An Approach to Forgetting in Disjunctive Logic Programs that Preserves Strong Equivalence

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 investigate forgetting in disjunctive logic programs, where forgetting an atom from a program amounts to a reduction in the signature of that program. The goal is to provide an approach that is syntax-independent, in that if two programs are strongly equivalent, then the results of forgetting an atom in each program should also be strongly equivalent. Our central definition of forgetting is impractical but satisfies this goal: Forgetting an atom is characterised by the set of SE consequences of the program that do not mention the atom to be forgotten. We then provide an equivalent, practical definition, wherein forgetting an atom p is given by those rules in the program that don't mention p, together with rules obtained by a single inference step from rules that do mention p. Forgetting is shown to have appropriate properties; as well, the finite characterisation results in a modest (at worst quadratic) blowup. Finally we have also obtained 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 *et al.* 2003). It constitutes a reduction in an agent's language or, more accurately, signature, and has been studied under different names, such as variable elimination, uniform interpolation and relevance (Subramanian *et al.* 1997). Forgetting has various potential uses 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 or reusing part of a knowledge base or in 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 ϕ in propositional logic, one disjoins the result of uniformly substituting $\top$ for p in ϕ 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 resolving conflicts (Eiter and Wang 2008; Zhang and Foo 1997), and ontology comparison and reuse (Kontchakov *et al.* 2008; Konev *et al.* 2013).

The knowledge base of an agent may be represented in a non-classical logic, in particular a nonmonotonic approach such as answer set programming (ASP) (Gelfond and Lifschitz 1988; Baral 2003; Gebser *et al.* 2012). However, the Boole definition clearly 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 *et al.* 2005; Zhang *et al.* 2005; Zhang and Foo 2006). The approach to forgetting in (Zhang *et al.* 2005; Zhang and Foo 2006) is syntactic, in the sense that their definition of forgetting is given in terms of program transformations, but is not based on answer set semantics or SE models[1] (for normal logic programs). A semantic theory of forgetting for normal logic programs under answer set semantics is introduced in (Wang *et al.* 2005), in which a sound and complete algorithm is developed based a series of program transformations. This theory is further developed and extended to disjunctive logic programs (Eiter and Wang 2006; 2008). However, this theory of forgetting is defined in terms of standard answer set semantics instead of SE models.

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,

[1] See the next section for definitions.

that is in terms of SE models. For example, the reuse of knowledge bases requires that when a sub-program Q in a large program P is substituted with another program Q', the resulting program should be equivalent to P. This is not the case for answer set semantics due to its nonmonotonicity. As a result, two definitions of forgetting have been introduced in HT-logic (Wang *et al.* 2012; 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 *et al.* 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 not fully resolved yet. In addition, it is even more challenging to develop efficient algorithm for computing a result of forgetting under SE models.

A key intuition behind forgetting is that the logical consequences of a set of formulas that don't mention forgotten symbols should still be believed after forgetting. This leads to a very simple (abstract) knowledge-level definition, provided that a consequence operator is provided in the underlying logic. In particular, the semantics of a logic usually associates a set of models $Mod(\mathcal{K})$ with each knowledge base $\mathcal{K}$. This makes it straightforward to formulate a definition of forgetting based on the above intuition. However, such a definition of forgetting suffers from the problem of inexpressibility, i.e., the result of forgetting may not be expressible in the logic. In this paper, we establish such a theory of forgetting for disjunctive logic programs under SE models. Besides several important properties, we show that the result of forgetting for a given disjunctive program is still a disjunctive program. This result confirms the existence and expressibility of forgetting for DLP under SE models and in fact provides an algorithm for computing forgetting under SE models. We investigate some optimisation techniques for the algorithm and report a prototype implementation of the algorithm.

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

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

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. We will use $\mathcal{L}_{\mathcal{A}}$ to denote the language (viz. set of rules) generated by $\mathcal{A}$.

Without loss of generality, we assume that there are no repeated literals in a rule. The *head* and *body* of a rule r, $H(r)$ and $B(r)$, are defined by:

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

Given a set X of literals, we define

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

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 $\{q \leftarrow \sim p.\quad p \leftarrow\}$ has answer set $\{p\}$.

SE Models

As defined by (Turner 2003), an *SE interpretation* on a signature $\mathcal{A}$ 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 $SE(P)$. 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. Also, we have $(Y, Y) \in SE(P)$ iff $Y \in Mod(P)$.

A program P is *satisfiable* just if $SE(P) \neq \emptyset$.[2] Thus, for example, we consider $P = \{p \leftarrow \sim p\}$ to be satisfiable, since $SE(P) \neq \emptyset$ even though $AS(P) = \emptyset$. Two 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 *et al.* 2001). We also write $P \models_s Q$ iff $SE(P) \subseteq SE(Q)$.

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

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, where 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, upper case are sets of atoms, and for a set of atoms $C = \{c_1, \ldots, c_n\}$, $\sim C$ stands for $\{\sim c_1, \ldots, \sim c_n\}$.

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
$A_1; A_2 \leftarrow B_1, B_2, \sim C_1, \sim C_2$

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 is an SE interpretation on $\mathcal{A}$ and w' is an SE interpretation on $\mathcal{A}'$ 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.

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 that of P over the original alphabet, but excluding atoms from $\mathcal{A}'$.

This definition is very simple. This characterization is abstract, at the *knowledge level*. 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.

[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.

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. $Forget(P, \mathcal{A}) = Forget(Forget(P, \mathcal{A} \setminus \{a\}), \{a\}))$
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, 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, trivially $\sigma(P) \setminus \mathcal{A}' \subseteq \sigma(P)$, 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 $\mathcal{A} \setminus \mathcal{A}'$. Very 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 $\mathcal{A}' \subseteq \mathcal{A}$.

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

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 $P \not\vdash_s r$ then $P' \not\vdash_s r$*
4. $IR(P', \mathcal{A}')$

For the last three parts we have that, 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). The last part is a "success" postulate: the result of forgetting $\mathcal{A}'$ yields a theory expressible without $\mathcal{A}'$.

A Finite Characterisation of Forgetting

Aside: Forgetting in Propositional Logic We first take a quick detour to forgetting in propositional logic to illustrate the general approach to finitely characterising forgetting. Let ϕ be a formula in propositional logic and let p be an atom; the standard definition for forgetting p from ϕ in propositional logic is defined to be $\phi[p/\top] \vee \phi[p/\bot]$. It is not difficult to show that this is equivalent to Definition 1, but suitably re-expressed in terms of propositional logic. This definition however is not particularly convenient. It is applicable only to finite sets of formulas. As well, it results in a formula whose main connective is a disjunction.

An alternative is given as follows. Assume that a formula (or formulas) for forgetting is expressed in clause form, where a (disjunctive) clause is expressed as a set of literals. For forgetting an atom p, consider the set of all clauses obtained by resolving on p:

Definition 3 *Let S be a set of propositional clauses and $p \in \mathcal{P}$. Define*

$$\begin{aligned} Res(S, p) = \{\phi \mid\ & \exists \phi_1, \phi_2 \in S \text{ such that} \\ & p \in \phi_1 \text{ and } \neg p \in \phi_2, \text{ and} \\ & \phi = (\phi_1 \setminus \{p\}) \cup (\phi_2 \setminus \{\neg p\})\} \end{aligned}$$

We obtain the following, where $Forget_{PC}$ refers to forgetting in propositional logic:

Theorem 2 *Let S be a set of propositional clauses over signature $\mathcal{P}$ and $p \in \mathcal{P}$.*

$$Forget_{PC}(P,p) \;\leftrightarrow\; S_{|(\mathcal{P}\setminus\{p\})} \cup Res(S,p).$$

This provides an arguably more convenient means of computing forgetting, in that it is easily implementable, and one remains with a set of clauses.

Back to Forgetting in Logic Programming: We can use the same overall 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 that don't mention a. It proves to be the case that the corresponding definition is a bit more intricate, since it involves various combinations of **WGPPE** and **S-HYP**, but overall the strategy is the same as for propositional logic.

In the definition below, $ResLP$ corresponds to Res for forgetting in propositional logic. In propositional logic, Res was used to compute all resolvents on an atom a. Here the same thing is done: we consider instances of **WGPPE** and **S-HYP** in place of propositional resolution; these instances are given by the two parts of the union, respectively, below.

Definition 4 *Let P be a disjunctive logic program and $a \in \mathcal{A}$.*
Define:

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

We obtain the following:

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.

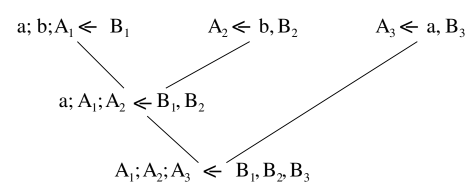

Figure 1a

Then essentially the steps involving the two applications of **WGPPE** can be "swapped", as illustrated in Figure 1b, where r_k is replaced by $r'_k = a; A_1; A_2 \leftarrow B_1, B_2$.

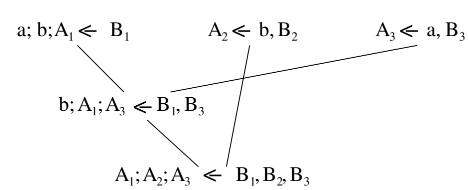

Figure 1b

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 results of forgetting are 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. While this may seem comparatively modest, it implies that forgetting a set of atoms may result in an exponential blowup.

Example 1 *Let $P = \{p \leftarrow \sim q. \;\; r \leftarrow p\}$. 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

By Theorem 3, we have 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 a 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 literal a in P.
Output: $Forget(P, a)$.
Procedure:

Step 1. Remove tautology rules, contradiction 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 $r' = A \leftarrow a_1, \dots a_n, \sim C$ 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 $r = A_1; \dots; A_n \leftarrow B_1, \dots, B_n, \sim C_1, \dots, \sim C_n, \sim A, \sim C$ to P'.

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

Some remarks for the algorithm are in order. Obviously, Step 1 is to preprocesss the input program by eliminating tautology rules, contradiction 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, such a part of input program is usually not very large and thus forgetting can be efficiently done although the input program can be very large. Step 3 and Step 4 implement two resolution rules **WGPPE** and **S-HYP**, respectively.

Conflict Resolving by Forgetting: Revisited

(Eiter and Wang 2006; 2008) explore how their semantic 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 this shortcoming of Eiter and Wang's forgetting can be overcome in our SE-forgetting for disjunctive programs.

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

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

Definition 5 *Let $\mathcal{S} = (P_1, P_2, \dots, P_n)$, where each logic program P_i represents the preferences/constraints of Agent i. A* compromise *of $\mathcal{S}$ is a sequence $C = (F_1, F_2, \dots, F_n)$ 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* $\mathsf{forget}(\mathcal{S}, C) = \mathsf{forget}(P_1, F_1) \cup \mathsf{forget}(P_2, F_2) \cup \dots \cup \mathsf{forget}(P_n, F_n)$.

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 A1 and A2 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. *A1 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. *A2 would like to submit the paper as a system description but not prefer 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 (P_0 stands for general constraints): $\mathcal{S} = (P_0, P_1, P_2)$ where R, S, J, P mean "regular paper," "system description," "Java" and "Prolog," respectively: $P_0 = \{\leftarrow R, S\}, P_1 = \{R \leftarrow . \quad J \leftarrow S, \sim P\}, P_2 = \{\leftarrow R. \quad S \leftarrow\}$.*

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 abundant in P_1 but becomes relevant when we consider the interaction of A1 with other agents (here A2).

As pointed out in (Cheng *et al.* 2006), it is necessary to develop a theory of forgetting for disjunctive programs such that *locally abundant* (or *locally irrelevant*) rules in the process of forgetting can be preserved. Our SE forgetting provides an ideal solution to the above problem. This can be seen from the definition of SE-forgetting and Algorithm 1 (if needed, we don't 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 $J \leftarrow S, \sim P$.

Conclusion

In this paper we have addressed forgetting under SE models 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. A result of the proposed forgetting under SE models is also a result of forgetting under answer sets but not vice versa. Moreover, we have developed an efficient algorithm for computing forgetting in disjunctive logic programs, which is complete and sound with respect to the original knowledge-level definition.

A prototype implementation, of forgetting has been implemented in Java and is available publicly at `http://www.ict.griffith.edu.au/~kewen/SE-Forget/`. While our experiments on the efficiency of the system are still underway, preliminary results show that the algorithm is very efficient. Currently we are still working on improving efficiency of the implementation and are experimenting on applying it to large practical logic programs and randomly generated 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

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

George Boole. *An Investigation of the Laws of Thought*. Walton, London, 1854. (Reprinted by Dover Books, New York, 1954).

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

Fu-Leung Cheng, Thomas Eiter, Nathan Robinson, Abdul Sattar, and Kewen Wang. LPForget: A system of forgetting in answer set programming. In *Proceedings of the 19th Joint Australian Conference on Artificial Intelligence*, pages 1101–1105, 2006.

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

Thomas Eiter and Kewen Wang. Forgetting and conflict resolving in disjunctive logic programming. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 238–243. AAAI Press, 2006.

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

Dov M. Gabbay, David Pearce, and Agustín Valverde. Interpolable formulas in equilibrium logic and answer set programming. *J. Artif. Intell. Res. (JAIR)*, 42:917–943, 2011.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

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

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

Boris Konev, Carsten Lutz, Dirk Walther, and Frank Wolter. Model-theoretic inseparability and modularity of description logic ontologies. *Artificial Intelligence*, 203:66–103, 2013.

Roman Kontchakov, Frank Wolter, and Michael Zakharyaschev. Can you tell the difference between DL-Lite ontologies? In *Proceedings of the*

11th International Conference on Principles of Knowledge Representation and Reasoning (KR-08), pages 285–295, 2008.

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

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

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

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

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

Kewen Wang, Abdul Sattar, and Kaile Su. A theory of forgetting in logic programming. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI)*, pages 682–688. AAAI Press, 2005.

Yisong Wang, Yan Zhang, Yi Zhou, and Mingyi Zhang. Forgetting in logic programs under strong equivalence. In *Proceedings of the Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2012.

Yisong Wang, Kewen Wang, and Mingyi Zhang. Forgetting for answer set programming revisited. In *Proceedings, The 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1162–1168, 2013.

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

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

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

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

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