The Ultimate Guide to Forgetting in Answer Set Programming

Ricardo Gonçalves and Matthias Knorr and João Leite
NOVA LINCs, Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
2829-516 Caparica, Portugal

Abstract

Many approaches for forgetting in Answer Set Programming (ASP) have been proposed in recent years, in the form of specific operators, or classes of operators, following different principles and obeying different properties. Whereas each approach was developed to somehow address some particular view on forgetting, thus aimed at obeying a specific set of properties deemed adequate for such view, we are lacking a comprehensive and uniform overview of existing operators and properties. We aim at overcoming this by thoroughly examining existing properties and (classes of) operators for forgetting in ASP, drawing a complete picture, which includes many novel (even surprising) results on relations between properties and operators. Our goal is to provide a guide to help users in choosing the most adequate operator for their application requirements.

Introduction

In this paper, we present a systematic study of forgetting in Answer Set Programming (ASP), thoroughly investigating the different approaches found in the literature, their properties and relationships – including many novel results – giving rise to a comprehensive guide aimed at helping users navigate this topic’s complex landscape and ultimately assist them in choosing suitable operators for each application.

Forgetting – or variable elimination – is an operation that allows the removal, from a knowledge base, of middle variables no longer deemed relevant. The importance of forgetting is witnessed by its application to cognitive robotics (Lin and Reiter 1997; Liu and Wen 2011; Rajaratnam et al. 2014), resolving conflicts (Lang, Liberatore, and Marquis 2003; Zhang and Foo 2006; Eiter and Wang 2008; Lang and Marquis 2010), and ontology abstraction and comparison (Wang et al. 2010; Kontchakov, Wolter, and Zakharyaschev 2010; Konev et al. 2012; 2013). With its early roots in Boolean Algebra (Lewis 1918), it has been extensively studied in the context of classical logic (Bledsoe and Hines 1980; Lang, Liberatore, and Marquis 2003; Larrosa 2000; Larrosa, Morano, and Niso 2005; Middeldorp, Okui, and Ida 1996; Moinard 2007; Weber 1986).

Only more recently, the operation of forgetting began to receive attention in the context of logic programming and non-monotonic reasoning, notably of ASP. It turns out that the rule-based nature and non-monotonic semantics of ASP create very unique challenges to the development of forgetting operators – just as to the development of other belief change operators such as those for revision and update, c.f. (Slota and Leite 2014) – 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 (Zhang and Foo 2006; Eiter and Wang 2008; Wong 2009; Wang et al. 2012; Wang, Wang, and Zhang 2013; Knorr and Alferes 2014; Wang et al. 2014; Delgrande and Wang 2015).

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 (Zhang and Zhou 2009; Wang et al. 2012; 2014), others specifically introduced for the case of ASP (Eiter and Wang 2008; Wong 2009; Wang et al. 2012; Wang, Wang, and Zhang 2013; Knorr and Alferes 2014; Delgrande and Wang 2015). 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.

The result 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. Recently, Ji, Wang and You (2015) sought to characterize an operator that would obey all of the properties in the literature. Whereas, in principle, having such an operator would be a good outcome, especially because each property proposed in the literature is prima facie desirable, it turns out that any such operator can only be defined for an extremely limited class of programs, which does not include the standard class of normal logic programs nor just about all the programs used in examples found in the literature on forgetting in ASP.
This strengthens the idea 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. The choice of operator will then depend on which properties are deemed more important for the specific application in hand, for which it is important to understand: a) the relationships between different properties, b) which properties are obeyed by which operators, and even c) whether some sets of properties make more sense than others.

To this end, in this paper, we present a systematic, comprehensive and thorough study of forgetting in ASP, going well beyond a simple survey since many novel results and insights are presented. After a brief section with preliminaries and a section where the concept of forgetting is defined, this paper is divided into four main sections. The first one contains a comprehensive account of the properties found in the literature and a section where the concept of forgetting is defined, as well as a map of the landscape of exist-

### Logic Programs

An (extended) logic program \( P \) is a finite set of (extended) rules, i.e., formulas of the form

\[
\bigwedge \neg \neg D \land \bigwedge \neg \neg C \land \bigwedge B \supset \bigvee A, \quad (2)
\]

where all elements in \( A = \{a_1, \ldots, a_k\}, B = \{b_1, \ldots, b_l\}, C = \{c_1, \ldots, c_m\}, D = \{d_1, \ldots, d_n\} \) are atoms.\(^3\) Such rules \( r \) are also commonly written as

\[
a_1 \lor \ldots \lor a_k \leftarrow b_1, \ldots, b_l, \text{not} \ c_1, \ldots, \text{not} \ c_m, \text{not} \ d_1, \ldots, \text{not} \ d_n, \quad (3)
\]

and we will use both forms interchangeably. Given \( r \), we distinguish its head, \( \text{head}(r) = A \), and its body, \( \text{body}(r) = \bigcup \neg \neg C \cup \neg \neg D \), representing a disjunction and a conjunction.

As shown by Cabalar and Ferraris (2007), any set of (propositional) formulas \( \cal{H} \) is HT-equivalent to an (extended) logic program which is why we can focus solely on these.

This class of logic programs, \( \cal{C}_e \), 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 \( r \) is Horn if also \( l = 0 \). The classes of disjunctive, normal and Horn programs, \( \cal{C}_d, \cal{C}_n, \text{and} \cal{C}_H \), 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 \( \cal{C}_{nd} \), called non-disjunctive programs, different from normal programs.

We have \( \cal{C}_H \subset \cal{C}_n \subset \cal{C}_d \subset \cal{C}_e \) and also \( \cal{C}_n \subset \cal{C}_{nd} \subset \cal{C}_e \).

We now recall the answer set semantics (Gelfond and Lifschitz 1991) for logic programs. Given a program \( P \) and a set \( I \) of atoms, the reduct \( P^I \) is defined as \( P^I = \{ A \leftarrow B : r \text{ of the form } (3) \in P, C \cap I = \emptyset, D \subseteq I \} \). A set \( I' \) of atoms is a model of \( \cal{P} \) if, for each \( r \in P^I \), \( I' \models B \) implies \( I' \models A \). \( I' \) is minimal in a set \( S \), denoted by \( I \models \text{MIN}(S) \), if there is no \( I' \subseteq S \) s.t. \( I' \subseteq I \). Then, \( I \) is an answer set of \( P \) if \( I \) is a minimal model of \( P^I \). Note that, for \( \cal{C}_{nd} \) and its subclasses, this minimal model is in fact unique. The set $^2\models$ is the standard consequence relation from classical logic.

\(^3\)Extended logic programs (Lifschitz, Tang, and Turner 1999) are actually more expressive, but this form is sufficient here.
of all answer sets of \( P \) is denoted by \( \text{AS}(P) \). Note that, for \( C_d \) and its subclasses, all \( I \in \text{AS}(P) \) are pairwise incomparable. If \( P \) has an answer set, then \( P \) is consistent. Also, the \( V \)-exclusion of a set of answer sets \( \mathcal{M} \), denoted \( \mathcal{M}_{\setminus V} \), is \( \{X \setminus V \mid X \in \mathcal{M}\} \). Two programs \( P_1, P_2 \) are equivalent if \( \text{AS}(P_1) = \text{AS}(P_2) \) and strongly equivalent if \( P_1 \equiv_{HT} P_2 \). It is well-known that answer sets and equilibrium models are equivalent (Lifschitz, Pearce, and Valverde 2001), but since the former notion is frequently used in the literature and arguably easier to use, we will mainly rely on it. Finally, determining if program \( P \) has an answer set is \( \Sigma_2^p \)-complete, and \( \text{NP} \)-complete if \( P \) is non-disjunctive (Dantsin et al. 2001).

**Forgetting**

The principal idea of forgetting in logic programming is to remove or hide certain atoms from a given program, while preserving its semantics for the remaining atoms.

**Example 1.** Consider the following program \( P \):

\[
d \leftarrow \neg c \quad a \leftarrow e \quad e \leftarrow b \quad b \leftarrow
\]

The result of forgetting about atom \( e \) from \( P \) should be a program over the remaining atoms of \( P \), i.e., it should not contain \( e \). Intuitively, in the result, the fact \( b \leftarrow \) should persist since it is independent of \( e \). In addition, the link between \( a \) and \( b \) should be preserved in some way, even if \( e \) is absent. Also, \( d \) should still follow from the result of forgetting as the original rule \( d \leftarrow \neg c \) does not contain \( e \).

As the example indicates, preserving the semantics for the remaining atoms is not necessarily tied to one unique program. 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 set of atoms \( V \) to be forgotten, returns a unique program, the result of forgetting about \( V \) from \( P \).

**Definition 1.** Given a class of logic programs \( C \) over \( A \), a forgetting operator is a partial function \( f : C \times 2^A \rightarrow C \) s.t. \( f(P, V) \) is a program over \( A(P) \setminus V \), for each \( P \in C \) and \( V \subseteq 2^A \). We call \( f(P, V) \) the result of forgetting about \( V \) from \( P \). Furthermore, \( f \) is called closed for \( C' \subseteq C \) if, for every \( P \in C' \) and \( V \subseteq 2^A \), we have \( f(P, V) \in C' \). A class \( F \) of forgetting operators is a set of forgetting operators.

Note that the requirement for being a partial function is a natural one given the existing notions in the literature, where some are not closed for certain classes of programs.

To remain as general and uniform as possible, we focus on classes of operators. Whenever a notion of forgetting in the literature is defined through a concrete forgetting operator only, we consider the class containing that single operator.

It is worth noting that some notions of forgetting do not explicitly require that atoms to be forgotten be absent from the result of forgetting, but instead that they be irrelevant:

**IR** \( f(P, V) \equiv_{HT} P' \) for some \( P' \) not containing any \( v \in V \). Although **IR** allows (irrelevant) occurrences of atoms in a result of forgetting, in the literature they are subsequently assumed to be not occurring, which is sanctioned by **IR**. Hence, focusing on operators that yield programs without the atoms to be forgotten is not a restriction in these cases.

**Properties of Forgetting**

Previous work on forgetting in ASP has introduced a variety of desirable properties. In this section, we recall the relevant properties found in the literature and investigate existing relations between them.

Unless otherwise stated, \( F \) is a class of forgetting operators, and \( C \) the class of programs over \( A \) of a given \( f \in F \).

The first three properties were proposed by Eiter and Wang (2008), though not formally introduced as such. The first two were in fact guiding principles for defining their notion of forgetting, while the third was later formalized by Wang et al. (2013).

**(sC)** \( F \) satisfies strengthened Consequence if, for each \( f \in F \), \( P \in C \) and \( V \subseteq A \), we have \( \text{AS}(f(P, V)) \subseteq \text{AS}(P) \cap V \). Strongened Consequence requires that the answer sets of the result of forgetting be answer sets of the original program, ignoring the atoms to be forgotten.

**(we)** \( F \) satisfies weak Equivalence if, for each \( f \in F \), \( P, P' \in C \) and \( V \subseteq A \), we have \( \text{AS}(f(P, V)) = \text{AS}(f(P', V)) \) whenever \( \text{AS}(P) = \text{AS}(P') \).

Weak Equivalence requires that forgetting preserves equivalence of programs (equality of answer sets).

**(SE)** \( F \) satisfies Strong Equivalence if, for each \( f \in F \), \( P, P' \in C \) and \( V \subseteq A \): if \( P \equiv_{HT} P' \), then \( f(P, V) \equiv_{HT} f(P', V) \).

Strong Equivalence requires that forgetting preserves strong equivalence of programs.

The next three properties, together with **IR**, were introduced by Zhang and Zhou (2009) in the context of forgetting in modal logics, and later adopted by Wang et al. (2012; 2014) for forgetting in ASP.

**(W)** \( F \) satisfies Weakening if, for each \( f \in F \), \( P \in C \) and \( V \subseteq A \), we have \( P \models_{HT} f(P, V) \).

Weakening requires that the \( HT \)-models of \( P \) also be \( HT \)-models of \( f(P, V) \), thus implying that \( f(P, V) \) has at most the same consequences as \( P \).

**(PP)** \( F \) satisfies Positive Persistence if, for each \( f \in F \), \( P \in C \) and \( V \subseteq A \): if \( P \equiv_{HT} P' \), with \( P' \in C \) and \( A(P') \subseteq A \setminus V \), then \( f(P, V) \equiv_{HT} f(P', V) \).

Positive Persistence requires that the consequences of \( P \) not containing atoms to be forgotten be preserved in the result of forgetting.

**(NP)** \( F \) satisfies Negative Persistence if, for each \( f \in F \), \( P \in C \) and \( V \subseteq A \): if \( P \not\equiv_{HT} P' \), with \( P' \in C \) and \( A(P') \subseteq A \setminus V \), then \( f(P, V) \not\equiv_{HT} f(P', V) \).

Negative Persistence requires that a program not containing atoms to be forgotten not be a consequence of \( f(P, V) \), unless it was already a consequence of \( P \).

The following property was introduced by Wong (2009), but the more descriptive name is novel here.

**(SI)** \( F \) satisfies Strong (addition) Invariance if, for each \( f \in F \), \( P \in C \) and \( V \subseteq A \), we have \( f(P, V) \cup R \models_{HF} f(P \cup R, V) \) for all programs \( R \in C \) with \( A(R) \subseteq A \setminus V \).

Strong 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. (2012) and formalized by Wang et al. (2013). It requires that a result of forgetting for \( P \) in \( C \) exists in the class \( C \), important to iterate. We extend this property s.t. it be explicitly tied to a class \( C \), thus allowing to speak about \( F \) being closed/not closed for different classes \( C \).

\[ (E_C) \quad F \text{ satisfies existence for } C, \text{ i.e., } F \text{ is closed for a class of programs } C \text{ if there exists } f \in F \text{ s.t. } f \text{ is closed for } C. \]

In the literature, classes of operators are often defined in ways such that only some of its members are closed for a certain class. Thus, class \( F \) being closed for some \( C \) only requires that there exists some “witness in favor of it”, instead of having to restrict the class to the closed operators.

The next property was introduced by Wang et al. (2013) building on the ideas behind (\( SC \)) by Eiter and Wang (2008).

\[ (CP) \quad F \text{ satisfies Consequence Persistence if, for each } f \in F, \text{ } P \in C \text{ and } V \subseteq \mathcal{A}, \text{ we have } \mathcal{A}S(f(P,V)) = \mathcal{A}S(P)\|V. \]

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 (2014) with the aim of imposing the preservation of all dependencies contained in the original program.

\[ (SP) \quad F \text{ satisfies Strong Persistence if, for each } f \in F, \text{ } P \in C \text{ and } V \subseteq \mathcal{A}, \text{ we have } \mathcal{A}S(f(P,V)) \cup \mathcal{A}S(P)\|V, \text{ for all programs } R \in C \text{ with } A(R) \subseteq \mathcal{A}\setminus V. \]

This 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 is due to Delgrande and Wang (2015), although its name is novel as well.

\[ (wC) \quad F \text{ satisfies weakened Consequence if, for each } f \in F, \text{ } P \in C \text{ and } V \subseteq \mathcal{A}, \text{ we have } \mathcal{A}S(f(P,V)) \subseteq \mathcal{A}S(f(P,V)). \]

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, and several relations between them exist. The following proposition establishes all known relevant relations between properties, some novel and some to be found in the literature.

**Proposition 1.** The following relations hold for all \( F \):\(^4\)

1. (CP) is incompatible with (W) as well as with (NP) (for \( F \) closed for \( C \), where \( C \) contains normal logic programs);
2. (W) is equivalent to (NP);
3. (SP) implies (PP);
4. (SP) implies (SE);
5. (W) and (PP) together imply (SE);
6. (CP) and (SI) together are equivalent to (SP);
7. (SC) and (wC) together are equivalent to (CP);
8. (CP) implies (wE);
9. (SE) and (SI) together imply (PP).

Items 1.-4. are known from the literature: 1. in (Wang, Wang, and Zhang 2013), 2.-3. in (Ji, You, and Wang 2015), and 4. in (Knorr and Alferes 2014). The remainder are novel.

Note first, that 1. and 2. also rely on (IR) in their original formulation, in particular, (W) is equivalent to (NP) and

\[ (IR) \quad \text{As (IR) is incorporated directly into our definition of forgetting operators, this reliance is ensured implicitly. This means that, by 2., the four properties proposed by Zhang and Zhou (2009) actually reduce to two distinct ones. In addition, 5. ensures that these two imply (SE), which is considered desirable by Wang et al. (2012) in addition to the former. So, five desired properties can be represented by two.} \]

As indicated by 3. and 4., (SP) seems to be an expressive property, further confirmed by the new result 6. that provides a non-trivial decomposition of (SP) into (CP) and (SI). These two are themselves expressive, as witnessed by other new results. Namely, 7. shows that (CP) is the combination of (SC) and (wC), and 8. that it implies preservation of equivalence, while 9. provides the non-trivial result that Strong Equivalence and Strong Invariance imply Positive Persistence. The latter means that 3. can actually be obtained without relying on (CP). Altogether, (SP) implies all properties in this section, except for (W) and (NP), to which it is incompatible under the condition given in 1., and (EC), where we need to consider concrete classes of programs.

**Operators of Forgetting**

We now turn our attention to operators of forgetting in ASP, reviewing the approaches found in the literature and establishing novel relations between them.

**Strong and Weak Forgetting** The first proposals are due to Zhang and Foo (2006) 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) (Brass and Dix 1999), defined as follows: for a normal logic program \( P \) and \( a \in \mathcal{A} \), \( R(P,a) \) is the set of all rules in \( P \) and all rules of the form \( \text{head}(r_1) \leftarrow \text{body}(r_1) \setminus \{a\} \cup \text{body}(r_2) \) for each \( r_1, r_2 \in P \) s.t. \( a \in \text{body}(r_1) \) and \( \text{head}(r_2) = a \).

Then, the two operators differ on how they subsequently remove rules containing \( a \), the atom to be forgotten. In Strong Forgetting, all rules containing \( a \) are simply removed:

\[ f_{strong}(P,a) = \{r \in R(P,a) \mid a \notin A(r)\} \]

In Weak Forgetting, rules with occurrences of \( not a \) in the body are kept, after \( not a \) is removed.

\[ f_{weak}(P,a) = \{\text{head}(r) \leftarrow \text{body}(r) \setminus \{not a\} \mid r \in R(P,a), a \notin \text{head}(r) \cup \text{body}(r)\} \]

The motivation for this difference is whether such \( not a \) is seen as support for the rule head (Strong) or not (Weak). In both cases, the actual operator for a set of atoms \( V \) is defined by the sequential application of the respective operator to each \( a \in V \). Both operators are closed for \( C_n \). The corresponding singleton classes are defined as follows.

\[ F_{strong} = \{f_{strong}\} \quad F_{weak} = \{f_{weak}\} \]

**Semantic Forgetting** Eiter and Wang (2008) proposed Semantic Forgetting to improve on some of the shortcomings of the two purely syntax-based operators \( f_{strong} \) and \( f_{weak} \).
Semantic Forgetting introduces a class of operators for consistent disjunctive programs\(^5\) defined as follows:
\[
F_{\text{sem}} = \{ f \mid \text{AS}(f(P,V)) = \text{MIZ}(\text{AS}(P)||V) \}
\]
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 \(P\) ignoring \(V\). Three concrete algorithms are presented, two based on semantic considerations and one syntactic. Unlike the former, the latter is not closed for classes\(^6\) \(C_d^n\) and \(C_n^d\), since double negation is required in general.

**Semantic Strong and Weak Forgetting** Wong (2009) 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.\(^7\) First, given a program \(P\) and an atom \(a\), the set of all consequences of \(P\) is defined as \(Cn(P,a) = \{r \mid r \text{ disjunctive}, P \models_{\text{HT}} r, A(r) \subseteq A(P)\}\). We obtain \(P_S(P,a)\) and \(P_W(P,a)\), the results of strongly and weakly forgetting a single atom from \(P\), as follows:

1. Consider \(P_S = Cn(P,a)\).
2. Obtain \(P_2\) by removing from \(P_1\): (i) \(r\) with \(a \in \text{body}(r)\), (ii) \(a\) from the head of each \(r\) with \(a \notin \text{body}(r)\).
3. Given \(P_2\), obtain \(P_S(P,a)\) and \(P_W(P,a)\) by replacing/removing certain rules \(r\) in \(P_2\) as follows:

<table>
<thead>
<tr>
<th>(r) with (a) in body</th>
<th>(r) with (a) in head</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S)remove only (a)</td>
<td>(W)remove only (a)</td>
</tr>
<tr>
<td>(W)remove only (a)</td>
<td>(W)remove only (a)</td>
</tr>
</tbody>
</table>

The generalization to sets of atoms \(V\), i.e., \(P_S(P,V)\) and \(P_W(P,V)\), can be obtained by simply sequentially forgetting each \(a \in V\), yielding the following classes of operators.
\[
\begin{align*}
F_S &= \{ f \mid f(P,V) \equiv_{\text{HT}} P_S(P,V) \} \\
F_W &= \{ f \mid f(P,V) \equiv_{\text{HT}} P_W(P,V) \}
\end{align*}
\]
While steps 2. and 3. are syntactic, different strongly equivalent representations of \(Cn(P,a)\) exist, thus providing different instances. Wong (2009) defined one construction based on inference rules for HT-consequence, closed for \(C_d\).

**HT-Forgetting** Wang et al. (2012; 2014) introduced HT-Forgetting, building on properties introduced by Zhang and Zhou (2009) in the context of modal logics, with the aim of overcoming problems with Wongs notions, namely that semantic forgetting should not be focused on the atoms to be forgotten. The notion is defined abstractly for classes of programs \(C\).
\[
F_{\text{HT}} = \{ f \mid \text{HT}(f(P,V)) = \text{HT}(P)||V \}
\]
A concrete operator is presented (Wang et al. 2014) that is shown to be closed for \(C_a\) and \(C_H\), and it is also shown that no operator exists that is closed for either \(C_d\) or \(C_n\).

**SM-Forgetting** Wang et al. (2013) 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).
\[
F_{\text{SM}} = \{ f \mid \text{HT}(f(P,V)) \text{ is a maximal subset of } \text{HT}(P)||V \}
\]
A concrete operator is provided that, like for \(F_{\text{HT}}\), is shown to be closed for \(C_a\) and \(C_H\). It is also shown that no operator exists that is closed for either \(C_d\) or \(C_n\).

**Strong AS-Forgetting** Knorr and Alferes (2014) introduced Strong AS-Forgetting with the aim of preserving not only the answer sets of \(P\) itself but also those of \(P \cup R\) for any \(R\) over the signature without the atoms to be forgotten. The notion is defined abstractly for classes of programs \(C\).
\[
F_{\text{Sas}} = \{ f \mid \text{AS}(f(P,V) \cup R) = \text{AS}(P \cup R)||V \}
\]
A concrete operator is defined for \(C_{nd}\), but not closed for \(C_n\) and only defined for certain programs with double negation.

**SE-Forgetting** Delgrande and Wang (2015) recently introduced SE-Forgetting based on the idea that forgetting an atom from program \(P\) is characterized by the set of those SE-consequences, i.e., HT-consequences, of \(P\) that do not mention the atoms to be forgotten. The notion is defined for disjunctive programs building on an inference system by Wong (2008) that preserves strong equivalence. Given that \(f_x\) is the consequence relation of this system, \(Cn_{A}(P) = \{ r \in A(P) \mid r \text{ disjunctive, } P \models_{A} r \}\). The class is defined by:
\[
F_{\text{SE}} = \{ f \mid f(P,V) \equiv_{\text{HT}} Cn_{A}(P) \cap A(P)||V \}
\]
An operator is provided, which is closed for \(C_d\).

While all these classes were introduced with differing motivations, they coincide under certain conditions, e.g., when restricted to specific classes of programs.

**Proposition 2.** For all Horn programs \(P\), every \(V \subseteq A(P)\), and all forgetting operators \(f_1, f_2\) in the classes \(F_{\text{strong}}, F_{\text{weak}}, F_S, F_{\text{HT}}, F_{\text{SM}}, F_{\text{Sas}},\) and \(F_{\text{SE}}\), it holds that \(f_1(P,V) \equiv_{\text{HT}} f_2(P,V)\).

**Example 2.** Consider the subset of rules of \(P\) in Ex. 1 that are Horn, \(P' = \{ a \leftarrow e, e \leftarrow b, b \leftarrow \} \). Then, \(f(P', \{ e \})\) for any \(f\) in any of these classes is strongly equivalent to \(a \leftarrow b\) and \(b \leftarrow\). All three known operators in \(F_{\text{sem}}\) actually also satisfy this condition, but the class is not sufficiently restricted to ensure this in general. \(F_W\) completely differs since any operator in \(F_W\) must include \(\leftarrow b\) in its result.

Wang et al. (2012; 2014) additionally show that, for \(C_H\), the result of \(F_{\text{HT}}\) is strongly equivalent to that of classical forgetting. We thus obtain as a corollary that this holds for all classes of forgetting operators mentioned in Prop. 2.

Perhaps surprisingly, two classes of operators coincide.
The properties play a role in characterizing the classes of operators. In fact, a precise characterization of some classes of operators depend on the application at hand, the first and perhaps most important step is to draw an exhaustive picture regarding which properties are obeyed by which classes of operators. This takes us to the central theorem of our paper, illustrated in one easy-to-read table.

**Theorem 2.** All results in Fig. 1 hold.

One first observation is that every class of operators obeys a different set of properties (apart from $F_S$ and $F_{SE}$, which coincide, c.f. Thm. 1). This is a strong indication that these properties play a role in characterizing the classes of operators. In fact, a precise characterization of some classes of operators in terms of the properties they satisfy sometimes exists (Wang et al. 2012; 2014; Delgrande and Wang 2015), although this not the case in general.

### Specific Properties

We now focus on analyzing specific properties and how they relate to the known classes of operators.

Starting with (sC) and (wE), we know, by Prop. 1, that any $F$ that is known to satisfy (CP) also satisfies these two. For the remaining classes, it is worth illustrating why $F_S$, $F_{HT}$, and $F_{SE}$ do not satisfy (sC) by looking at the example where we forget about $a$ from $P = \{a \leftarrow \mathrm{not} a\}$; all three classes require the result to be strongly equivalent to $\bot$, i.e., the forgetting operation introduces a new answer-set. Turning to (wE), it requires that the results of forgetting about $p$ from $P = \{q \leftarrow \mathrm{not} p,q \leftarrow \mathrm{not} q\}$ and from $Q = \{q \leftarrow \}$ have the same answer-sets, while the three classes $F_S$, $F_{HT}$, and $F_{SE}$ require that the results be strongly equivalent to $f(P,p) = \{q \leftarrow \mathrm{not} q\}$ and $f(Q,p) = \{q \leftarrow \}$, which are obviously not equivalent. $F_W$ satisfies both properties; in the previous two examples, $\bot$ must be returned in the former, while $f(P,p)$ includes $q \leftarrow$ in the latter.

The properties (SE), (W), (PP), and (NP) have received more attention in the literature, although focusing more on

---

**Figure 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 $C$ in consideration. White background means a novel result, and gray a previously known one.\(^6\)

<table>
<thead>
<tr>
<th></th>
<th>sC</th>
<th>wE</th>
<th>SE</th>
<th>W</th>
<th>PP</th>
<th>NP</th>
<th>SI</th>
<th>CP</th>
<th>SP</th>
<th>wC</th>
<th>$E_{cH}$</th>
<th>$E_{cH}$</th>
<th>$E_{cH}$</th>
<th>$E_{cH}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_{\text{strong}}$</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$F_{\text{weak}}$</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$F_{\text{S}}$</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>$F_{\text{V}}$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>$F_{\text{HT}}$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>$F_{\text{SM}}$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>$F_{\text{Sas}}$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>$F_{\text{SE}}$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
</tbody>
</table>

the properties not satisfied by previous approaches to motivate
the introduction of a new one. As a result, several novel
positive results were yet to be proved, and are included
in the table. It is perhaps worth pointing out that despite Wang
et al. (2014) having discussed that FS and FW do not sat-
sify (PP), they did so using a counterexample – Ex. 3 in this
paper – that is not really part of the language for which FS
and FW are defined, since it relies on rules with double nega-
tion as missing consequences, hence an unfair argument. For
the language for which they are defined, they satisfy (PP).
Worth illustrating is why FSas does not satisfy (W) (NP):

**Example 4.** With P = {a ← not b, b ← not c}, f(P, b) for
any f ∈ FSas must contain a ← not not c which is not a
consequence of P, hence (W) (and (NP)) is not satisfied.

(SI) has received less attention, yet often this non-trivial
property is satisfied. Wong (2009) showed (SI) for strong
and weak forgetting, but using t-equivalence instead of HT-
equivalence, whose semantics differs. The negative results
for FSE follow by correspondence to FS and, for FSM, from
forgetting about b from P as in Ex. 4: f(P, b) ⊢HT ⊤ for
f ∈ FSM, so adding c ← results precisely in a program con-
taining this fact. If we add c ← before forgetting, then the
HT-models of the result of forgetting, ignoring all occur-
cences of b, correspond precisely to ⟨{c}, {c}, {c}, {a, c}⟩,
and ⟨{a, c}, {a, c}⟩. To preserve the answer sets, only the
last of these three can be considered. Hence, a ← and c ←
or (strongly equivalent rules) occur in the result of forgetting
for any f ∈ FSM, and (SI) does not hold.

The new negative results for (CP) and (SP) can be il-
ustrated with forgetting about b from P = {a ← not b, b ← not a}, i.e., the first two rules of Ex. 3. Since aS(P) =
{a}, the result must have two answer sets {a} and ⊥, which is not possible for disjunctive programs obtained from
operators in FS, FW, and FSE. The same example serves as
counterexample for all negative results of (wC), while positive
results follow for classes satisfying (CP) from Prop. 1. Notably, the counterexample also applies to FSE, thus invalid-
itating Thm. 2 in (Delgrande and Wang 2015), and explain-
ing why all results for (wC) are novel.

The results on (EC) for different classes of programs C
reveal that all classes of operators are closed for CH. Yet,
when admitting negation, each F apart from FSas is closed
for the maximal class of programs considered, but often not
for intermediate ones, with the exception of Fsem and FW.
Interestingly, the two known semantic operators in Fsem are
not closed for CH, while the known syntactic one is, despite
not being closed in general. Note that ‘¬’ was used w.r.t.
to the definitions of each F; the singleton classes Fstrong
and Fweak are precisely defined for normal programs; the
intuition behind minimization embedded in Fsem’s defini-
tion does not combine well with double negation; and, for
FS, FW, and FSE, the consequence relation that is applied
is defined for disjunctive rules.

We omit any complexity results from the table since they
do not contribute to distinguishing classes of operators, as
discussed before.

**Classes of Operators**
The results in Fig. 1 provide us valuable information to com-
pare classes of operators and provide some guidelines re-
garding the choice of a forgetting operator.

The first concern is perhaps the required class of programs
C. If some class of operators is not closed or even not defined
for C, then it is certainly not a good choice. An exception
may be the only known operator for FSas, whose applicabil-
ity condition can be checked in linear time (on the number of
rules), if one accepts that forgetting is not always possible.

With these considerations on (EC) in mind, we can ana-
lyze the remaining properties. Fig. 2 shows a lattice of inclu-
sions between the sets of properties satisfied by each known
class of forgetting operators, adding ⊤ to represent all the
properties (except (EC)) and (IR) as the bottom element.

Fig. 2 makes it apparent that there is one kind of prop-
erty that divides the classes into two groups, namely whether
some kind of preservation of answer sets from the original
program to its result of forgetting is supported or not. The
classes for which this is the case are Fsem, FW, FSM, and
FSas by either satisfying (sC) or (CP). Among them, the two
classes that satisfy (CP) turn out to be separated by only one
property as follows (using semi-formal notation).

$$FSM + (SI) \sim FSas$$

Hence, the previously observed difference between the two
on (SP) is only a consequence of the difference on (SI).

Unlike the former, the two that satisfy (sC), Fsem and FW,
are closed for all the program classes for which they are con-
sidered. For FW, adding (wC) seemingly yields FSas:

$$FW + (wC) \sim FSas$$

However, note that picking an operator of FW and somehow
enforcing (wC) will not provide an operator of FSas. It can
be shown that the effect of forgetting V from P for f ∈ FW
yields a result that replaces all v ∈ V as if they were false,
independently of the actual rules in P. E.g., forgetting about
p from p ← would yield ⊥, which is not aligned with the
original idea of forgetting, i.e., removing all v ∈ V without
affecting other derivations, and adding (wC) would disal-
low such operation. In general, an operator of a superclass
does not necessarily belong to a subclass in the hierarchy of
Fig. 2. This also means that at least one of FW and FSas is

---

7Such equations are in fact not to be read as precise characteri-
zations of classes, but rather a form of visualization of differences.
not precisely characterized by their sets of satisfied properties, and given the intrinsic connection of $F_{Sas}$ to (SP), we conjecture that $F_W$ is the culprit.

Among the four classes that do not support preservation of answer sets, $F_{strong}$ and $F_{weak}$ are closely related due to their similar definition. Both coincide on satisfying (SI), a consequence of their syntactic definition that only manipulates rules containing the atoms to be forgotten, and differ on (W) and (PP), a consequence of the different treatment of negated occurrences of the atoms to be forgotten.

For the third node without this preservation support, $F_S/F_{SE}$, there is also a close proximity to $F_W$ based on their definition, yet, their characterizations differ substantially. Both satisfy (SE) and (PP), but differ on five other properties, which means that in this case the variation in the definition has a much more profound effect on the set of satisfied properties. At the same time, we already know that there is a close relation to the remaining class $F_{HT}$ as witnessed in Prop. 3. This is matched by a close correspondence in terms of satisfied properties.

$$F_S/F_{SE} + (SI) \leadsto F_{HT}$$

Again, (SI) plays a distinguishing role. Notably this clarifies the apparent mismatch of the characterizations for $F_{HT}$ and $F_{SE}$, both claiming that it is precisely given by (IR), (W), (PP), and (NP). This is indeed true for each of them for the maximal class of programs considered, but, intuitively, restricting $F_{HT}$ to $C_d$ cancels (SI).

$F_{HT}$ is also closely connected to $F_{SM}$, as the latter restricts the HT-models of the result s.t. (CP) holds. It turns out that this not only cancels (W) (see 1. of Prop. 1), but also (SI).

**Beyond the Existing Literature**

Whereas the results and discussion so far provide insights into the properties and relations between existing classes of operators, which may help when picking one for a given application, the landscape of forgetting operators may admit additional classes, which will be the focus of this section.

**Subclasses of Programs**

While the results in Thm. 2 for (EC) naturally differentiate between the classes of programs $C$ considered, the remaining properties are stated for the most general class of programs for which the class of operators is defined. This begs the question whether restricting $C$ would affect the results shown in Fig. 1, which we will now address.

We first consider Horn programs, for which we already know that all classes of operators are closed. From Prop. 2, we know that several classes that satisfy different sets of properties in the general case actually coincide. As expected, these classes all satisfy the same set of properties when restricted to Horn programs. The reason can be traced back to the incompatibility result 1. in Prop. 1, only stated for program classes above $C_H$, i.e., it does not apply here.

**Theorem 3.** For Horn programs, the following holds:

- $F_{strong}, F_{weak}, F_S, F_{HT}, F_{SM}, F_{Sas},$ and $F_{SE}$ satisfy (W) and (SP);
- $F_{sem}$ satisfies (CP);
- $F_W$ satisfies (SC), (WE), (SE) and (SI).

Actually, only the minimally necessary properties are mentioned, the remaining can be obtained from Prop. 1. This means, in particular, that the first group of classes satisfies all the presented properties (with (EC) limited to (ECH), of course), and, surprisingly, that reducing to Horn programs does not change the set of properties satisfied by $F_W$.

For normal programs, it turns out that the introduction of negation in the body immediately makes the result coincide with the general one. This is witnessed by the fact that all counterexamples are normal programs, in particular those mentioned in the previous section.

**Additional Classes of Operators**

We now take a step back and look at the bigger picture of the sets of properties beyond those that have been shown to hold for existing classes of operators. Fig. 3 provides a lattice with all possible combinations of properties, taking into consideration the following set of simplifications, that actually also can be understood as indicators as to which properties are important for the comparison.

- Since (SP) is only satisfied by $F_{Sas}$, hence not well-suited for a comparison, we use (SI) and (CP) instead. Also, all the implications of (SI) and (CP) are left implicit.
- Since (W) and (NP) (together with (IR)) are equivalent, we only use (W).
- Whenever (W) and (PP) hold, we omit (SE).
- Whenever (SE) and (SI) hold, we omit (PP).
- Since (SC) is implied by (CP), we only show the former whenever the latter does not hold.
- (WC) requires that the set of answer sets can only ever increase through forgetting, which is at odds with the non-monotonic nature of ASP. (WC) only seems meaningful if combined with (SC), as (CP), to preserve answer sets, which is seconded by the existing classes of forgetting operators. We therefore omit (WC).
- (SC) requires that the answer sets of the result of forgetting from $P$ be a subset of those of $P$. As the choice of the subset is deterministic for a concrete operator, (SC) holds as well. Thus, even though (SC) and (WE) are not formally related, any concrete operator satisfies either both or none, hence we omit (SC).

Accordingly, the graph only shows (SC), (SE), (W), (PP), (SI), and (CP) and each node automatically “inherits” all properties from the nodes below to which it is connected, even if some are not explicitly shown because they are implied according to Prop. 1. The lattice is arranged in levels based on the number of properties that are satisfied, where (CP) includes (SC), and implicit properties are taken into account. Thus, e.g., (SI),(CP) appears one level below $\top$ because of all the properties implied by the combination of the two. Links are made between neighboring levels apart from one case where a dashed line ensures that the link between (W),(SI) and (W),(PP),(SI) is not lost.

Though quite a few nodes in the graph correspond to known classes or special cases for Horn, further classes can be explicitly determined. As an example, consider the concrete forgetting operators for Semantic Forgetting that also satisfy (SE). Based on these, a new class of forgetting operators can be defined, essentially adding a condition to the def-
An example operator is sketched for each one:

- $(\text{CP})$: compute all answer sets, remove all atoms to be forgotten, create a program that represents this set of reduced answer sets;
- $(\text{SE})(\text{CP})$: compute all answer sets, remove all atoms to be forgotten, create a canonical program representing the resulting set of answer sets.

Whether these new classes would be preferable to the already existing ones in certain aspects/applications remains an open question that requires further study.

Providing operators that precisely satisfy only a certain set of properties does not suffice to characterize the class:

- $(\text{W})(\text{SI})$: delete all rules with atoms to be forgotten. This operator matches the properties satisfied by Strong Forgetting, but it clearly does not fit into the class (even if we ignored that we defined $F_{\text{strong}}$ as a singleton), since by the way deletion is applied, the idea of (WGPPE) is lost.

Further operators can be defined in a similar style, providing evidence that certain sets of properties are probably of little interest as they are satisfied by absurd operators. We mention those pointed out in Fig. 3:

- $(\text{IR})$: delete all rules; then add some arbitrary rules over the remaining alphabet (after forgetting);\(^{10}\)
- $(\text{W})$: delete all rules with atoms to be forgotten and an arbitrary 50% of the remaining rules;
- $(\text{PP})$: perform weak forgetting; in the resulting program, pick an arbitrary set of rules and turn them into facts by removing their body;
- $(\text{SE})$: compute all answer sets; remove all atoms to be forgotten from them; create a set of facts that represents the intersection of all such reduced answer sets;
- $(\text{SI})$: perform the WGPPE replacement step of strong and weak forgetting: in the resulting program, arbitrarily delete rules with negative occurrences (in the body) of atoms to be forgotten, or just remove their bodies, and delete all rules with positive occurrences (in the body or head) of atoms to be forgotten;
- $(\text{SE})(\text{W})$: delete the entire program;
- $(\text{SE})(\text{PP})$: add, as facts, to the result of performing SE-forgetting, the atoms that belong to some answer-set of the original program and for which there is some rule of the original program that contains some negative and no positive occurrence of forgotten atoms.

Other sets of properties, unannotated in Fig. 3, do not seem to permit the definition of such simplistic operators, so their interest as a class requires further study.

**Conclusions**

The landscape of forgetting in ASP comprises many operators and classes of operators defined to obey some subset of a large number of desirable properties proposed in the literature, while lacking a systematic account of all these results and their relations, making it all too difficult to get a clear understanding of the state-of-the-art, and even to choose the most adequate operator for some specific application.

This paper aimed at addressing this problem, presenting a systematic study of forgetting in ASP, including a thorough investigation of both properties and existing classes of forgetting operators, going well beyond a survey of the state-of-the-art as many novel results were included, thus achieving a truly comprehensive picture of the landscape. The wider picture of yet unexplored terrain was also approached, revealing new candidates for classes of operators as well as sets of properties whose apparent unreasonableness advises against further consideration.

One reasonable way to move forward is to investigate forgetting with semantics other than ASP, such as (Wang et al. 2014) based on the FLP-semantics (Truszczyński 2010), or (Alferes, Knorr, and Wang 2013; Knorr and Alferes 2014) based on the well-founded semantics (Gelder, Ross, and Schlipf 1991).

\(^{10}\)The term “arbitrary” is used freely to represent some deterministic set, e.g., the first elements of some specific ordering.
We would like to thank the reviewers for their comments, which helped improve this paper. R. Gonçalves, M. Knorr and J. Leite were partially supported by FCT under strategic project NOVA LINCS (PEst/UID/CEC/04516/2013). R. Gonçalves was partially supported by FCT grant SFRH/BPD/100906/2014 and M. Knorr was partially supported by FCT grant SFRH/BPD/86970/2012.

Acknowledgments

References


