# A Syntax-Independent Approach to Forgetting in Disjunctive Logic Programs

**James P. Delgrande**
School of Computing Science
Simon Fraser University
Burnaby, B.C. V5A 1S6
Canada
jim@cs.sfu.ca

**Kewen Wang**
School of Information and Communication Technology
Griffith University,
Brisbane, QLD 4111
Australia
k.wang@griffith.edu.au

## Abstract

In this paper, we present an approach to forgetting in disjunctive logic programs, where forgetting an atom from a program amounts to a reduction in the signature of that program. Notably, the approach is syntax-independent, so that if two programs are strongly equivalent, then the result of forgetting a given atom in each program is also strongly equivalent. Our central definition of forgetting is abstract: forgetting an atom from program $P$ is characterised by the set of those SE consequences of $P$ that do not mention the atom to be forgotten. We provide an equivalent, syntactic, characterization in which forgetting an atom $p$ is given by those rules in the program that do not mention $p$, together with rules obtained by a single inference step from those rules that do mention $p$. Forgetting is shown to have appropriate properties; in particular, answer sets are preserved in forgetting an atom. As well, forgetting an atom via the syntactic characterization results in a modest (at worst quadratic) blowup in the program size. Finally, we provide a prototype implementation of this approach to forgetting.

## Introduction

*Forgetting* is an operation for eliminating variables from a knowledge base (Lin and Reiter 1994; Lang, Liberatore, and Marquis 2003). It constitutes a reduction in an agent's language or, more accurately, the agent's signature. It has also been studied under different names, such as variable elimination, uniform interpolation and relevance (Subramanian, Greiner, and Pearl 1997). Forgetting has various possible applications in a reasoning system. For example, in query answering, if one can determine what is relevant to a query, then forgetting the irrelevant part of a knowledge base may yield a more efficient operation. Forgetting may also provide a formal account and justification of predicate hiding, for example for privacy issues. As well, forgetting may be useful in summarising a knowledge base, reusing part of a knowledge base or clarifying relations between predicates.

The best-known definition of forgetting is with respect to classical propositional logic, and is due to George Boole (Boole 1854). To forget an atom $p$ from a formula $\phi$ in propositional logic, one disjoins the result of uniformly substituting $\top$ for $p$ in $\phi$ with the result of substituting $\bot$; that is, forgetting is given by $\phi[p/\top] \vee \phi[p/\bot]$. (Lin and Reiter 1994) investigated the theory of forgetting for first order logic and its application in reasoning about action. Forgetting has been applied in various settings, among them resolving conflicts (Eiter and Wang 2008; Zhang and Foo 1997), and ontology comparison and reuse (Kontchakov, Wolter, and Zakharyaschev 2008; Konev et al. 2013).

In recent years, answer set programming (ASP) (Gelfond and Lifschitz 1988; Baral 2003; Gebser et al. 2012) has become prominent as a knowledge representation language within the logic programming paradigm. However, given the nonmonotonic foundation of ASP, the Boole definition for forgetting does not extend readily to logic programs. In the past few years, several approaches have been proposed for forgetting in ASP (Eiter and Wang 2006; 2008; Wang, Sattar, and Su 2005; Zhang, Foo, and Wang 2005; Zhang and Foo 2006). These approaches are generally syntactic, and the result of forgetting may differ between programs that are strongly equivalent.[1] For example, in (Zhang, Foo, and Wang 2005; Zhang and Foo 2006) forgetting is defined in terms of program transformations, and is not based on answer set semantics or SE models. A semantic theory of forgetting for normal logic programs under answer set semantics is introduced in (Wang, Sattar, and Su 2005), in which a sound and complete algorithm is developed based on a series of program transformations; this theory is further developed and extended to disjunctive logic programs in (Eiter and Wang 2006; 2008). However, this theory of forgetting is defined in terms of answer sets rather than SE models, and so again is not syntax-independent.

In order to use forgetting in its full generality, for dealing with relevance or predicate hiding, or in composing, decomposing, and reusing answer set programs, it is desirable for a definition to be given in terms of the *logical content* of a program, that is in terms of SE models. For example, the reuse of knowledge bases requires that when a subprogram $Q$ in a large program $P$ is substituted with an equivalent program $Q'$, the resulting program should be equivalent to $P$. This is not the case where equivalence is defined in terms of preserving answer sets, due to the nonmonotonic character of ASP. As a result, two definitions of for-



[1] See the next section for definitions.

getting have been introduced in HT-logic (Wang et al. 2012; Wang, Wang, and Zhang 2013). These approaches indirectly establish theories of forgetting under SE models, as HT-logic provides a natural extension of SE models. The approach to interpolation for equilibrium logic introduced in (Gabbay, Pearce, and Valverde 2011) is more general than forgetting. However, the issue of directly establishing a theory of forgetting for disjunctive logic programs under SE models is still missing, and the issue of developing efficient algorithms for computing the result of forgetting under SE models is not addressed.

A key intuition regarding forgetting is that the logical consequences of a set of formulas that do not mention forgotten symbols should still be believed after forgetting. This leads to a very simple (abstract) knowledge-level definition in terms of a consequence operator in the underlying logic: forgetting a symbol from a knowledge base is characterised by the set of consequences that do not mention that symbol. In this paper, we establish such a theory of forgetting for disjunctive logic programs that preserves strong equivalence.

In the next section we give a brief introduction to ASP, with emphasis on the notion of *SE models* and on the notion of *SE consequence*. The following section gives a high-level abstract characterisation of forgetting in ASP, and shows that it has the appropriate properties (for example, that it is indeed syntax-independent, and that forgetting a set of atoms is independent of the order of forgetting the individual atoms). We then give an equivalent syntactic characterisation of forgetting, and we show that forgetting an atom results in at worst a quadratic blowup in the size of the knowledge base. This also immediately leads to an algorithm for computing forgetting under SE models. We investigate some optimisation techniques for the algorithm and report a prototype implementation of the algorithm. Last, we compare our approach to related work, and briefly conclude.

## Answer Set Programming

Here we briefly review pertinent concepts in answer set programming; for details see (Gelfond and Lifschitz 1988; Baral 2003; Gebser et al. 2012).

Let $\mathcal{A}$ be an alphabet, consisting of a set of *atoms*. A (*disjunctive*) *logic program* over $\mathcal{A}$ is a finite set of rules of the form

$$a_1; \ldots; a_m \leftarrow b_1, \ldots, b_n, {\sim}c_1, \cdots, {\sim}c_p. \qquad (1)$$

where $a_i, b_j, c_k \in \mathcal{A}$, and $m, n, p \geq 0$ and $m + n + p > 0$. Binary operators ';' and ',' express disjunction and conjunction respectively. For atom $a$, ${\sim}a$ is (default) negation. $\mathcal{L}_\mathcal{A}$ denotes the language (viz. set of rules) generated by $\mathcal{A}$.

The *head* and *body* of a rule as in (1), $H(r)$ and $B(r)$, are defined by:

$$\begin{array}{rcl} H(r) & = & \{a_1, \ldots, a_m\} \quad \text{and} \\ B(r) & = & \{b_1, \ldots, b_n, {\sim}c_1, \ldots, {\sim}c_p\}. \end{array}$$

Given a set $X$ of literals, we define

$$\begin{array}{rcl} X^+ & = & \{a \in \mathcal{A} \mid a \in X\}, \\ X^- & = & \{a \in \mathcal{A} \mid {\sim}a \in X\}, \text{ and} \\ {\sim}X & = & \{{\sim}a \mid a \in X \cap \mathcal{A}\}. \end{array}$$

For simplicity, we sometimes use a set-based notation, expressing a rule as in (1) as

$$H(r) \leftarrow B(r)^+, {\sim}B(r)^- .$$

The *reduct* of a program $P$ with respect to a set of atoms $Y$, denoted $P^Y$, is the set of rules:

$$\{H(r) \leftarrow B(r)^+ \mid r \in P,\ B(r)^- \cap Y = \emptyset\}.$$

Note that the reduct consists of negation-free rules only. An *answer set* $Y$ of a program $P$ is a subset-minimal model of $P^Y$. A program induces 0, 1, or more *answer sets*. The set of all answer sets of a program $P$ is denoted by $AS(P)$. For example, the program $P = \{a \leftarrow, \quad c; d \leftarrow a, {\sim}b\}$ has answer sets $AS(P) = \{\{a, c\}, \{a, d\}\}$. Notably, a program is nonmonotonic with respect to its answer sets. For example, the program $\{q \leftarrow {\sim}p\}$ has answer set $\{q\}$ while $\{q \leftarrow {\sim}p, \ \ p \leftarrow\}$ has answer set $\{p\}$.

### SE Models

As defined by (Turner 2003), an *SE interpretation* is a pair $(X, Y)$ of interpretations such that $X \subseteq Y \subseteq \mathcal{A}$. An SE interpretation is an *SE model* of a program $P$ if $Y \models P$ and $X \models P^Y$, where $\models$ is the relation of logical entailment in classical logic. The set of all SE models of a program $P$ is denoted by $SE(P)$; this notation extends in the obvious fashion to the set of SE models for a given alphabet, viz. $SE(\mathcal{A})$. Then, $Y$ is an answer set of $P$ iff $(Y, Y) \in SE(P)$ and no $(X, Y) \in SE(P)$ with $X \subset Y$ exists.

Program $P$ is *satisfiable* just if $SE(P) \neq \emptyset$.[2] Thus, we consider $P = \{p \leftarrow {\sim}p\}$ to be satisfiable, since $SE(P) \neq \emptyset$ even though $AS(P) = \emptyset$. Programs $P$ and $Q$ are *strongly equivalent*, symbolically $P \equiv_s Q$, iff $SE(P) = SE(Q)$. Alternatively, $P \equiv_s Q$ holds iff $AS(P \cup R) = AS(Q \cup R)$, for every program $R$ (Lifschitz, Pearce, and Valverde 2001). We also write $P \models_s Q$ iff $SE(P) \subseteq SE(Q)$.

### SE Consequence

While the notion of SE models puts ASP on a monotonic footing with respect to model theory, (Wong 2008) has subsequently provided an inferential system for rules that preserves strong equivalence. His notion of *SE consequence* is shown to be sound and complete with respect to the semantic notion of *SE models*. His inference system is given as follows, where lower case letters are atoms and upper case are sets of atoms.

**Inference Rules for SE Consequence**:

**Taut** $x \leftarrow x$

**Contra** $\leftarrow x, {\sim}x$

**Nonmin** From $A \leftarrow B, {\sim}C$ infer
$A; X \leftarrow B, Y, {\sim}C, {\sim}Z$

**WGPPE** From $A_1 \leftarrow B_1, x, {\sim}C_1$ and
$A_2; x \leftarrow B_2, {\sim}C_2$ infer
$A_1; A_2 \leftarrow B_1, B_2, {\sim}C_1, {\sim}C_2$

[2] Note that many authors define satisfiability in terms of answer sets, in that for them a program is satisfiable if it has an answer set, i.e., $AS(P) \neq \emptyset$.

**S-HYP** From $A_1 \leftarrow B_1, \sim x_1, \sim C_1,$
$\ldots,$
$A_n \leftarrow B_n, \sim x_n, \sim C_n,$
$A \leftarrow x_1, \ldots, x_n, \sim C$ infer
$A_1; \ldots; A_n \leftarrow$
$B_1, \ldots, B_n, \sim C_1, \ldots, \sim C_n, \sim A, \sim C$

Several of these rules are analogous to or similar to well-known rules in the literature. For example, **Nonmin** is *weakening*; **WGPPE** is analogous to *cut*; and **S-HYP** is a version of *hyper-resolution*. Let $\vdash_s$ denote the consequence relation generated by these rules, for convenience allowing sets of rules on the right hand side of $\vdash_s$. Then $P \leftrightarrow_s P'$ abbreviates $P \vdash_s P'$ and $P' \vdash_s P$. As well, define

$$\mathcal{C}n_{\mathcal{A}}(P) = \{r \in \mathcal{L}_{\mathcal{A}} \mid P \vdash_s r\}.$$

Then the above set of inference rules is sound and complete with respect to the entailment $\models_s$.

**Theorem 1 ((Wong 2008))** *$P \models_s r$ iff $P \vdash_s r$.*

# The Approach

## Formal Preliminaries

Since forgetting in our approach amounts to decreasing the alphabet, or signature, of a logic program, we need additional notation for relating signatures. Let $\mathcal{A}$ and $\mathcal{A}'$ be two signatures where $\mathcal{A} \subset \mathcal{A}'$. Then $\mathcal{A}$ is a *reduction*[3] of $\mathcal{A}'$, and $\mathcal{A}'$ is an *expansion* of $\mathcal{A}$. Furthermore, if $w \in SE(\mathcal{A})$ and $w' \in SE(\mathcal{A}')$ where $w$ and $w'$ agree on the interpretation of symbols in $\mathcal{A}$ then $w$ is the *$\mathcal{A}$-reduction* of $w'$, and $w'$ is an *$\mathcal{A}'$-expansion* of $w$. For fixed $\mathcal{A} \subset \mathcal{A}'$, reductions are clearly unique whereas expansions are not.

For a logic program $P$, $\sigma(P)$ denotes the signature of $P$, that is, the set of atoms mentioned in $P$. SE models are defined with respect to an understood alphabet; for SE model $w$ we also use $\sigma(w)$ to refer to this alphabet. Thus for example if $\mathcal{A} = \{a, b, c\}$ then, with respect to $\mathcal{A}$, the SE model $w = (\{a\}, \{a, b\})$ is more perspicuously written as $(\{a, \neg b, \neg c\}, \{a, b, \neg c\})$, and so in this case $\sigma(w) = \{a, b, c\}$.

If $\mathcal{A} \subset \mathcal{A}'$ and for SE models $w$, $w'$ we have $\sigma(w) = \mathcal{A}$ and $\sigma(w') = \mathcal{A}'$ then we use $w'_{|\mathcal{A}}$ to denote the reduction of $w'$ with respect to $\mathcal{A}$ and we use $w_{\uparrow \mathcal{A}'}$ to denote the set of expansions of $w$ with respect to $\mathcal{A}'$. This notation extends to sets of models in the obvious way. As well, we use the notion of a reduction for logic programs; that is, for $\mathcal{A} \subseteq \mathcal{A}'$,

$$P_{|\mathcal{A}} = \{r \in P \mid \sigma(r) \subseteq \mathcal{A}\}.$$

## An Abstract Characterisation of Forgetting

As described, our goal is to define forgetting with respect to the logical content of a logic program. For example, if we were to forget $b$ from the program $\{a \leftarrow b.,\ b \leftarrow c.\}$, we would expect the rule $a \leftarrow c$ to be in the result, since it is implicit in the original program. Consequently, our primary definition is the following.

[3]The standard term in model theory is *reduct* (Chang and Keisler 2012; Doets 1996; Hodges 1997). However *reduct* has its own meaning in ASP, and so we adopt this variation.

**Definition 1** *Let $P$ be a disjunctive logic program over signature $\mathcal{A}$. The result of* forgetting *$\mathcal{A}'$ in $P$, denoted $Forget(P, \mathcal{A}')$, is given by:*

$$Forget(P, \mathcal{A}') = \mathcal{C}n_{\mathcal{A}}(P) \cap \mathcal{L}_{\mathcal{A} \setminus \mathcal{A}'}.$$

That is, the result of forgetting a set of atoms $\mathcal{A}'$ in program $P$ is simply the set of SE consequences of $P$ over the original alphabet, but excluding atoms from $\mathcal{A}'$. If $\mathcal{A}'$ is a singleton, say $\{p\}$, then we sometimes drop the set braces and write $Forget(P, p)$.

The above concept of forgetting is defined at the *knowledge level*. So, our definition is abstract, but is simple and intuitive. As a consequence, many formal results are very easy to show. On the other hand, the definition is not immediately practically useful, since forgetting results in an infinite set of rules. Consequently a key question is to determine a finite characterisation (that is to say, a uniform interpolant) of $Forget$. We explore these issues next.

The following results are elementary, but show that the definition of forgetting has the "right" properties.

**Proposition 1** *Let $P$ and $P'$ be disjunctive logic program and let $\mathcal{A}$ (possibly primed or subscripted) be alphabets.*

1. $P \vdash_s Forget(P, \mathcal{A})$
2. *If* $P \leftrightarrow_s P'$ *then* $Forget(P, \mathcal{A}) \leftrightarrow_s Forget(P', \mathcal{A})$
3. $Forget(P, \mathcal{A}) = \mathcal{C}n_{\mathcal{A}'}(Forget(P, \mathcal{A}))$
   *where* $\mathcal{A}' = \sigma(P) \setminus \mathcal{A}$.
4. $Forget(P, \mathcal{A}) =$
   $Forget(Forget(P, \mathcal{A} \setminus \{a\}), \{a\})$ *where* $a \in \mathcal{A}$
5. $Forget(P, \mathcal{A}_1 \cup \mathcal{A}_2) = Forget(Forget(P, \mathcal{A}_1), \mathcal{A}_2))$
6. $P$ *is a* conservative extension *of* $Forget(P, \mathcal{A})$

Thus, Part 1 asserts that forgetting results in no consequences not in the original theory. As well, the result of forgetting is independent of syntax and yields a deductively-closed theory (Parts 2 and 3). Part 4 gives an iterative means of determining forgetting on an element-by-element basis. The next part, which generalises the previous, shows that forgetting is decomposable with respect to a signature, which in turn implies that forgetting is a commutative operation with respect to its second argument. Last, $P$ is a conservative extension of the result of forgetting, which is to say that $\sigma(P) \setminus \mathcal{A} \subseteq \sigma(P)$ and the consequences of $P$ and $Forget(P, \mathcal{A})$ coincide over the language $\mathcal{L}_{\sigma(P) \setminus \mathcal{A}}$.

With regards to SE models, we obtain the following results giving an alternative characterisation of forgetting. Here only we use the notation $SE_{\mathcal{A}}(P)$ to indicate the SE models of program $P$ over alphabet $\mathcal{A}$.

**Proposition 2** *Let $\mathcal{A}' \subseteq \mathcal{A}$, and let $\sigma(P) \subseteq \mathcal{A}$.*

1. $SE_{\mathcal{A} \setminus \mathcal{A}'}(Forget(P, \mathcal{A}')) = SE_{\mathcal{A}}(P)_{|(\mathcal{A} \setminus \mathcal{A}')}$
2. $SE_{\mathcal{A}}(Forget(P, \mathcal{A}')) = (SE_{\mathcal{A}}(P)_{|(\mathcal{A} \setminus \mathcal{A}')})_{\uparrow \mathcal{A}}$

The first part provides a semantic characterisation of forgetting: the SE models of $Forget(P, \mathcal{A}')$ are exactly the SE models of $P$ restricted to the signature $\mathcal{A} \setminus \mathcal{A}'$. Informally, what this means is that the SE models of $Forget(P, \mathcal{A}')$ can be determined by simply dropping the symbols in $\mathcal{A}'$ from the SE models of $P$. The second part, which is a simple

corollary of the first, expresses forgetting with respect to the original signature.

Of course, one may wish to re-express the effect of forgetting in the original language of $P$; in fact, many approaches to forgetting assume that the underlying language is unchanged. To this end, we can consider a variant of Definition 1 as follows, where $\mathcal{A}' \subseteq \mathcal{A}$.

$$Forget_{\mathcal{A}}(P, \mathcal{A}') = \mathcal{C}n_{\mathcal{A}}(Forget(P, \mathcal{A}')) \qquad (2)$$

That is, $Forget(P, \mathcal{A}')$ is re-expressed in the original language with signature $\mathcal{A}$. The result is a theory over the original language, but where the resulting theory carries no contingent information about the domain of application regarding elements of $\mathcal{A}'$.

The following definition is useful in stating results concerning forgetting.

**Definition 2** *Signature $\mathcal{A}$ is* irrelevant *to $P$, $IR(P, \mathcal{A})$, iff there is $P'$ such that $P \leftrightarrow_s P'$ and $\sigma(P') \cap \mathcal{A} = \emptyset$.*

Zhang and Zhou (2009) give four postulates characterising their approach to forgetting in the modal logic S5. An analogous result follows here with respect to forgetting re-expressed in the original signature:

**Proposition 3** *Let $\mathcal{A}' \subseteq \mathcal{A}$ and let $\sigma(P)$, $\sigma(P') \subseteq \mathcal{A}$.*
*Then $P' = Forget_{\mathcal{A}}(P, \mathcal{A}')$ iff*

1. $P \vdash_s P'$
2. *If $IR(r, \mathcal{A}')$ and $P \vdash_s r$ then $P' \vdash_s r$*
3. *If $IR(r, \mathcal{A}')$ and $P \nvdash_s r$ then $P' \nvdash_s r$*
4. $IR(P', \mathcal{A}')$

Hence, if a rule $r$ is independent of a signature $\mathcal{A}'$, then forgetting $\mathcal{A}'$ has no effect on whether that formula is a consequence of the original knowledge base or not (Parts 2 and 3). Part 4 is a "success" postulate: the result of forgetting $\mathcal{A}'$ yields a theory expressible without $\mathcal{A}'$.

The above results refer to general properties for forgetting, though with respect to a disjunctive logic program. The following result is specific to disjunctive programs:

**Theorem 2** *Let $P$ be a disjunctive logic program, let $\mathcal{A}$ be a set of atoms, and let $X$ be an answer set for $P$.*
*Then $X \setminus \mathcal{A}$ is an answer set for $Forget(P, \mathcal{A})$.*

Thus, forgetting in disjunctive programs preserves answer sets. Some comments on this result are in order: SE models of general logic programs satisfy the constraint that if $(X, Y) \in SE(P)$ then $(Y, Y) \in SE(P)$. Disjunctive logic programs satisfy the additional constraint of being *complete*, that is, if for each $(X, Y) \in SE(P)$, then also $(X, Z) \in SE(P)$ for any $Z \supseteq Y$ where $(Z, Z) \in SE(P)$ (Eiter et al. 2004). The notion of completeness is essential for the preservation of answer sets. For example, the set of SE models $S = \{(\emptyset, a), (a, a), (ab, ab)\}$ characterises some general logic program with answer set $\{a, b\}$. Forgetting $b$ results in a general program with SE models $(\emptyset, a)$, $(a, a)$, which has no answer sets. However $S$ is not complete. The least set of SE models that is complete and contains $S$ is $S' = S \cup \{(\emptyset, ab), (a, ab)\}$. The disjunctive logic program corresponding to $S'$ has no answer sets, so forgetting $b$ in that program trivially preserves answer sets.

## A Finite Characterisation of Forgetting

Forgetting in propositional logic can be computed using resolution (see, e.g. (Delgrande 2014)), in part by finding all resolvents on an atom to be forgotten. This is an arguably convenient means of computing forgetting, in that it is easily implementable, and one remains with a set of clauses if the original theory is given as a set of propositional clauses. We can use a similar strategy for computing forgetting in a disjunctive logic program. In particular, for forgetting an atom $a$, we can use the inference rules from (Wong 2008) to compute "resolvents" of rules mentioning $a$ such that the derived rules do not mention $a$.

In the definition below, $ResLP$ is analogous to using resolution for forgetting in propositional logic. In this case, we consider instances of **WGPPE** and **S-HYP** that, from rules mentioning an atom $a$ to be forgotten, can be used to derive rules that do not mention $a$; these instances are given by the two parts of the union, respectively, below.

**Definition 3** *Let $P$ be a disjunctive logic program and let $a \in \mathcal{A}$.*
*Define:*

$$\begin{aligned}
&ResLP(P, a) = \\
&\quad \{r \mid \exists r_1, r_2 \in P \text{ such that} \\
&\qquad r_1 = A_1 \leftarrow B_1, a, {\sim}C_1, \\
&\qquad r_2 = A_2; a \leftarrow B_2, {\sim}C_2, \\
&\qquad r = A_1; A_2 \leftarrow B_1, B_2, {\sim}C_1, {\sim}C_2 \ \} \\
&\quad \cup \\
&\quad \{r \mid \exists r_1, \ldots, r_n, r' \in P \text{ such that } a = a_1 \\
&\qquad r_i = A_i \leftarrow B_i, {\sim}a_i, {\sim}C_i, \quad 1 \leq i \leq n \\
&\qquad r' = A \leftarrow a_1, \ldots a_n, {\sim}C \quad \textit{and} \\
&\qquad r = A_1; \ldots; A_n \leftarrow \\
&\qquad\qquad B_1, \ldots, B_n, {\sim}C_1, \ldots, {\sim}C_n, {\sim}A, {\sim}C \ \}
\end{aligned}$$

We obtain that the result of forgetting an atom $a$ in program $P$ is strongly equivalent to the program consisting of those rules in $P$ that don't mention $a$, together with rules derived according to $ResLP$.

**Theorem 3** *Let $P$ be a disjunctive logic program over $\mathcal{A}$ and $a \in \mathcal{A}$. Assume that any rule $r \in P$ is satisfiable, non-tautologous, and contains no redundant occurrences of any atom. Then:*

$$Forget(P, a) \ \leftrightarrow_s \ P_{|(\mathcal{A} \setminus \{a\})} \ \cup \ ResLP(P, a).$$

**Proof Outline:** From Definition 1, $Forget(P, a)$ is defined to be the set of those SE consequences of program $P$ that do not mention $a$. Thus for disjunctive rule $r$, $r \in Forget(P, a)$ means that $P \vdash_s r$ and $a \notin \sigma(r)$. Thus the left-to-right direction is immediate: Any $r \in P_{|(\mathcal{A} \setminus \{a\})}$ or $r \in ResLP(P, a)$ is a SE consequence of $P$ that does not mention $a$.

For the other direction, assume that we have a proof of $r$ from $P$, represented as a sequence of rules. If no rule in the proof mentions $a$, then we are done. Otherwise, since $r$ does not mention $a$, there is a last rule in the proof, call it $r_n$ that does not mention $a$, but is obtained from rules that do mention $a$. The case where $r_n$ is obtained via **Taut**, **Contra**,

or **Nonmin** is easily handled. If $r_n$ is obtained via **WGPPE** or **S-HYP** then there are rules $r_k$ and $r_l$ that mention $a$ (and perhaps other rules in the case of **S-HYP**). If $r_k$,$r_l \in P$ then $r_n \in ResLP(P, a)$. If one of $r_k$, $r_l$ is not in $P$ (say, $r_k$) then there are several cases, but in each case it can be shown that the proof can be transformed to another proof where the index of $r_k$ in the proof sequence is decreased and the index of no rule mentioning $a$ is increased. This process must terminate (since a proof is a finite sequence), where the premisses of the proof are either rules of $P$ that do not mention $a$, elements of $ResLP(P, a)$, or tautologies.

Consider the following case, where $r_n = A_1; A_2; A_3 \leftarrow B_1, B_2, B_3$, and we use the notation that each $A_i$ is a set of implicitly-disjoined atoms while each $B_i$ is a set of implicitly-conjoined literals. Assume that $r_n$ is obtained by an application of **WGPPE** from $r_k = a; A_1; A_2 \leftarrow B_1, B_2$ and $r_l = A_3 \leftarrow a, B_3$. Assume further that $r_k$ is obtained from $r_i = a; b; A_1 \leftarrow B_1$ and $r_j = A_2 \leftarrow b, B_2$ by an application of **WGPPE**. This situation is illustrated in Figure 1a.

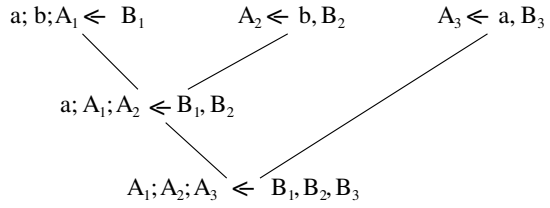


Figure 1a

Then essentially the steps involving the two applications of **WGPPE** can be "swapped", as illustrated in Figure 1b, where $r_k$ is replaced by $r'_k = b; A_1; A_3 \leftarrow B_1, B_3$.

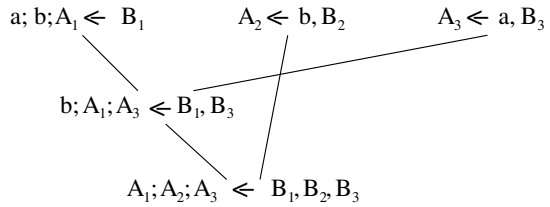


Figure 1b

Thus the step involving $a$ is informally "moved up" in the proof. There are 12 other cases, involving various combinations of the inference rules, but all proceed the same as in the above. □

The theorem is expressed in terms of forgetting a single atom. Via Proposition 1.4 this readily extends to forgetting a set of atoms. Moreover, since we inherit the results of Propositions 1 and 3, we get that the result of forgetting is independent of syntax, even though the expression on the right hand side of Theorem 3 is a set of rules obtained by transforming and selecting rules in $P$. It can also be observed that forgetting an atom results in at worst a quadratic blowup in the size of the program. This blowup for forgetting comes from two sources (i.e., WGPPE and S-HYP). This can also be seen from Steps 3 and 4 in Algorithm 1 in the next section. While this may seem comparatively modest, it implies that forgetting a set of atoms may result in an exponential blowup.

**Example 1** *Let* $P = \{p \leftarrow \sim q, \quad r \leftarrow p\}$. *Forgetting* $p$ *yields* $\{r \leftarrow \sim q\}$ *(where* $r \leftarrow \sim q$ *is obtained by an application of* **WGPPE***), while forgetting* $q$ *and* $r$ *yield programs* $\{r \leftarrow p\}$ *and* $\{p \leftarrow \sim q\}$ *respectively.*

## Computation of Forgetting

Using Theorem 3, we obtain the following algorithm for computing the result of forgetting. A rule $r$ is a *tautology* if it is of the form $r = A; b \leftarrow b, B, \sim C$. A rule $r$ is *contradictory* if it is of the form $r = A; c \leftarrow B, \sim c, \sim C$. A rule $r$ is *minimal* if there is no rule $r'$ in $P$ such that $B(r') \subseteq B(r)$, $H(r') \subseteq H(r)$, and one of these two subset relations is proper; otherwise, $r$ is non-minimal.

**Algorithm 1 (Computing a result of forgetting)**
*Input*: Disjunctive program $P$ and atom $a$.
*Output*: $Forget(P, a)$.
*Procedure:*

*Step 1.* Remove tautologies, contradictory rules, and non-minimal rules from $P$. The resulting disjunctive program is still denoted $P$.

*Step 2.* Collect all rules in $P$ that do not contain the atom $a$, denoted $P'$.

*Step 3.* For each pair of rules $r_1 = A_1 \leftarrow B_1, a, \sim C_1$ and $r_2 = A_2; a \leftarrow B_2, \sim C_2$, add the rule $r = A_1; A_2 \leftarrow B_1, B_2, \sim C_1, \sim C_2$ to $P'$

*Step 4.* For each rule $r' = A \leftarrow a_1, \ldots a_n, \sim C$ where for some $i$, $a_i = a$, and for each set of $n$ rules $\{r_i = A_i \leftarrow B_i, \sim a_i, \sim C_i \mid 1 \leq i \leq n\}$, add the rule $r = A_1; \ldots; A_n \leftarrow B_1, \ldots, B_n, \sim C_1, \ldots, \sim C_n, \sim A, \sim C$ to $P'$.

*Step 5.* Return $P'$ as $Forget(P, a)$.

Some remarks for the algorithm are in order. Obviously, Step 1 preprocesses the input program by eliminating tautologous rules, contradictory rules and non-minimal rules from $P$. Initially, all rules that do not contain $a$, which are trivial SE-consequences of $P$, are included in the result of forgetting. In many practical applications, a given atom will occur in only a relatively small number of rules, and thus forgetting can be efficiently carried out, even though the input program may be very large. Step 3 and Step 4 implement two resolution rules **WGPPE** and **S-HYP**, respectively.

The correctness and completeness of Algorithm 1 are an easy corollary of Theorem 3.

**Theorem 4** *For any disjunctive program* $P$ *and an atom* $a$*, Algorithm 1 outputs* $Forget(P, a)$.

## An Application: Conflict Resolving by Forgetting

(Eiter and Wang 2006; 2008) explore how their forgetting for logic programs can be used to resolve conflicts in multi-agent systems. However, their notion of forgetting is based on answer sets and thus does not preserve the syntactic structure of original logic programs, as pointed out in (Cheng et al. 2006). In this subsection, we demonstrate how our SE-forgetting can be used to overcome the shortcoming of Eiter and Wang's forgetting.

The basic idea of conflict resolving (Eiter and Wang 2006; 2008) consists of two intuitions:

1. an answer set corresponds to an agreement among some agents;
2. conflicts are resolved by forgetting some literals/concepts for some agents/ontologies.

**Definition 4** *Let $\mathcal{S} = (P_1, P_2, \ldots, P_n)$, where each logic program $P_i$ represents the preferences/constraints of Agent $i$. A* compromise *of $\mathcal{S}$ is a sequence $C = (F_1, F_2, \ldots, F_n)$ where each $F_i$ is a set of atoms to be forgotten from $P_i$. An* agreement *of $\mathcal{S}$ on $C$ is an answer set of* $\mathsf{forget}(\mathcal{S}, C) = \mathsf{forget}(P_1, F_1) \cup \mathsf{forget}(P_2, F_2) \cup \cdots \cup \mathsf{forget}(P_n, F_n)$.

For specific applications, we may need to impose certain conditions on each $F_i$. However, the two algorithms (Algorithms 1 and 2) in (Cheng et al. 2006) may not produce intuitive results if directly used in a practical application. Consider a simple scenario with two agents.

**Example 2** *(Cheng et al. 2006) Suppose that two agents $A_1$ and $A_2$ try to reach an agreement on submitting a paper to a conference, as a* regular paper *or as a* system description*. If a paper is prepared as a system description, then the system may be implemented either in Java or Prolog. The preferences and constraints are as follows.*

1. *The same paper cannot be submitted as both a regular paper and system description.*
2. *$A_1$ would like to submit the paper as a regular one and, in case the paper is submitted as a system description and there is no conflict, he would prefer to use Java.*
3. *$A_2$ would like to submit the paper as a system description and not as a regular paper.*

*Obviously, the preferences of these two agents are jointly inconsistent and thus it is impossible to satisfy both at the same time. The scenario can be encoded as a collection of three disjunctive programs, where $P_0$ expresses general constraints). Then $\mathcal{S} = (P_0, P_1, P_2)$ where $P_0 = \{\leftarrow R, S\}$, $P_1 = \{R \leftarrow . \quad J \leftarrow S, \sim P\}$, and $P_2 = \{\leftarrow R. \quad S \leftarrow\}$. We use $R, S, J$, and $P$ for abbreviations of "regular paper," "system description," "Java" and "Prolog," respectively.*

*Intuitively, if $A_1$ can make a compromise by forgetting $R$, then there will be an agreement $\{S, J\}$, that is, a system description is prepared and Java is used for implementing the system. However, if we directly use forgetting in conflict resolution, by forgetting $R$, we can only obtain an agreement $\{S\}$ which does not contain $J$. In fact, this is caused by the removal of $J \leftarrow S, \sim P$ in the process of forgetting. This rule is redundant in $P_1$ but becomes relevant when we consider the interaction of $A_1$ with other agents (here $A_2$).*

As pointed out in (Cheng et al. 2006), it is necessary to develop a theory of forgetting for disjunctive programs such that *locally redundant* (or *locally irrelevant*) rules in the process of forgetting can be preserved. Our SE forgetting provides a solution to the above problem. This can be seen from the definition of SE-forgetting and Algorithm 1 (if needed, we do not have to eliminate non-minimal rules in Step 1). In fact, $Forget(P_1, R) = \{J \leftarrow S, \sim P\}$, which preserves the locally redundant rule $J \leftarrow S, \sim P$.

## Related Work

Earlier approaches to forgetting in answer set programming are not based on SE models. The concepts of strong and weak forgetting in (Zhang, Foo, and Wang 2005; Zhang and Foo 2006) are defined in terms of a set of program transformations. So these proposals are syntactic. A semantic approach to forgetting under answer sets is proposed in (Wang, Sattar, and Su 2005) and extended to disjunctive programs in (Eiter and Wang 2006; 2008). This definition of forgetting is syntax-dependent wrt SE-models.

More recently, some attempts have been made to define forgetting under SE models (Wang et al. 2012; Wang, Wang, and Zhang 2013). These approaches aim at forgetting in HT-logic while here we focus on forgetting in disjunctive programs, beginning with an abstract notion of forgetting. Note that our approach is not a special case of their's. Our definition of forgetting guarantees the existence of results of forgetting whereas (Wang et al. 2012) does not (see their Examples 1 and 2). To overcome this shortcoming (i.e., nonexistence of forgetting), (Wang, Wang, and Zhang 2013) proposed an improved approach to forgetting in HT-logic. While the modified definition guarantees the existence of forgetting in HT-logic, their result for forgetting in a disjunctive program may not be expressible in disjunctive programs. In addition, their definitions are more complex than ours and lack an efficient algorithm comparable to our Algorithm 1.

(Gabbay, Pearce, and Valverde 2011) is also relevant to forgetting under SE models. This work is primarily concerned with various logical properties, such as an interpolation property for equilibrium logic and answer set logic. Consequently, their objectives are different from our's. Gabbay et al. touch briefly on uniform interpolation with respect to disjunctive logic programs, but a uniform interpolant is generated directly from a program's set of answer sets. Similar to (Gabbay, Pearce, and Valverde 2011), the approaches of (Eiter and Wang 2006; 2008; Wang et al. 2012; Wang, Wang, and Zhang 2013) generate the result of forgetting through the collection of answer sets/equilibria. Consequently, in these approaches the structure of the original program is lost. Moreover, the size of the result of forgetting may be exponentially large in the size of the input program. In contrast, we do not generate answer sets; we retain the structure of the original program (so, for example, rules that do not mention a forgotten atom are untouched); and for forgetting an atom we have at worst a quadratic increase in program size.

## Conclusion

In this paper we have addressed forgetting in disjunctive logic programs, wherein forgetting amounts to a reduction in the signature of a program. Essentially, the result of forgetting an atom (or set of atoms) from a program is the set of SE consequences of the program that do not mention that atom or set of atoms. This definition then is at the *knowledge level*, that is, it is abstract and is independent of how a program is represented. Hence this theory of forgetting is useful for tasks such as knowledge base comparison and

reuse. Moreover, we gave an equivalent (with respect to SE consequence) syntactic definition of forgetting, and from this definition developed an efficient algorithm for computing forgetting. Hence this alternative definition, and the algorithm, is complete and sound with respect to the original knowledge-level definition.

A prototype implementation of SE-forgetting has been implemented in Java and is available at `http://1drv.ms/1sNNClN`. Our experiments on the efficiency of the system show that it can be used to efficiently handle SE-forgetting in large logic programs. We plan to apply this notion of forgetting to knowledge base comparison and reuse. For future work we also plan to investigate a similar approach to forgetting for other classes of logic programs.

## References


Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Boole, G. 1854. *An Investigation of the Laws of Thought*. London: Walton. (Reprinted by Dover Publications, 1954).

Chang, C. C., and Keisler, H. J. 2012. *Model Theory*. Dover Publications, third edition.

Cheng, F.-L.; Eiter, T.; Robinson, N.; Sattar, A.; and Wang, K. 2006. LPForget: A system of forgetting in answer set programming. In *Proceedings of the 19th Joint Australian Conference on Artificial Intelligence*, 1101–1105.

Delgrande, J. 2014. Toward a knowledge level analysis of forgetting. In *Proceedings of the Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 606–609.

Doets, K. 1996. *Basic Model Theory*. CSLI Publications.

Eiter, T., and Wang, K. 2006. Forgetting and conflict resolving in disjunctive logic programming. In *Proc. 21st National Conference on Artificial Intelligence*, 238–243. AAAI Press.

Eiter, T., and Wang, K. 2008. Forgetting in answer set programming. *Artificial Intelligence* 172(14):1644–1672.

Eiter, T.; Fink, M.; Tompits, H.; and Woltran, S. 2004. On eliminating disjunctions in stable logic programming. In *Proc. of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning*, 447–458.

Gabbay, D. M.; Pearce, D.; and Valverde, A. 2011. Interpolable formulas in equilibrium logic and answer set programming. *Journal of Artificial Intelligence Research* 42:917–943.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Morgan & Claypool Publishers.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, 1070–1080. The MIT Press.

Hodges, W. 1997. *A Shorter Model Theory*. Cambridge, UK: Cambridge University Press.

Konev, B.; Lutz, C.; Walther, D.; and Wolter, F. 2013. Model-theoretic inseparability and modularity of description logic ontologies. *Artificial Intelligence* 203:66–103.

Kontchakov, R.; Wolter, F.; and Zakharyaschev, M. 2008. Can you tell the difference between DL-Lite ontologies? In *Proc. 11th International Conference on Principles of Knowledge Representation and Reasoning*, 285–295.

Lang, J.; Liberatore, P.; and Marquis, P. 2003. Propositional independence : Formula-variable independence and forgetting. *Journal of Artificial Intelligence Research* 18:391–443.

Lifschitz, V.; Pearce, D.; and Valverde, A. 2001. Strongly equivalent logic programs. *ACM Transactions on Computational Logic* 2(4):526–541.

Lin, F., and Reiter, R. 1994. Forget it! In *AAAI Fall Symposium on Relevance*.

Subramanian, D.; Greiner, R.; and Pearl, J. 1997. Special issue on relevance. *Artificial Intelligence* 97(1-2).

Turner, H. 2003. Strong equivalence made easy: Nested expressions and weight constraints. *Theory and Practice of Logic Programming* 3(4):609–622.

Wang, Y.; Zhang, Y.; Zhou, Y.; and Zhang, M. 2012. Forgetting in logic programs under strong equivalence. In *Proceedings of the Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*.

Wang, K.; Sattar, A.; and Su, K. 2005. A theory of forgetting in logic programming. In *Proc. 20th National Conference on Artificial Intelligence (AAAI)*, 682–688. AAAI Press.

Wang, Y.; Wang, K.; and Zhang, M. 2013. Forgetting for answer set programming revisited. In *Proc. 23rd International Joint Conference on Artificial Intelligence*, 1162–1168.

Wong, K.-S. 2008. Sound and complete inference rules for SE-consequence. *Journal of Artificial Intelligence Research* 31(1):205–216.

Zhang, Y., and Foo, N. 1997. Answer sets for prioritized logic programs. In *Proceedings of the International Symposium on Logic Programming (ILPS-97)*, 69–84. MIT Press.

Zhang, Y., and Foo, N. 2006. Solving logic program conflict through strong and weak forgetting. *Artificial Intelligence* 170:739–778.

Zhang, Y., and Zhou, Y. 2009. Knowledge forgetting: Properties and applications. *Artificial Intelligence* 173(16-17):1525–1537.

Zhang, Y.; Foo, N. Y.; and Wang, K. 2005. Solving logic program conflict through strong and weak forgettings. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 627–634.