A Bird's-Eye View of Forgetting in Answer-Set Programming

João Leite(✉)

NOVA LINCS & Departamento de Informática,
Universidade Nova de Lisboa, Lisbon, Portugal
jleite@fct.unl.pt

Abstract. Forgetting is an operation that allows the removal, from a knowledge base, of middle variables no longer deemed relevant, while preserving all relationships (direct and indirect) between the remaining variables. When investigated in the context of Answer-Set Programming, many different approaches to forgetting have been proposed, following different intuitions, and obeying different sets of properties.

This talk will present a bird's-eye view of the complex landscape composed of the properties and operators of forgetting defined over the years in the context of Answer-Set Programming, zooming in on recent findings triggered by the formulation of the so-called *strong persistence*, a property based on the strong equivalence between an answer-set program and the result of forgetting modulo the forgotten atoms, which seems to best encode the requirements of the forgetting operation.

Keywords: Forgetting · Answer-Set Programming · Variable elimination

1 Introduction

Whereas keeping memory of information and knowledge has always been at the heart of research in Knowledge Representation and Reasoning, with tight connections to broader areas such as Databases and Artificial Intelligence, we have recently observed a growing attention being devoted to the complementary problem of *forgetting*.

Forgetting – or variable elimination – is an operation that allows the removal of *middle* variables no longer deemed relevant. It 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 [3], which includes the *right to*

The author was partially supported by FCT UID/CEC/04516/2013. This paper is substantially based on [1,2].

M. Balduccini and T. Janhunen (Eds.): LPNMR 2017, LNAI 10377, pp. 10–22, 2017.
DOI: 10.1007/978-3-319-61660-5_2

be forgotten. Recent applications of forgetting to cognitive robotics [4–6], resolving conflicts [7–10], and ontology abstraction and comparison [11–14], further witness its importance.

With its early roots in Boolean Algebra [15], forgetting has been extensively studied in the context of classical logic [7,16–21] and, more recently, in the context of logic programming, notably of Answer Set Programming (ASP). The non-monotonic rule-based nature of ASP creates very unique challenges to the development of forgetting operators – just as it happened with other belief change operations such as revision and update, cf. [22–28] – making it a special endeavour with unique characteristics distinct from those for classical logic.

Over the years, many have proposed different approaches to forgetting in ASP, through the characterization of the result of forgetting a set of atoms from a given program up to some equivalence class, and/or through the definition of concrete operators that produce a specific program for each input program and atoms to be forgotten [8,9,29–34].

All these approaches were typically proposed to obey some specific set of properties that their authors deemed adequate, some adapted from the literature on *classical* forgetting [30,33,35], others specifically introduced for the case of ASP [9,29–32,34]. Examples of such properties include *strengthened consequence*, which requires that the answer sets of the result of forgetting be bound to the answer-sets of the original program modulo the forgotten atoms, or the so-called *existence*, which requires that the result of forgetting belongs to the same class of programs admitted by the forgetting operator, so that the same reasoners can be used and the operator be iterated, among many others.

All this resulted is a *complex* landscape filled with operators and properties, with very little effort put into drawing a map that could help to better understand the relationships between properties and operators. This was recently addressed in [1], through the presentation of a systematic study of *forgetting* in Answer Set Programming (ASP), thoroughly investigating the different approaches found in the literature, their properties and relationships.

In the first part of this invited talk, we will present a bird's-eye view of this complex landscape investigated in [1].

One of the main conclusions drawn from observing the landscape of existing operators and properties is that there cannot be a one-size-fits-all forgetting operator for ASP, but rather a family of operators, each obeying a specific set of properties. Furthermore, it is clear that not all properties bear the same relevance. Whereas some properties can be very important, such as *existence*, since it guarantees that we can use the same automated reasoners after forgetting, despite not being a property specific of forgetting operators, other properties are less important, sometimes perhaps even questionable, as discussed in [1].

There is nevertheless one property – *strong persistence* [32] – which seems to best capture the essence of forgetting in the context of ASP. The property of *strong persistence* essentially requires that all existing relations between the atoms not to be forgotten be preserved, captured by requiring 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. Referring to the notation introduced in the appendix, an operator f is said to obey strong persistence if, for any program P and any set of atoms to be forgotten V, it holds that $\mathcal{AS}(\mathsf{f}(P,V) \cup R) = \mathcal{AS}(P \cup R)_{\|V}$, for all programs R not containing atoms in V, where $\mathsf{f}(P,V)$ denotes the result of forgetting V from P, $\mathcal{AS}(P)$ the answer sets of P, and $\mathcal{AS}(P)_{\|V}$ their restriction to atoms not in V.

Whereas it seems rather undisputed that *strong persistence* is a desirable property, it was not clear to what extent one could define operators that satisfy it. Whereas in [32], the authors proposed an operator that obeys such property, it is only defined for a restricted class of programs and can only be applied to forget a single atom from a program in a very limited range of situations.

The limits of forgetting while obeying *strong persistence* were investigated in [2]. There, after showing that sometimes it is simply not possible to forget some set of atoms from a program, while maintaining the relevant relations between other atoms, since the atoms to be forgotten play a pivotal role, the following three fundamental questions addressed: (a) *When can't we forget some set of atoms from an ASP while obeying strong persistence?*, (b) *When (and how) can we forget some set of atoms from an ASP while obeying strong persistence?*, and (c) *What can we forget from a specific ASP while obeying strong persistence?*

In the second part of this invited talk, we will zoom in on the limits of *forgetting* under *strong persistence* investigated in [2], and point to the future.

2 Forgetting in Answer-Set Programming

Forgetting. The principal idea of forgetting in ASP is to remove or hide certain atoms from a given program, while preserving its semantics for the remaining atoms. As the result, rather often, a representative up to some notion of equivalence between programs is considered. In this sense, many notions of forgetting for logic programs are defined semantically, i.e., they introduce a class of operators that satisfy a certain semantic characterization. Each single operator in such a class is then a concrete function that, given a program P and a non-empty set of atoms V to be forgotten, returns a unique program, the result of forgetting about V from P. Given a class of logic programs[1] $\mathcal{C}$ over $\mathcal{A}$, a *forgetting operator* (over $\mathcal{C}$) is a partial function $\mathsf{f} : \mathcal{C} \times 2^{\mathcal{A}} \to \mathcal{C}$ s.t. $\mathsf{f}(P,V)$ is a program over $\mathcal{A}(P)\backslash V$, for each $P \in \mathcal{C}$ and $V \subseteq \mathcal{A}$. We call $\mathsf{f}(P,V)$ the *result of forgetting about V from P*. Unless stated otherwise, we will be focusing on $\mathcal{C} = \mathcal{C}_e$, and we leave $\mathcal{C}$ implicit. Furthermore, f is called *closed* for $\mathcal{C}' \subseteq \mathcal{C}$ if, for every $P \in \mathcal{C}'$ and $V \subseteq \mathcal{A}$, we have $\mathsf{f}(P,V) \in \mathcal{C}'$. A *class* F *of forgetting operators (over $\mathcal{C}$)* is a set of forgetting operators (over $\mathcal{C}'$) s.t. $\mathcal{C}' \subseteq \mathcal{C}$.

Properties. Over the years, many have introduced a variety of properties that forgetting operators *should* obey, which we now briefly discuss.

The first three properties were proposed by Eiter and Wang [9], though not formally introduced as such. The first two were in fact guiding principles for

[1] See Appendix for definitions and notation on Answer-Set Programming.

defining their notion of forgetting, while the third was later formalized by Wang et al. [31]:

– F[2] satisfies *strengthened Consequence* **(sC)** if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}$ and $V \subseteq \mathcal{A}$, we have $\mathcal{AS}(\mathsf{f}(P,V)) \subseteq \mathcal{AS}(P)_{\|V}$. Strengthened Consequence requires that the answer sets of the result of forgetting be answer sets of the original program, ignoring the atoms to be forgotten.

– F satisfies *weak Equivalence* **(wE)** if, for each $\mathsf{f} \in \mathsf{F}$, $P, P' \in \mathcal{C}$ and $V \subseteq \mathcal{A}$, we have $\mathcal{AS}(\mathsf{f}(P,V)) = \mathcal{AS}(\mathsf{f}(P',V))$ whenever $\mathcal{AS}(P) = \mathcal{AS}(P')$. Weak Equivalence requires that forgetting preserves equivalence of programs.

– F satisfies *Strong Equivalence* **(SE)** if, for each $\mathsf{f} \in \mathsf{F}$, $P, P' \in \mathcal{C}$ and $V \subseteq \mathcal{A}$: if $P \equiv P'$, then $\mathsf{f}(P,V) \equiv \mathsf{f}(P',V)$. Strong Equivalence requires that forgetting preserves strong equivalence of programs.

The next three properties were introduced by Zhang and Zhou [35] in the context of forgetting in modal logics, and later adopted by Wang et al. [30,33] for forgetting in ASP:

– F satisfies *Weakening* **(W)** if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}$ and $V \subseteq \mathcal{A}$, we have $P \models_{\mathsf{HT}} \mathsf{f}(P,V)$. Weakening requires that the HT-models of the original program also be HT-models of the result of forgetting, thus implying that the result of forgetting has at most the same consequences as the original program.

– F satisfies *Positive Persistence* **(PP)** if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}$ and $V \subseteq \mathcal{A}$: if $P \models_{\mathsf{HT}} P'$, with $P' \in \mathcal{C}$ and $\mathcal{A}(P') \subseteq \mathcal{A}\backslash V$, then $\mathsf{f}(P,V) \models_{\mathsf{HT}} P'$. Positive Persistence requires that the HT-consequences of the original program not containing atoms to be forgotten be preserved in the result of forgetting.

– F satisfies *Negative Persistence* **(NP)** if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}$ and $V \subseteq \mathcal{A}$: if $P \not\models_{\mathsf{HT}} P'$, with $P' \in \mathcal{C}$ and $\mathcal{A}(P') \subseteq \mathcal{A}\backslash V$, then $\mathsf{f}(P,V) \not\models_{\mathsf{HT}} P'$. Negative Persistence requires that a program not containing atoms to be forgotten not be a HT-consequence of the result of forgetting, unless it was already a HT-consequence of the original program.

The property *Strong (addition) Invariance* was introduced by Wong [29], and assigned this name in [1]:

– F satisfies *Strong (addition) Invariance* **(SI)** if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}$ and $V \subseteq \mathcal{A}$, we have $\mathsf{f}(P,V) \cup R \equiv \mathsf{f}(P \cup R, V)$ for all programs $R \in \mathcal{C}$ with $\mathcal{A}(R) \subseteq \mathcal{A}\backslash V$. Strong (addition) Invariance requires that it be (strongly) equivalent to add a program without the atoms to be forgotten before or after forgetting.

The property called *existence* was discussed by Wang et al. [30], formalized by Wang et al. [31], and refined by [1]. It requires that a result of forgetting for P in $\mathcal{C}$ exists in the class $\mathcal{C}$, important to iterate:

– F satisfies *Existence for* $\mathcal{C}$ **($E_{\mathcal{C}}$)**, i.e., F is *closed for a class of programs* $\mathcal{C}$ if there exists $\mathsf{f} \in \mathsf{F}$ s.t. f is closed for $\mathcal{C}$. Existence for class $\mathcal{C}$ requires that the a result of forgetting for a program in $\mathcal{C}$ exists in the class $\mathcal{C}$. Operators that satisfy Existence for class $\mathcal{C}$ are said to be closed for that class.

The property *Consequence Persistence* was introduced by Wang et al. [31] building on the ideas behind **(sC)** by Eiter and Wang [9]:

[2] Unless stated otherwise, F is a class of forgetting operators, and $\mathcal{C}$ the class of programs over $\mathcal{A}$ of a given $\mathsf{f} \in \mathsf{F}$.

– F satisfies *Consequence Persistence* **(CP)** if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}$ and $V \subseteq \mathcal{A}$, we have $\mathcal{AS}(\mathsf{f}(P,V)) = \mathcal{AS}(P)_{\|V}$. Consequence persistence requires that the answer sets of the result of forgetting correspond exactly to the answer sets of the original program, ignoring the atoms to be forgotten.

The following property was introduced by Knorr and Alferes [32] with the aim of imposing the preservation of all dependencies contained in the original program:

– F satisfies *Strong Persistence* **(SP)** if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}$ and $V \subseteq \mathcal{A}$, we have $\mathcal{AS}(\mathsf{f}(P,V) \cup R) = \mathcal{AS}(P \cup R)_{\|V}$, for all programs $R \in \mathcal{C}$ with $\mathcal{A}(R) \subseteq \mathcal{A} \backslash V$. Strong Persistence strengthens **(CP)** by imposing that the correspondence between answer-sets of the result of forgetting and those of the original program be preserved in the presence of any additional set of rules not containing the atoms to be forgotten.

The final property here[3] is due to Delgrande and Wang [34], although its name was assigned in [1]:

– F satisfies *weakened Consequence* **(wC)** if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}$ and $V \subseteq \mathcal{A}$, we have $\mathcal{AS}(P)_{\|V} \subseteq \mathcal{AS}(\mathsf{f}(P,V))$. Weakened Consequence requires that the answer sets of the original program be preserved while forgetting, ignoring the atoms to be forgotten.

These properties are not orthogonal to one another: **(CP)** is incompatible with **(W)** as well as with **(NP)** (for F closed for $\mathcal{C}$, where $\mathcal{C}$ contains normal logic programs); **(W)** is equivalent to **(NP)**; **(SP)** implies **(PP)**; **(SP)** implies **(SE)**; **(W)** and **(PP)** together imply **(SE)**; **(CP)** and **(SI)** together are equivalent to **(SP)**; **(sC)** and **(wC)** together are equivalent to **(CP)**; **(CP)** implies **(wE)**; **(SE)** and **(SI)** together imply **(PP)**.

Operators. Over the years, many operators of forgetting have been introduced, implementing certain intuitions and obeying particular sets of properties.

Strong and Weak Forgetting. The first proposals are due to Zhang and Foo [8] introducing two syntactic operators for normal logic programs, termed Strong and Weak Forgetting. Both start with computing a reduction corresponding to the well-known weak partial evaluation (WGPPE) [38]. Then, the two operators differ on how they subsequently remove rules containing the atom to be forgotten. In Strong Forgetting, all rules containing the atom to be forgotten are simply removed. In Weak Forgetting, rules with negative occurrences of the atom to be forgotten in the body are kept, after such occurrences are removed. The motivation for this difference is whether such negative occurrences of the atom to be forgotten are seen as support for the rule head (Strong) or not (Weak). Both operators are closed for $\mathcal{C}_n$.

Semantic Forgetting. Eiter and Wang [9] proposed Semantic Forgetting to improve on some of the shortcomings of the two purely syntax-based operators of Zhang and Foo [8]. The basic idea is to characterize a result of forgetting just by its answer sets, obtained by considering only the minimal sets among the answer sets of the initial program ignoring the atoms to be forgotten.

[3] An additional set of properties was introduced in [29]. The reader is referred to [36,37] for a detailed discussion regarding these properties.

Three concrete algorithms are presented, two based on semantic considerations and one syntactic. Unlike the former, the latter is not closed for $\mathcal{C}_d^+$ and $\mathcal{C}_n^+$ ($^+$ denotes the restriction to consistent programs), since double negation is required in general.

Semantic Strong and Weak Forgetting. Wong [29] argued that semantic forgetting should not be focused on answer sets only, as these do not contain all the information present in a program. He defined two classes of forgetting operators for disjunctive programs, building on HT-models. The basic idea is to start with the set of rules HT-entailed by the original program without those with positive occurrences of the atoms to be forgotten, and after removing positive occurrences of the atoms to be forgotten from the head of rules, whenever their negation appears in the body, and then, in Semantic Strong Forgetting, rules containing the atoms to be forgotten are removed, while in Semantic Weak Forgetting rules with negative (resp. positive) occurrences of the atoms to be forgotten in the body (resp. head) are kept, after such occurrences are removed. Wong [29] defined one construction closed for $\mathcal{C}_d$.

HT-Forgetting. Wang et al. [30,33] introduced HT-Forgetting, building on properties introduced by Zhang and Zhou [35] in the context of modal logics, with the aim of overcoming problems with Wongs notions, namely that each of them did not satisfy one of the properties **(PP)** and **(W)**. HT-Forgetting is defined for extended programs, characterising the set of HT-models of the result of forgetting as being composed of the HT-models of the original program, modulo any occurrence of the atoms to be forgotten. A concrete operator is presented [33] that is shown to be closed for $\mathcal{C}_e$ and $\mathcal{C}_H$, and it is also shown that no HT-Forgetting operator exists that is closed for either $\mathcal{C}_d$ or $\mathcal{C}_n$.

SM-Forgetting. Wang et al. [31] defined a modification of HT-Forgetting, SM-Forgetting, for extended programs, with the objective of preserving the answer sets of the original program (modulo the forgotten atoms). As with HT-Forgetting, it is defined for extended programs though a characterisation of the HT-models of the result of forgetting, which are taken to be maximal subsets of the HT-models of the original program, modulo any occurrence of the atoms to be forgotten, such that the set of their answer-sets coincides with the set of answer-sets of the original program, modulo the forgotten atoms. A concrete operator is provided that is shown to be closed for $\mathcal{C}_e$ and $\mathcal{C}_H$. It is also shown that no SM-Forgetting operator exists that is closed for either $\mathcal{C}_d$ or $\mathcal{C}_n$.

Strong AS-Forgetting. Knorr and Alferes [32] introduced Strong AS-Forgetting with the aim of preserving both the answer sets of the original program, and also those of the original program augmented by any set of rules over the signature without the atoms to be forgotten. A concrete operator is defined for $\mathcal{C}_{nd}$, but not closed for $\mathcal{C}_n$ and only defined for certain programs with double negation.

SE-Forgetting. Delgrande and Wang [34] recently introduced SE-Forgetting based on the idea that forgetting an atom from a program is characterized by the set of those SE-consequences, i.e., HT-consequences, of the program that do not mention the atoms to be forgotten. The notion is defined for disjunctive programs building on an inference system by Wong [39] that preserves strong equivalence.

	sC	**wE**	**SE**	**W**	**PP**	**NP**	**SI**	**CP**	**SP**	**wC**	$\mathbf{E}_{\mathcal{C}_H}$	$\mathbf{E}_{\mathcal{C}_n}$	$\mathbf{E}_{\mathcal{C}_d}$	$\mathbf{E}_{\mathcal{C}_{nd}}$	$\mathbf{E}_{\mathcal{C}_e}$
F_{strong}	×	×	×	✓	×	✓	✓	×	×	×	✓	✓	-	-	-
F_{weak}	×	×	×	×	✓	×	✓	×	×	×	✓	✓	-	-	-
F_{sem}	✓	✓	×	×	×	×	×	×	×	×	✓	✓	✓	-	-
F_{S}	×	×	✓	✓	✓	✓	×	×	×	×	✓	×	✓	-	-
F_{W}	✓	✓	✓	×	✓	×	✓	×	×	×	✓	✓	✓	-	-
F_{HT}	×	×	✓	✓	✓	✓	✓	×	×	×	✓	×	×	×	✓
F_{SM}	✓	✓	✓	×	✓	×	×	✓	×	✓	✓	×	×	×	✓
F_{Sas}	✓	✓	✓	×	✓	×	✓	✓	✓	✓	✓	×	×	×	×
F_{SE}	×	×	✓	✓	✓	✓	×	×	×	×	✓	×	✓	-	-

Fig. 1. Satisfaction of properties for known classes of forgetting operators. For class F and property **P**, '✓' represents that F satisfies **P**, '×' that F does not satisfy **P**, and '-' that F is not defined for the class $\mathcal{C}$ in consideration.

An operator is provided, which is closed for $\mathcal{C}_d$. Gonçalves et al. [1] have shown SE-Forgetting to coincide with Semantic Strong Forgetting [29].

Figure 1 summarises the satisfaction of properties for known classes of forgetting operators.

3 Forgetting Under Strong Persistence

Among the desirable properties of classes of forgetting operators recalled in the previous section, *strong persistence* **(SP)** [32] is of particular interest, as it ensures that forgetting preserves all existing relations between all atoms occurring in the program, but the forgotten. In this sense, a class of operators satisfying **(SP)** removes the desired atoms, but has no negative semantical effects on the remainder. The importance of **(SP)** is also witnessed by the fact that a class of operators that satisfies **(SP)** also satisfies all the other previously mentioned properties with the exception of **(W)** and **(NP)**, which happen to be equivalent and can hardly be considered desirable [1].

However, determining a forgetting operator that satisfies **(SP)** has been a difficult problem, since, for the verification whether a certain program P' should be the result of forgetting about V from P, none of the well-established equivalence relations can be used, i.e., neither equivalence nor strong equivalence hold in general between P and P', not even relativized equivalence [40], even though it is close in spirit to the ideas of **(SP)**. Hence, maybe not surprisingly, there was no known general class of operators that satisfies **(SP)** and which is closed (for the considered class of logic programs).

And, until recently, the two known positive results concerning the satisfiability of **(SP)** were the existence of several known classes of operators that satisfy **(SP)** *when restricted to Horn programs* [1], which is probably of little relevance given the crucial role played by (default) negation in ASP, and the existence of one specific operator that permits forgetting about V from P while satisfying **(SP)** [32], but only in a very restricted range of situations based on a non-trivial

syntactical criterion which excludes large classes of cases where forgetting about V from P is possible.

All this begged the question of whether there exists a forgetting operator, defined over a class of programs $\mathcal{C}$ beyond the class of Horn programs, that satisfies **(SP)**, which was given a negative answer in [2]: – it is not always possible to forget a set of atoms from a given logic program satisfying the property **(SP)**.

Whereas this negative result shows that in general it is not always possible to forget while satisfying **(SP)**, its proof presented in [2] provided some hints on why this is the case. Some atoms play an important role in the program, being pivotal in establishing the relations between the remaining atoms, making it simply not possible to forget them and expect that the relations between other atoms be preserved. That is precisely what happens with the pair of atoms p and q in the program

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

It is simply not possible to forget them both and expect all the semantic relations between a and b to be kept. No program over atoms $\{a, b\}$ would have the same answer sets as those of the original program (modulo p and q), when both are extended with an arbitrary set of rules over $\{a, b\}$.

This observation lead to another central question: under what circumstances is it not possible to forget about a given set of atoms V from P while satisfying **(SP)**? In particular, given a concrete program, which sets of atoms play such a pivotal role that they cannot be jointly forgotten without affecting the semantic relations between the remaining atoms in the original program?

This question was answered in [2] through the introduction of a criterion (Ω) which characterizing the instances $\langle P, V\rangle$ for which we cannot expect forgetting operators to satisfy $\textbf{(SP)}_{\langle P,V\rangle}$.[4]

Definition 1 (Criterion Ω). *Let P be a program over $\mathcal{A}$ and $V \subseteq \mathcal{A}$. An instance $\langle P, V\rangle$* satisfies criterion Ω *if there exists $Y \subseteq \mathcal{A} \setminus V$ 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

$$\begin{aligned} R^{Y,A}_{\langle P,V\rangle} &= \{X \setminus V \mid \langle X, Y \cup A\rangle \in \mathcal{HT}(P)\} \\ Rel^{Y}_{\langle P,V\rangle} &= \{A \subseteq V \mid \langle Y \cup A, Y \cup A\rangle \in \mathcal{HT}(P) \text{ and} \\ &\qquad \nexists A' \subset A \text{ such that } \langle Y \cup A', Y \cup A\rangle \in \mathcal{HT}(P)\}. \end{aligned}$$

It turns out that Ω is a necessary and sufficient criterion to determine that some set of atoms V cannot be forgotten from a program P while satisfying *strong persistence*.

[4] $\textbf{(SP)}_{\langle P,V\rangle}$ is a restriction of property **(SP)** to specific forgetting instances. A forgetting operator f over $\mathcal{C}$ satisfies $\textbf{(SP)}_{\langle P,V\rangle}$ if $\mathcal{AS}(\mathsf{f}(P, V) \cup R) = \mathcal{AS}(P \cup R)_{\|V}$, for all programs $R \in \mathcal{C}$ with $\mathcal{A}(R) \subseteq \mathcal{A} \setminus V$.

Whereas at a technical level, criterion Ω is closely tied to certain conditions on the HT-models of the program at hand, it seems that what cannot be forgotten from a program are atoms used in rules that are somehow equivalent to *choice rules* [41], and those atoms are pivotal in the sense that they play an active role in determining the truth of other atoms in some answer sets i.e., there are rules whose bodies mention these atoms and they are true at least in some answer sets.

Nevertheless, sometimes it is possible to forget while satisfying *strong persistence* and, in such cases, the following class $\mathsf{F_{SP}}$ of forgetting operators, dubbed *SP-Forgetting*, precisely characterises the desired result of forgetting:

$$\mathsf{F_{SP}} = \{\mathsf{f} \mid \mathcal{HT}(\mathsf{f}(P,V)) = \{\langle X, Y\rangle \mid Y \subseteq \mathcal{A}(P) \backslash V \wedge X \in \bigcap \mathcal{R}^{Y}_{\langle P,V\rangle}\}\}$$

Thus, given an instance $\langle P, V\rangle$, we can test whether Ω is not satisfied, i.e., whether we are allowed to forget V from P while preserving **(SP)**, in which case the HT-models that characterise a result can be obtained from $\mathsf{F_{SP}}$. It was further shown in [2] that $\mathsf{F_{SP}}$ is closed in the general case and for Horn programs, but not for disjunctive or normal programs.

If we restrict our attention to the cases where we can forget, i.e., where the considered instance does not satisfy Ω, then most of the properties mentioned before are satisfied. In particular, restricted to instances $\langle P, V\rangle$ that do not satisfy Ω, $\mathsf{F_{SP}}$ satisfies **(sC)**, **(wE)**, **(SE)**, **(PP)**, **(SI)**, **(CP)**, **(SP)** and **(wC)**. The properties which are not satisfied – **(W)** and **(NP)** – have been proved orthogonal to **(SP)** [1], hence of little relevance in our view.

4 Outlook

We began by presenting a bird's-eye view of *forgetting* in Answer Set Programming (ASP), covering the different approaches found in the literature, their properties and relationships. We then zoomed in on the important property of *strong persistence*, and reviewed the most relevant known results, including that it is not always possible to forget a set of atoms from a program while obeying this property, a precise characterisation of what can and cannot be forgotten from a program established through a necessary and sufficient criterion, and a characterisation of the class of forgetting operators that achieve the correct result whenever forgetting is possible.

But what happens if we *must* forget, but cannot do it without violating *strong persistence*? This may happen for legal and privacy issues, including, for example, the implementation of court orders to eliminate certain pieces of illegal information. Investigating weaker requirements, e.g. by imposing only a subset of the three properties – **(sC)**, **(wC)** and **(SI)** – that together compose **(SP)**, or by considering weaker notions of equivalence such as *uniform equivalence* [42,43], is the subject of ongoing work.

Other interesting avenues for future research include investigating different forms of forgetting which may be required in practice, such as those that preserve

some aggregated meta-level information about the forgotten atoms, or even going beyond maintaining all relationships between non-forgotten atoms which may be required by certain legislation.

Acknowledgments. I would like to thank my close colleagues Ricardo Gonçalves and Matthias Knorr for all their dedication and contributions to our joint projects, turning it into a more fun and rewarding ride.

A Answer-Set Programming

We assume a *propositional signature* $\mathcal{A}$, a finite set of propositional atoms[5]. An *(extended) logic program* P over $\mathcal{A}$ is a finite set of *(extended) 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 $\mathcal{A}$.[6] Such rules r are also commonly written in a more succinct way as

$$A \leftarrow B, not\ C, not\ not\ D\ , \qquad (2)$$

where we have $A = \{a_1, \ldots, a_k\}$, $B = \{b_1, \ldots, b_l\}$, $C = \{c_1, \ldots, c_m\}$, $D = \{d_1, \ldots, d_n\}$, and we will use both forms interchangeably. By $\mathcal{A}(P)$ we denote the set of atoms appearing in P. This class of logic programs, $\mathcal{C}_e$, 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*, $\mathcal{C}_d$, $\mathcal{C}_n$, and $\mathcal{C}_H$, are defined resp. as a finite set of disjunctive, normal, and Horn rules. We also call extended rules with $k \leq 1$ *non-disjunctive*, thus admitting a non-standard class $\mathcal{C}_{nd}$, called *non-disjunctive programs*, different from normal programs. Given a program P and a set I of atoms, the *reduct* P^I is defined as $P^I = \{A \leftarrow B : r \text{ of the form (2) in } P, C \cap I = \emptyset, D \subseteq I\}$.

An *HT-interpretation* is a pair $\langle X, Y\rangle$ s.t. $X \subseteq Y \subseteq \mathcal{A}$. Given a program P, an HT-interpretation $\langle X, Y\rangle$ is an *HT-model of* P if $Y \models P$ and $X \models P^Y$, where $\models$ denotes the standard consequence relation for classical logic. We admit that the set of HT-models of a program P are restricted to $\mathcal{A}(P)$ even if $\mathcal{A}(P) \subset \mathcal{A}$. 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 $R \in \mathcal{C}_e$. It is well-known that $P_1 \equiv P_2$ exactly when $\mathcal{HT}(P_1) = \mathcal{HT}(P_2)$ [45]. We say that P' is an *HT-consequence* of P, denoted by $P \models_{\mathsf{HT}} P'$, whenever $\mathcal{HT}(P) \subseteq \mathcal{HT}(P')$. The V*-exclusion* of a set of answer sets (a set of HT-interpretations) $\mathcal{M}$, denoted $\mathcal{M}_{\|V}$, is $\{X \backslash V \mid X \in \mathcal{M}\}$ $(\{\langle X \backslash V, Y \setminus V\rangle \mid \langle X, Y\rangle \in \mathcal{M}\})$. Finally, given two sets of atoms $X, X' \subseteq \mathcal{A}$, we write $X \sim_V X'$ whenever $X \backslash V = X' \backslash V$.

[5] Often, the term propositional variable is used synonymously.

[6] Extended logic programs [44] are actually more expressive, but this form is sufficient here.

References

1. Goncalves, R., Knorr, M., Leite, J.: The ultimate guide to forgetting in answer set programming. In: Baral, C., Delgrande, J., Wolter, F. (eds.) Proceedings of KR, pp. 135–144. AAAI Press (2016)
2. Gonçalves, R., Knorr, M., Leite, J.: You can't always forget what you want: on the limits of forgetting in answer set programming. In: Fox, M.S., Kaminka, G.A. (eds.) Proceedings of ECAI. IOS Press (2016)
3. European Parliament: General data protection regulation. Official Journal of the European Union L119/59, May 2016
4. Lin, F., Reiter, R.: How to progress a database. Artif. Intell. **92**(1–2), 131–167 (1997)
5. Liu, Y., Wen, X.: On the progression of knowledge in the situation calculus. In: Walsh, T. (ed.) Proceedings of IJCAI, IJCAI/AAAI, pp. 976–982 (2011)
6. Rajaratnam, D., Levesque, H.J., Pagnucco, M., Thielscher, M.: Forgetting in action. In: Baral, C., Giacomo, G.D., Eiter, T. (eds.) Proceedings of KR. AAAI Press (2014)
7. Lang, J., Liberatore, P., Marquis, P.: Propositional independence: formula-variable independence and forgetting. J. Artif. Intell. Res. (JAIR) **18**, 391–443 (2003)
8. Zhang, Y., Foo, N.Y.: Solving logic program conflict through strong and weak forgettings. Artif. Intell. **170**(8–9), 739–778 (2006)
9. Eiter, T., Wang, K.: Semantic forgetting in answer set programming. Artif. Intell. **172**(14), 1644–1672 (2008)
10. Lang, J., Marquis, P.: Reasoning under inconsistency: a forgetting-based approach. Artif. Intell. **174**(12–13), 799–823 (2010)
11. Wang, Z., Wang, K., Topor, R.W., Pan, J.Z.: Forgetting for knowledge bases in DL-Lite. Ann. Math. Artif. Intell. **58**(1–2), 117–151 (2010)
12. Kontchakov, R., Wolter, F., Zakharyaschev, M.: Logic-based ontology comparison and module extraction, with an application to dl-lite. Artif. Intell. **174**(15), 1093–1141 (2010)
13. Konev, B., Ludwig, M., Walther, D., Wolter, F.: The logical difference for the lightweight description logic EL. J. Artif. Intell. Res. (JAIR) **44**, 633–708 (2012)
14. Konev, B., Lutz, C., Walther, D., Wolter, F.: Model-theoretic inseparability and modularity of description logic ontologies. Artif. Intell. **203**, 66–103 (2013)
15. Lewis, C.I.: A survey of symbolic logic. University of California Press (1918). Republished by Dover (1960)
16. Bledsoe, W.W., Hines, L.M.: Variable elimination and chaining in a resolution-based prover for inequalities. In: Bibel, W., Kowalski, R. (eds.) CADE 1980. LNCS, vol. 87, pp. 70–87. Springer, Heidelberg (1980). doi:10.1007/3-540-10009-1_7
17. Larrosa, J.: Boosting search with variable elimination. In: Dechter, R. (ed.) CP 2000. LNCS, vol. 1894, pp. 291–305. Springer, Heidelberg (2000). doi:10.1007/3-540-45349-0_22
18. Larrosa, J., Morancho, E., Niso, D.: On the practical use of variable elimination in constraint optimization problems: 'still-life' as a case study. J. Artif. Intell. Res. (JAIR) **23**, 421–440 (2005)
19. Middeldorp, A., Okui, S., Ida, T.: Lazy narrowing: strong completeness and eager variable elimination. Theor. Comput. Sci. **167**(1&2), 95–130 (1996)
20. Moinard, Y.: Forgetting literals with varying propositional symbols. J. Log. Comput. **17**(5), 955–982 (2007)

21. Weber, A.: Updating propositional formulas. In: Expert Database Conference, pp. 487–500 (1986)
22. Alferes, J.J., Leite, J.A., Pereira, L.M., Przymusinska, H., Przymusinski, T.C.: Dynamic updates of non-monotonic knowledge bases. J. Logic Program. **45**(1–3), 43–70 (2000)
23. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: On properties of update sequences based on causal rejection. Theor. Pract. Logic Program. (TPLP) **2**(6), 721–777 (2002)
24. Sakama, C., Inoue, K.: An abductive framework for computing knowledge base updates. Theor. Pract. Logic Program. (TPLP) **3**(6), 671–713 (2003)
25. Slota, M., Leite, J.: Robust equivalence models for semantic updates of answer-set programs. In: Brewka, G., Eiter, T., McIlraith, S.A. (eds.) Proceeding of KR, pp. 158–168. AAAI Press (2012)
26. Slota, M., Leite, J.: A unifying perspective on knowledge updates. In: Cerro, L.F., Herzig, A., Mengin, J. (eds.) JELIA 2012. LNCS, vol. 7519, pp. 372–384. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33353-8_29
27. Delgrande, J.P., Schaub, T., Tompits, H., Woltran, S.: A model-theoretic approach to belief change in answer set programming. ACM Trans. Comput. Log. **14**(2), 14 (2013)
28. Slota, M., Leite, J.: The rise and fall of semantic rule updates based on SE-models. TPLP **14**(6), 869–907 (2014)
29. Wong, K.S.: Forgetting in Logic Programs. Ph.D. thesis, The University of New South Wales (2009)
30. Wang, Y., Zhang, Y., Zhou, Y., Zhang, M.: Forgetting in logic programs under strong equivalence. In: Brewka, G., Eiter, T., McIlraith, S.A. (eds.) Proceedings of KR, pp. 643–647. AAAI Press (2012)
31. Wang, Y., Wang, K., Zhang, M.: Forgetting for answer set programs revisited. In: Rossi, F. (ed.) Proceedings of IJCAI, IJCAI/AAAI (2013)
32. Knorr, M., Alferes, J.J.: Preserving strong equivalence while forgetting. In: Fermé, E., Leite, J. (eds.) JELIA 2014. LNCS, vol. 8761, pp. 412–425. Springer, Cham (2014). doi:10.1007/978-3-319-11558-0_29
33. Wang, Y., Zhang, Y., Zhou, Y., Zhang, M.: Knowledge forgetting in answer set programming. J. Artif. Intell. Res. (JAIR) **50**, 31–70 (2014)
34. Delgrande, J.P., Wang, K.: A syntax-independent approach to forgetting in disjunctive logic programs. In: Bonet, B., Koenig, S. (eds.) Proceedings of AAAI, pp. 1482–1488. AAAI Press (2015)
35. Zhang, Y., Zhou, Y.: Knowledge forgetting: properties and applications. Artif. Intell. **173**(16–17), 1525–1537 (2009)
36. Gonçalves, R., Knorr, M., Leite, J.: Forgetting in ASP: the forgotten properties. In: Michael, L., Kakas, A. (eds.) JELIA 2016. LNCS, vol. 10021, pp. 543–550. Springer, Cham (2016). doi:10.1007/978-3-319-48758-8_37
37. Gonçalves, R., Knorr, M., Leite, J.: On some properties of forgetting in ASP. In: Booth, R., Casini, G., Klarman, S., Richard, G., Varzinczak, I.J. (eds.) Proceedings of the International Workshop on Defeasible and Ampliative Reasoning (DARe-16). CEUR Workshop Proceedings, vol. 1626. CEUR-WS.org (2016)
38. Brass, S., Dix, J.: Semantics of (disjunctive) logic programs based on partial evaluation. J. Log. Program. **40**(1), 1–46 (1999)
39. Wong, K.S.: Sound and complete inference rules for SE-consequence. J. Artif. Intell. Res. (JAIR) **31**, 205–216 (2008)
40. Eiter, T., Fink, M., Woltran, S.: Semantical characterizations and complexity of equivalences in answer set programming. ACM Trans. Comput. Log. **8**(3) (2007)

41. Lifschitz, V.: What is answer set programming? In: Fox, D., Gomes, C.P. (eds.) Proceedings of AAAI, pp. 1594–1597. AAAI Press (2008)
42. Sagiv, Y.: Optimizing datalog programs. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 659–698. Morgan Kaufmann (1988)
43. Eiter, T., Fink, M.: Uniform equivalence of logic programs under the stable model semantics. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 224–238. Springer, Heidelberg (2003). doi:10.1007/978-3-540-24599-5_16
44. Lifschitz, V., Tang, L.R., Turner, H.: Nested expressions in logic programs. Ann. Math. Artif. Intell. **25**(3–4), 369–389 (1999)
45. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Trans. Comput. Log. **2**(4), 526–541 (2001)

http://www.springer.com/978-3-319-61659-9

Logic Programming and Nonmonotonic Reasoning
14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings
Balduccini, M.; Janhunen, T. (Eds.)
2017, XIII, 359 p. 41 illus., Softcover
ISBN: 978-3-319-61659-9