^q_{as}(R_3 \cup R_4)$. This is illustrated in Ex. 4. If P does not have rules with cyclic support on q, then only rules **1a**, **1b**, and **2** are used to obtain the result of forgetting about q from P.

If P has rules with cyclic support for q, i.e., R_3 is not empty, then f_{AC} creates an anonymous cycle with a fresh anonymous atom δ_q from Σ_{an}. This anonymous atom is then used as an arbiter between rules derived by **3a**, **3b** and **4**. The derivation rules **3a** and **3b** replace the positive occurrences of q in rule bodies by the anonymous atom δ_q, along with the body literals of the corresponding rule of R_3. Derivation rule **4** replaces negative occurrences of q in rules by a proof that there is no non-cyclic support for q, i.e., an element of $\mathcal{D}^q_{as}(R_4)$ and $not\ \delta_q$. These derivation rules are illustrated in Ex. 5.

We now prove that our operator f_{AC} behaves in a desirable way, in the sense that it preserves the HT-models of the original program (modulo the forgotten atom), thus necessarily preserving the (direct or indirect) relationships between the remaining atoms.

Theorem 1. *Let P be a program with anonymous cycles over $\langle \Sigma, \Sigma_{an} \rangle$, and $q \in \Sigma$. Then, we have that:*

$$\mathcal{HT}(f_{AC}(P, q)) = \mathcal{HT}(P)_{\|\{q\}}$$

We have defined an operator that forgets only one atom from a given program. In order to forget a set of atoms, we need to iterate the operator. Iteration is only possible if the operator is closed under the considered class of programs. Although fundamental, this property is not satisfied by some operators in the literature, namely the one in [16], thus not allowing the iteration of the operators. In the case of f_{AC}, we can prove that it is closed for the class of programs with anonymous cycles.

Proposition 3. *Let P be a logic program with anonymous cycles over $\langle \Sigma, \Sigma_{an} \rangle$ and $q \in \Sigma$. Then $f_{AC}(P, q)$ is a logic program with anonymous cycles over $\langle \Sigma \backslash \{q\}, \Sigma_{an} \rangle$.*

When forgetting a set of atoms iteratively, although the concrete result of forgetting depends on the order by which the atoms are forgotten, the following result shows that this is not a problem, in the sense that the results are strongly equivalent.

Proposition 4. *Let P be a program with anonymous cycles over $\langle \Sigma, \Sigma_{an} \rangle$, and $q_1, q_2 \in \Sigma$. Then we have that:*

$$f_{AC}(f_{AC}(P, q_1), q_2) \equiv f_{AC}(f_{AC}(P, q_2), q_1)$$

In order to define a concrete extension of the operator f_{AC} that allows forgetting a sets of atoms, we assume a fixed linear order on Σ, which we denote by $<$.

Definition 7. *Let P be a program with anonymous cycles over $\langle \Sigma, \Sigma_{an} \rangle$ and $V = \{q_1, q_2, \ldots, q_n\} \subseteq \Sigma$ a set of atoms with $q_i < q_j$ for each $1 \leq i < j \leq n$. The result of forgetting about V from P, denoted by $f^*_{AC}(P, V)$, is defined inductively as:*

$$f^*_{AC}(P, \emptyset) = P$$
$$f^*_{AC}(P, \{q_1, q_2, \ldots, q_n\}) = f^*_{AC}(f_{AC}(P, q_1), \{q_2, \ldots, q_n\})$$

Example 6. Consider the following program P:

$$a \leftarrow q. \qquad c \leftarrow p. \qquad p \leftarrow not\ q. \qquad q \leftarrow b, not\ p.$$

In order to forget about the set $V = \{p, q\}$ from P, we start by forgetting about p from P using f_{AC}. Since there are no cycles on p, f_{AC} does not introduce anonymous cycles, and we thus obtain the rules:

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

If we now subsequently forget also q, and since now there are cycles on the atom to be forgotten, f_{AC} introduces anonymous cycles, and thus the following program with anonymous cycles is obtained:

$$a \leftarrow b, \delta. \qquad c \leftarrow not\ \delta. \qquad c \leftarrow not\ b. \qquad \delta \leftarrow not\ not\ \delta.$$

which is the result of $f^*_{AC}(P, \{p, q\})$ [6].

[6] We could have chosen a different order by which p and q are forgotten, but, according to Prop. 4, the result would be strongly equivalent.

We can now state the main result of the paper. Namely, our new operator f^*_{AC} satisfies **(SP)**, thus making it the first (syntactical) operator that satisfies **(SP)** for all forgetting instances.

Theorem 2. *The operator f^*_{AC} satisfies* **(SP)***, i.e. for each program with anonymous cycles P over $\langle \Sigma, \Sigma_{an} \rangle$, and $V \subseteq \Sigma$, we have that*

$$\mathcal{AS}(f^*_{AC}(P, V) \cup R) = \mathcal{AS}(P \cup R)_{\|V},$$

for all programs R with $\Sigma(R) \subseteq \Sigma \backslash V$.

This result guarantees that we can use the operator f^*_{AC} to forget about a set of atoms from a program, while preserving all the dependencies between the atoms that were not forgotten. Given the already mentioned impossibility results of [11], the use of anonymous cycles is essential to allow the preservation of all dependencies. An important consequence of the previous theorem is the fact that f^*_{AC} satisfies also several other properties of forgetting. In particular, f^*_{AC} satisfies **(SI)**, which guarantees that all rules of a program P not mentioning the atoms to be forgotten be preserved when forgetting. It is worth noting that, although **(SI)** is a desirable property for forgetting, several classes of forgetting operators in the literature fail to satisfy this condition.

Proposition 5. *The operator f^*_{AC} satisfies* **(SI)**.

An important consequence of this result, together with the fact that f^*_{AC} is a syntactic operator, is the fact that the result of forgetting about a set of atoms from a program P according to f^*_{AC} can ben obtained by a syntactic manipulation of only those rules of P that mention the atoms to be forgotten, while the remaining rules are simply preserved in the result of forgetting.

5 Conclusions

We enhanced the language of logic programming to include anonymous cycles so that we can express relations between atoms that, under traditional logic programming semantics, were only possible to be expressed by the help of third atoms that act as arbiters. We then used this enhanced language to formulate a syntactic operator that forgets atoms from logic programs while obeying *strong persistence*, and is closed under this language, which means that it can be iterated, making it possible for any number of atoms to be forgotten from a program, in any order, while still maintaining $(\mathbf{SP})$. This is an improvement over existing operators, which either were only defined for very restricted classes of programs [16], or did not obey most desirable properties [25,6].

Future work includes investigating syntactic operators defined over traditional logic programs for the semantics in [13], that correspond to weaker versions of $(\mathbf{SP})$, as well as investigating how forgetting relates to other operations such as updating , and how it translates to hybrid knowledge representation formalisms [17,22,3].

Acknowledgments M. Berthold was partially supported by the International MSc Program in Computational Logic (MCL). R. Gonçalves, M. Knorr, and J. Leite were partially supported by FCT projects FORGET (PTDC/CCI-INF/32219/2017) and NOVA LINCS (UID/CEC/04516/2013).

References

1. Aguado, F., Cabalar, P., Fandinno, J., Pearce, D., Pérez, G., Vidal, C.: Forgetting auxiliary atoms in forks. In: Procs. of the ASPOCP@LPNMR. vol. 1868. CEUR-WS.org (2017)
2. Brass, S., Dix, J.: Semantics of (disjunctive) logic programs based on partial evaluation. J. Log. Program. **40**(1), 1–46 (1999)
3. Brewka, G., Ellmauthaler, S., Gonçalves, R., Knorr, M., Leite, J., Pührer, J.: Reactive multi-context systems: Heterogeneous reasoning in dynamic environments. Artif. Intell. **256** (2018)
4. Cabalar, P., Pearce, D., Valverde, A.: Minimal logic programs. In: Procs. of ICLP. Springer (2007)
5. Delgrande, J.P., Wang, K.: A syntax-independent approach to forgetting in disjunctive logic programs. In: Procs. of AAAI. pp. 1482–1488. AAAI Press (2015)
6. Eiter, T., Wang, K.: Semantic forgetting in answer set programming. Artif. Intell. **172**(14), 1644–1672 (2008)
7. Gonçalves, R., Janhunen, T., Knorr, M., Leite, J., Woltran, S.: Variable elimination for dlp-functions. In: Procs. of KR. pp. 643–644. AAAI Press (2018)
8. Gonçalves, R., Janhunen, T., Knorr, M., Leite, J., Woltran, S.: Forgetting in modular answer set programming. In: Procs. of AAAI. AAAI Press (2019)
9. Gonçalves, R., Knorr, M., Leite, J.: Forgetting in ASP: the forgotten properties. In: Procs. of JELIA. LNCS, vol. 10021, pp. 543–550 (2016)
10. Goncalves, R., Knorr, M., Leite, J.: The ultimate guide to forgetting in answer set programming. In: Procs. of KR. pp. 135–144. AAAI Press (2016)
11. Gonçalves, R., Knorr, M., Leite, J.: You can't always forget what you want: on the limits of forgetting in answer set programming. In: Procs. of ECAI. pp. 957–965. IOS Press (2016)
12. Gonçalves, R., Knorr, M., Leite, J.: Iterative variable elimination in ASP. In: Procs. of EPIA. LNCS, vol. 10423, pp. 643–656. Springer (2017)
13. 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)
14. Inoue, K., Sakama, C.: Negation as failure in the head. J. Log. Program. **35**(1), 39–78 (1998)
15. Inoue, K., Sakama, C.: Equivalence of logic programs under updates. In: Procs. of JELIA. Springer (2004)
16. Knorr, M., Alferes, J.J.: Preserving strong equivalence while forgetting. In: Procs. of JELIA. LNCS, vol. 8761, pp. 412–425. Springer (2014)
17. Knorr, M., Alferes, J.J., Hitzler, P.: Local closed world reasoning with description logics under the well-founded semantics. Artif. Intell. **175**(9-10), 1528–1554 (2011)
18. Leite, J.: A bird's-eye view of forgetting in answer-set programming. In: Procs. of LPNMR. LNCS, vol. 10377, pp. 10–22. Springer (2017)
19. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Trans. Comput. Log. **2**(4), 526–541 (2001)
20. Slota, M., Leite, J.: Back and forth between rules and SE-models. In: Procs. of LPNMR. Springer (2011)
21. Slota, M., Leite, J.: The rise and fall of semantic rule updates based on se-models. TPLP **14**(6), 869–907 (2014)
22. Slota, M., Leite, J., Swift, T.: On updates of hybrid knowledge bases composed of ontologies and rules. Artif. Intell. **229**, 33–104 (2015)
23. Wang, Y., Wang, K., Zhang, M.: Forgetting for answer set programs revisited. In: Rossi, F. (ed.) Procs. of IJCAI. pp. 1162–1168. IJCAI/AAAI (2013)
24. Wang, Y., Zhang, Y., Zhou, Y., Zhang, M.: Knowledge forgetting in answer set programming. J. Artif. Intell. Res. (JAIR) **50**, 31–70 (2014)
25. Zhang, Y., Foo, N.Y.: Solving logic program conflict through strong and weak forgettings. Artif. Intell. **170**(8-9), 739–778 (2006)