Syntactic ASP Forgetting with Forks

Felicidad Aguado[1][0000−0002−4334−9267], Pedro Cabalar[1][0000−1111−2222−3333], Jorge Fandinno[2][0000−0002−3917−8717], David Pearce[3][0000−0001−7407−326X], Gilberto Pérez[1][0000−0001−6269−6101], and Concepción Vidal[1][0000−0002−5561−6406]

[1] University of Corunna, Spain,
{felicidad.aguado,cabalar,gperez,concepcion.vidalm}@udc.es,
[2] University of Nebraska at Omaha, NE, USA,
jfandinno@unomaha.edu
[3] Universidad Politécnica de Madrid, Spain,
david.pearce@upm.es

Abstract. In this paper, we present a syntactic transformation, called the *unfolding* operator, that allows forgetting an atom in a logic program (under ASP semantics). The main advantage of unfolding is that, unlike other syntactic operators, it is always applicable and guarantees strong persistence, that is, the result preserves the same stable models with respect to any context where the forgotten atom does not occur. The price for its completeness is that the result is an expression that may contain the fork operator. Yet, we illustrate how, in some cases, the application of fork properties may allow us to reduce the fork to a logic program, even in conditions that could not be treated before using the syntactic methods in the literature.

Keywords: Answer Set Programming, Equilibrium Logic, Forgetting, Strong Persistence, Strong Equivalence, Forks

1 Introduction

A common representational technique in Answer Set Programming [13, 15] (ASP) is the use of auxiliary atoms. Their introduction in a program may be due to many different reasons, for instance, looking for a simpler reading, providing new constructions (choice rules, aggregates, transitive closure, etc) or reducing the corresponding ground program. When a program (or program fragment) Π for signature $\mathcal{AT}$ uses auxiliary atoms $A \subseteq \mathcal{AT}$, they do not have a relevant meaning outside Π. Accordingly, they are usually removed[4] from the final stable models, so the latter only use atoms in $V = \mathcal{AT} \setminus A$, that is, the relevant or *public vocabulary* that encodes the solutions to our problem in mind. Thus, when seen from outside, Π becomes a black box that hides internal atoms from A and provides solutions in terms of public atoms from V. A reasonable question is whether we can transform these black boxes into white boxes, that is, whether we can

[4] Most ASP solvers allow hiding the extension of some chosen predicates.

reformulate some program Π exclusively in terms of public atoms V, forgetting the auxiliary ones in A. A *forgetting operator* $\mathtt{f}(\Pi, A) = \Pi'$ transforms a logic program Π into a new program Π' that does not contain atoms in A but has a *similar* behaviour on the public atoms V. Of course, the key point here is the definition of similarity between Π and Π' (relative to V) something that gave rise to different alternative forgetting operators, further classified in families, depending on the properties they satisfy – see [9] for an overview. From all this wide spectrum, however, when our purpose is forgetting auxiliary atoms, similarity can only be understood as *preserving the same knowledge* for public atoms in V, and this can be formalised as a very specific property. In particular, both programs Π and $\Pi' = \mathtt{f}(\Pi, A)$ should not only produce the same stable models (projected on V) but also keep doing so even if we add a new piece of program Δ without atoms in A. This property, known as *strong persistence*, was introduced in [12] but, later on, [10] proved that it is not always possible to forget A in an arbitrary program Π under strong persistence. Moreover, [10] also provided a semantic condition, called Ω, on the models of Π in the logic of Here-and-There (HT) [11] (the monotonic basis of *Equilibrium Logic* [16]) so that atoms A are forgettable in Π iff Ω does not hold. When this happens, their approach can be used to construct $\mathtt{f}(\Pi, A)$ from the HT models using, for instance, the method from [5, 7]. Going one step further in this model-based orientation for forgetting, [1] overcame the limitation of unforgettable sets of atoms at the price of introducing a new type of disjunction, called *fork* and represented as '|'. To this aim, [1] defined an HT-based denotational semantics for forks.

Semantic-based forgetting is useful when we are interested in obtaining a compact representation. For instance, the method from [7] allows obtaining a minimal logic program from a set of HT-countermodels. However, this is done at a high computational cost (similar to Boolean function minimisation techniques). When combined with the Ω-condition or, similarly, with the use of HT-denotations, this method becomes practically unfeasible without the use of a computer. This may become a problem, for instance, when we try to prove properties of some new use of auxiliary atoms in a given setting, since one would expect a human-readable proof rather than resorting to a computer-based exhaustive exploration of models. On the other hand, semantic forgetting may easily produce results that look substantially different from the original program, even when this is not necessary. For example, if we apply an empty forgetting $\mathtt{f}(\Pi, \emptyset)$ strictly under this method, we will usually obtain a different program Π', strongly equivalent to Π, but built up from countermodels of the latter, possibly having a very different syntactic look.

An alternative and in some sense complementary orientation for forgetting is the use of *syntactic transformations*. [12] introduced the first syntactic forgetting operator, $\mathtt{f}_{as}$, that satisfied strong persistence. This operator forgot a single atom $A = \{a\}$ at a time and was applicable, under some conditions, to non-disjunctive logic programs. More recently, [4] presented a more general syntactic operator $\mathtt{f}_{sp}$, also for a single atom $A = \{a\}$, that can be applied to any arbitrary logic program and satisfies strong persistence when the atom can be

forgotten (i.e. the Ω condition does not hold). Moreover, Berthold et al. [4] also provided three syntactic sufficient conditions (that they call *a-forgettable*) under which Ω does not hold, and so, under which $\mathtt{f}_{sp}$ is strongly persistent. Perhaps the main difficulty of $\mathtt{f}_{sp}$ comes from its complex definition: it involves 10 different types of rule-matching that further deal with multiple partitions of Π (using a construction called *as-dual*). As a result, even though it offers full generality when the atom is forgettable, its application by hand does not seem very practical, requiring too many steps and a careful reading of the transformations.

In this paper, we provide a general syntactic operator, called *unfolding*, that is always applicable and allows forgetting an atom in a program, although it produces a result that may combine forks and arbitrary propositional formulas. We also discuss some examples in which a fork can be removed in favour of a formula, something that allows one to obtain a standard program (since formulas can always be reduced to that form [6]). We show examples where sufficient syntactic conditions identified so far are not applicable, whereas our method can still safely be applied to obtain a correct result, relying on properties of forks. Unfolding relies on another syntactic operator for forgetting a single atom, $\mathtt{f}_c$, based on the *cut rule* from a sequent calculus and is close to the application of $\mathtt{f}_{sp}$ from [4]. This operator produces a propositional formula without forks, but is only applicable under some sufficient syntactic conditions.

The rest of the paper is organised as follows. Section 2 contains a background with definitions and results from HT, stable models and the semantics of forks. Section 3 presents the cut transformation that produces a propositional formula. Then, Section 4 introduces the unfolding, which makes use of the cut and produces a fork in the general case. Finally, Section 5 concludes the paper.

2 Background

We begin by recalling some basic definitions and results related to the logic of HT. Let $\mathcal{AT}$ be a finite set of atoms called the *alphabet* or *vocabulary*. A *(propositional) formula* φ is defined using the grammar:

$$\varphi \ ::= \ \bot \ \mid \ p \ \mid \ \varphi \wedge \varphi \ \mid \ \varphi \vee \varphi \ \mid \ \varphi \rightarrow \varphi$$

where p is an atom $p \in \mathcal{AT}$. We define the *language* $\mathcal{L}_{\mathcal{AT}}$ as the set of all propositional formulas that can be formed over alphabet $\mathcal{AT}$. We use Greek letters φ, ψ, γ and their variants to stand for formulas. Implication $\varphi \rightarrow \psi$ will be sometimes reversed as $\psi \leftarrow \varphi$. We also define the derived operators $\neg\varphi \stackrel{\text{def}}{=} (\varphi \rightarrow \bot)$, $\top \stackrel{\text{def}}{=} \neg\bot$ and $\varphi \leftrightarrow \psi \stackrel{\text{def}}{=} (\varphi \rightarrow \psi) \wedge (\varphi \leftarrow \psi)$. We use letters p, q, a, b for representing atoms in $\mathcal{AT}$, but normally use a for an auxiliary atom to be forgotten. A *theory* Γ is a finite[5] set of formulas that can be also understood as their conjunction. When a theory consists of a single formula $\Gamma = \{\varphi\}$ we will frequently

[5] As we will see, the cut operator support is a conjunction built from a finite set of rules that is sometimes negated. Generalising to infinite theories would require infinitary Boolean connectives.

omit the brackets. Given any theory Γ, we write $\Gamma[\gamma/\varphi]$ to denote the uniform substitution of all occurrences of subformula γ in Γ by formula φ. An *extended disjunctive rule* r is an implication of the form:

$$p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n \wedge \neg\neg p_{n+1} \wedge \cdots \wedge \neg\neg p_k \;\rightarrow\; p_{k+1} \vee \cdots \vee p_h$$

where all p_i above are atoms in $\mathcal{AT}$ and $0 \leq m \leq n \leq k \leq h$. The antecedent and consequent of a rule r are respectively called the *body* and the *head*. We define the sets of atoms $Hd(r) \stackrel{\text{def}}{=} \{p_{k+1}, \ldots, p_h\}$, $Bd^+(r) \stackrel{\text{def}}{=} \{p_1, \ldots, p_m\}$, $Bd^-(r) \stackrel{\text{def}}{=} \{p_{m+1}, \ldots, p_n\}$, $Bd^{--}(r) \stackrel{\text{def}}{=} \{p_{n+1}, \ldots, p_k\}$ and $Bd(r) \stackrel{\text{def}}{=} Bd^+(r) \cup Bd^-(r) \cup Bd^{--}(r)$. We say that r is an *extended normal rule* if $|Hd(r)| \leq 1$. A rule with $Hd(r) = \emptyset$ is called a *constraint*. A normal rule with $Bd(r) = \emptyset$ and $|Hd(r)| = 1$ is called a *fact*. Given some atom a, a rule r is said to contain an *a-choice* if $a \in Bd^{--}(r) \cap Hd(r)$, that is, the rule has the form $\varphi \wedge \neg\neg a \rightarrow \psi \vee a$. A program is a finite set of rules, sometimes represented as their conjunction. We say that program Π belongs to a syntactic category if all its rules belong to that category. For instance, Π is an extended normal program if all its rules are extended normal. We will usually refer to the most general class, extended disjunctive logic programs, just as logic programs for short.

A *classical interpretation* T is a set of atoms $T \subseteq \mathcal{AT}$. We write $T \models \varphi$ to stand for the usual classical satisfaction of a formula φ. An HT-*interpretation* is a pair $\langle H, T\rangle$ (respectively called "here" and "there") of sets of atoms $H \subseteq T \subseteq \mathcal{AT}$; it is said to be *total* when $H = T$. The fact that an interpretation $\langle H, T\rangle$ *satisfies* a formula φ, written $\langle H, T\rangle \models \varphi$, is recursively defined as follows:

- $\langle H, T\rangle \not\models \bot$
- $\langle H, T\rangle \models p$ iff $p \in H$
- $\langle H, T\rangle \models \varphi \wedge \psi$ iff $\langle H, T\rangle \models \varphi$ and $\langle H, T\rangle \models \psi$
- $\langle H, T\rangle \models \varphi \vee \psi$ iff $\langle H, T\rangle \models \varphi$ or $\langle H, T\rangle \models \psi$
- $\langle H, T\rangle \models \varphi \rightarrow \psi$ iff both (i) $T \models \varphi \rightarrow \psi$ and (ii) $\langle H, T\rangle \not\models \varphi$ or $\langle H, T\rangle \models \psi$

An HT-interpretation $\langle H, T\rangle$ is a *model* of a theory Γ if $\langle H, T\rangle \models \varphi$ for all $\varphi \in \Gamma$. Two formulas (or theories) φ and ψ are HT-equivalent, written $\varphi \equiv \psi$, if they have the same HT-models. The logic of HT satisfies the law of substitution of logical equivalents so, in particular:

$$\Pi \wedge a \;\equiv\; \Pi \wedge (a \leftrightarrow \top) \;\equiv\; \Pi[a/\top] \wedge a \tag{1}$$

$$\Pi \wedge \neg a \;\equiv\; \Pi \wedge (a \leftrightarrow \bot) \;\equiv\; \Pi[a/\bot] \wedge \neg a \tag{2}$$

$$\Pi \wedge \neg\neg a \;\equiv\; \Pi \wedge (\neg a \leftrightarrow \bot) \;\equiv\; \Pi[\neg a/\bot] \wedge \neg\neg a \tag{3}$$

A total interpretation $\langle T, T\rangle$ is an *equilibrium model* of a formula φ iff $\langle T, T\rangle \models \varphi$ and there is no $H \subset T$ such that $\langle H, T\rangle \models \varphi$. If so, we say that T is a *stable model* of φ. We write $SM(\varphi)$ to stand for the set of stable models of φ and $SM_V(\varphi) \stackrel{\text{def}}{=} \{\, T \cap V \mid T \in SM(\varphi) \,\}$ for their projection onto some vocabulary V.

In [1], we extended logic programs to include a new construct ' | ' we called *fork* and whose intuitive meaning is that the stable models of two logic programs $\Pi_1 \mid \Pi_2$ correspond to the union of stable models from Π_1 and Π_2 in any context Π', that is $SM((\Pi_1 \mid \Pi_2) \wedge \Pi') = SM(\Pi_1 \wedge \Pi') \cup SM(\Pi_2 \wedge \Pi')$. Using

this construct, we studied the property of projective strong equivalence (PSE) for forks: two forks satisfy PSE for a vocabulary V iff they yield the same stable models projected on V for any context over V. We also provided a semantic characterisation of PSE that allowed us to prove that it is always possible to forget (under strong persistence) an auxiliary atom in a fork, something proved to be false in standard HT. We recall now some definitions from [1] and [3].

Definition 1. *Given $T \subseteq \mathcal{AT}$, a* T-support $\mathcal{H}$ *is a set of subsets of T, that is $\mathcal{H} \subseteq 2^T$, satisfying that $\mathcal{H} \neq \emptyset$ iff $T \in \mathcal{H}$.*

To increase readability, we write a support as a sequence of interpretations between square brackets. For instance, possible supports for $T = \{a, b\}$ are $[\{a, b\}\ \{a\}]$, $[\{a, b\}\ \{b\}\ \emptyset]$ or the empty support $[\,]$. Given a propositional formula φ and $T \subseteq \mathcal{AT}$, the set of HT-models $\{H \subseteq T \mid \langle H, T\rangle \models \varphi\}$ forms a T-support we denote as $[\![\,\varphi\,]\!]^T$.

For any T-support $\mathcal{H}$ and set of atoms V, we write $\mathcal{H}_V$ to stand for $\{H \cap V \mid H \in \mathcal{H}\}$. We say that a T-support $\mathcal{H}$ is *V-feasible* iff there is no $H \subset T$ in $\mathcal{H}$ satisfying that $H \cap V = T \cap V$. The name comes from the fact that, if this condition does not hold for some $\mathcal{H} = [\![\,\varphi\,]\!]^T$ with $T \subseteq V$, then T cannot be stable for any formula $\varphi \wedge \psi$ with $\psi \in \mathcal{L}(V)$ because $\langle H, T\rangle \models \varphi \wedge \psi$ and $H \subset T$.

We can define an order relation $\preceq$ between T-supports by saying that, given two T-supports, $\mathcal{H}$ and $\mathcal{H}'$, $\mathcal{H} \preceq \mathcal{H}'$ iff either $\mathcal{H} = [\,]$ or $[\,] \neq \mathcal{H}' \subseteq \mathcal{H}$. It is clear that $[\,]$ and $[T]$ are the bottom and top elements, respectively, in the class of all T-supports. Given a T-support $\mathcal{H}$, we define its complementary support $\overline{\mathcal{H}}$ as:

$$\overline{\mathcal{H}} \stackrel{\text{def}}{=} \begin{cases} [\,] & \text{if } \mathcal{H} = 2^T \\ [\ T\ ] \cup \{H \subseteq T \mid H \notin \mathcal{H}\} & \text{otherwise} \end{cases}$$

We also consider the *ideal* of $\mathcal{H}$ defined as $\downarrow\mathcal{H} = \{\mathcal{H}' \mid \mathcal{H}' \preceq \mathcal{H}\} \setminus \{\ [\,]\ \}$. Note that, the empty support $[\,]$ is not included in the ideal, so $\downarrow[\,] = \emptyset$. If Δ is any set of supports:

$$^{\downarrow}\Delta \stackrel{\text{def}}{=} \bigcup_{\mathcal{H} \in \Delta} \downarrow\mathcal{H} = \bigcup_{\mathcal{H} \in \Delta} \{\ \mathcal{H}' \preceq \mathcal{H} \mid \mathcal{H}' \neq [\,]\ \}$$

Definition 2. *A* T-view Δ *is a set of T-supports that is $\preceq$-closed, i.e., $^{\downarrow}\Delta = \Delta$.*

A fork is defined using the grammar:

$$F \ ::= \ \bot \ \mid \ p \ \mid \ (F \mid F) \ \mid \ F \wedge F \ \mid \ \varphi \vee \varphi \ \mid \ \varphi \rightarrow F$$

where φ is a propositional formula and $p \in \mathcal{AT}$ is an atom. We write $\mathcal{L}_{\mathcal{AT}}$ to stand for the language formed by all forks for signature $\mathcal{AT}$. Given a fork $(F \mid G)$, we say that F and G are its left and right *branches*, respectively.

We provide next the semantics of forks in terms of *T-denotations*. To this aim, we will use a weaker version of the membership relation, $\hat{\in}$, defined as follows. Given a T-view Δ, we write $\mathcal{H}\,\hat{\in}\,\Delta$ iff $\mathcal{H} \in \Delta$ or both $\mathcal{H} = [\,]$ and $\Delta = \emptyset$.

Definition 3 (T-denotation of a fork). *Let $\mathcal{AT}$ be a propositional signature and $T \subseteq \mathcal{AT}$ a set of atoms. The* T-denotation *of a fork F, written $\langle\!\langle\, F \,\rangle\!\rangle^T$, is a T-view recursively defined as follows:*

$$
\begin{aligned}
\langle\!\langle\, \bot \,\rangle\!\rangle^T &\stackrel{\text{def}}{=} \emptyset \\
\langle\!\langle\, p \,\rangle\!\rangle^T &\stackrel{\text{def}}{=} {\downarrow}[\![\, p \,]\!]^T \quad \textit{for any atom } p \\
\langle\!\langle\, F \wedge G \,\rangle\!\rangle^T &\stackrel{\text{def}}{=} {\downarrow}\{\ \mathcal{H} \cap \mathcal{H}' \mid \mathcal{H} \in \langle\!\langle\, F \,\rangle\!\rangle^T \textit{ and } \mathcal{H}' \in \langle\!\langle\, G \,\rangle\!\rangle^T \ \} \\
\langle\!\langle\, \varphi \vee \psi \,\rangle\!\rangle^T &\stackrel{\text{def}}{=} {\downarrow}\{\ \mathcal{H} \cup \mathcal{H}' \mid \mathcal{H} \,\hat{\in}\, \langle\!\langle\, \varphi \,\rangle\!\rangle^T \textit{ and } \mathcal{H}' \,\hat{\in}\, \langle\!\langle\, \psi \,\rangle\!\rangle^T \ \} \\
\langle\!\langle\, \varphi \to F \,\rangle\!\rangle^T &\stackrel{\text{def}}{=} \begin{cases} \{2^T\} & \textit{if } [\![\, \varphi \,]\!]^T = [\ ] \\ {\downarrow}\{\ \overline{[\![\, \varphi \,]\!]^T} \cup \mathcal{H} \mid \mathcal{H} \in \langle\!\langle\, F \,\rangle\!\rangle^T \ \} & \textit{otherwise} \end{cases} \\
\langle\!\langle\, F \mid G \,\rangle\!\rangle^T &\stackrel{\text{def}}{=} \langle\!\langle\, F \,\rangle\!\rangle^T \cup \langle\!\langle\, G \,\rangle\!\rangle^T
\end{aligned}
$$

If F is a fork and $T \subseteq V \subseteq \mathcal{AT}$, we can define the T-view:

$$\langle\!\langle\, F \,\rangle\!\rangle^T_V \stackrel{\text{def}}{=} {\downarrow}\{\ \mathcal{H}_{|V} \mid \mathcal{H} \in \langle\!\langle\, F \,\rangle\!\rangle^Z \text{ s.t. } Z \cap V = T \text{ and } \mathcal{H} \text{ is } V\text{-feasible} \ \}$$

Definition 4 (Projective Strong Equivalence). *Let F and G be forks and $V \subseteq \mathcal{AT}$ a set of atoms. We say that F and G* are V-strongly equivalent*, in symbols $F \cong_V G$, if for any fork L in $\mathcal{L}_V$, $SM_V(F \wedge L) = SM_V(G \wedge L)$. When $V = \mathcal{AT}$ we write $F \cong G$ dropping the V subindex and simply saying that F and G are* strongly equivalent*.*

The properties listed in the following theorem were proved in [1].

Theorem 1. *Let F and G be arbitrary forks, and φ and ψ propositional formulas all of them for signature $\mathcal{AT}$, and let $V \subseteq \mathcal{AT}$. Then:*

(i) $F \cong_V G$ *iff* $\langle\!\langle\, F \,\rangle\!\rangle^T_V = \langle\!\langle\, G \,\rangle\!\rangle^T_V$*, for every* $T \subseteq V$

(ii) $F \cong G$ *iff* $\langle\!\langle\, F \,\rangle\!\rangle^T = \langle\!\langle\, G \,\rangle\!\rangle^T$*, for every* $T \subseteq \mathcal{AT}$

(iii) $\langle\!\langle\, \varphi \,\rangle\!\rangle^T = {\downarrow}[\![\, \varphi \,]\!]^T$ *for every* $T \subseteq \mathcal{AT}$

(iv) $\varphi \cong \psi$ *iff* $[\![\, \varphi \,]\!]^T = [\![\, \psi \,]\!]^T$*, for every* $T \subseteq \mathcal{AT}$*, iff* $\varphi \equiv \psi$ *in* HT*.*

(v) The set of atoms $\mathcal{AT} \setminus V$ can be forgotten in F as a strongly persistent propositional formula[6] *iff for each $T \subseteq V$, $\langle\!\langle\, F \,\rangle\!\rangle^T_V$ has a unique maximal support.* □

Proposition 1. *For every pair α and β of propositional formulas:*

$$(\top \mid \alpha) \cong (\neg\alpha \mid \alpha) \cong \neg\neg\alpha \to \alpha \cong \alpha \vee \neg\alpha \tag{4}$$

$$(\bot \mid \alpha) \cong \alpha \tag{5}$$

$$(\neg\alpha \mid \neg\neg\alpha) \cong \top \tag{6}$$

$$(\alpha \wedge \neg\beta \mid \alpha \wedge \neg\neg\beta) \cong \alpha \tag{7}$$

Proposition 2. *Let F, F', G and G' be forks for some signature $\mathcal{AT}$ and let $V \subseteq \mathcal{AT}$. If $F \cong_V F'$ and $G \cong_V G'$ then $(F \mid G) \cong_V (F' \mid G')$.* □

[6] This is, therefore, equivalent to not satisfying the Ω condition from [10].

3 The Cut Operator

Given any program Π, let us define the syntactic transformation $behead^a(\Pi)$ as the result of removing all rules with $a \in Hd(r) \cap Bd^+(r)$ and all head occurrences of a from rules where $a \in Hd(r) \cap Bd^-(r)$. Intuitively, $behead^a(\Pi)$ removes from Π all rules that, having a in the head, do not provide a support for a. In fact, rules with $a \in Hd(r) \cap Bd^+(r)$ are tautological, whereas rules of the form $\varphi \wedge \neg a \to a \vee \psi$ are strongly equivalent to $\varphi \wedge \neg a \to \psi$. As a result:

Proposition 3. *For any logic program Π: $\Pi \cong behead^a(\Pi)$.* □

The cut operator is defined in terms of the well-known *cut inference rule* from the sequent calculus which, when rephrased for program rules, amounts to:

$$\frac{\varphi \wedge a \to \psi \qquad \varphi' \to a \vee \psi'}{\varphi \wedge \varphi' \to \psi \vee \psi'} \tag{CUT}$$

where φ, φ' are conjunctions of elements that can be an atom a, its negation $\neg a$ or its double negation $\neg\neg a$, and ψ' and ψ are disjunctions of atoms. If r and r' stand for $\varphi \wedge a \to \psi$ and $\varphi' \to a \vee \psi'$ respectively, then we denote $Cut(a, r, r')$ to stand for the resulting implication $\varphi \wedge \varphi' \to \psi \vee \psi'$.

Example 1 (Example 9 from [4]). Let Π_1 be the program:

$$a \to t \tag{8}$$
$$\neg a \to v \tag{9}$$
$$s \to a \tag{10}$$
$$r \to a \vee u \tag{11}$$

Then, $Cut(a, (8), (11)) = (r \to t \vee u)$ is the result of the cut application:

$$\frac{\top \wedge a \to t \qquad r \to a \vee u}{\top \wedge r \to t \vee u}$$

In this program we can also perform a second cut through atom a corresponding to $Cut(a, (8), (10)) = (s \to t)$. □

Given a rule r with $a \in Bd^+(r)$, we define the formula:

$$NES(\Pi, a, r) \stackrel{\text{def}}{=} \bigwedge\{\ Cut(a, r, r') \mid r' \in \Pi,\ a \in Hd(r')\ \}$$

that is, $NES(\Pi, a, r)$ collects the conjunction of all possible cuts in Π for a given atom a and a selected rule r with a in the positive body. For instance, in our example program Π_1 for rule (8) we get:

$$NES(\Pi_1, a, (8)) = (r \to t \vee u) \wedge (s \to t). \tag{12}$$

When $r = \neg a = (\top \wedge a \to \bot)$ we can observe that:

$$\begin{aligned} NES(\Pi, a, \neg a) &= \bigwedge\{(\top \wedge \varphi' \to \bot \vee \psi') \mid (\varphi' \to a \vee \psi') \in \Pi\} \\ &= \bigwedge\{(\varphi' \to \psi') \mid (\varphi' \to a \vee \psi') \in \Pi\} \end{aligned}$$

That is, we just take the rules with a in the head, but after removing a from that head. As an example, $NES(\Pi_1, a, \neg a) = (s \to \bot) \wedge (r \to u) = \neg s \wedge (r \to u)$. Note that, since a was the only head atom in (10), after removing it, we obtained an empty head $\bot$ leading to $(s \to \bot)$.

An interesting relation emerges from the negation of NES that can be connected with the so-called *external support* from [8] . In particular, we can use de Morgan and the HT equivalence $\neg(\varphi' \to \psi') \equiv \neg\neg\varphi' \wedge \neg\psi'$ to conclude:

$$\neg NES(\Pi, a, \neg a) = \neg\neg \bigvee \{(\varphi' \wedge \neg\psi') \mid (\varphi' \to a \vee \psi') \in \Pi\} \;=\; \neg\neg ES_\Pi(a)$$

where $ES_\Pi(a)$ corresponds to the external support[7] $ES_\Pi(Y)$ from [8] for any set of atoms Y, but applied here to $Y = \{a\}$. In the example:

$$\neg NES(\Pi_1, a, \neg a) = \neg(\neg s \wedge (r \to u)) \equiv \neg\neg s \vee (\neg\neg r \wedge \neg u) \tag{13}$$

Definition 5 (Cut operator $\mathtt{f}_c$). *Let Π be a logic program for alphabet $\mathcal{AT}$ and let $a \in \mathcal{AT}$. Then $\mathtt{f}_c(\Pi, a)$ is defined as the result of:*

(i) *Remove atom 'a' from non-supporting heads obtaining $\Pi' = behead^a(\Pi)$;*
(ii) *Replace each rule $r \in \Pi'$ with $a \in B^+(r)$ by $NES(\Pi', a, r)$.*
(iii) *From the result, remove every rule r with $Hd(r) = \{a\}$;*
(iv) *Finally, replace the remaining occurrences of 'a' by $\neg NES(\Pi', a, \neg a)$.* □

Example 2 (Example 1 continued). Step (i) has no effect, since $behead^a(\Pi_1) = \Pi_1$. For step (ii), the only rule with a in the positive body is (8) and so, the latter is replaced by (12). Step (iii) removes rule (10) and, finally, Step (iv) replaces a by (13) in rules (9) and (11). Finally, $\mathtt{f}_c(\Pi_1, a)$ becomes to the conjunction of:

$$(s \to t) \wedge (r \to t \vee u) \tag{14}$$
$$\neg(\neg\neg s \vee (\neg\neg r \wedge \neg u)) \to v \tag{15}$$
$$r \to \neg\neg s \vee (\neg\neg r \wedge \neg u) \vee u \tag{16}$$

Now, by simple HT transformations [6], it is easy to see that the antecedent of (15) amounts to $\neg s \wedge (\neg r \vee \neg\neg u))$, so (15) can be replaced by the two rules (17) and (18) below, whereas (16) is equivalent to the conjunction of (19) below that stems from $r \to \neg\neg s \vee \neg u \vee u$, plus the rule $r \to \neg\neg s \vee \neg\neg r \vee u$ that is tautological and can be removed.

$$\neg s \wedge \neg r \to v \tag{17}$$
$$\neg s \wedge \neg\neg u \to v \tag{18}$$
$$r \wedge \neg s \wedge \neg\neg u \to u \tag{19}$$

To sum up, $\mathtt{f}_c(\Pi_1, a)$ is strongly equivalent to program (14)∧(17)∧(18)∧(19). □

[7] In fact, [2] presented a more limited forgetting operator $\mathtt{f}_{es}$ based on the external support.

The program we obtained above is the same one obtained with the $\mathtt{f}_{sp}$ operator in [4] although the process to achieve it, is slightly different. This is because, in general, $\mathtt{f}_c(\Pi, a)$ takes a logic program Π but produces a *propositional formula* where a has been forgotten, whereas $\mathtt{f}_{sp}$ produces the logic program in a direct way. Although, at a first sight, this could be seen as a limitation of $\mathtt{f}_c$, the truth is that it is not an important restriction, since there exist well-known syntactic methods [6, 14] to transform a propositional formula[8] into a (strongly equivalent) logic program under the logic of HT. Moreover, in the case of $\mathtt{f}_{sp}$, directly producing a logic program comes with the cost of a more complex transformation, with ten different cases and the combinatorial construction of a so-called *as-dual* set of rules generated from multiple partitions of the original program[9]. We suggest that well-known logical rules such as de Morgan or distributivity (many of them still valid in intuitionistic logic) are far easier to learn and apply than the $\mathtt{f}_{sp}$ transformation when performing syntactic transformations by hand. On the other hand, we may sometimes be interested in keeping the propositional formula representation inside HT (for instance, for studying strong equivalence or the relation to other constructions) rather than being forced to unfold the formula into a logic program, possibly leading to a combinatorial explosion due to distributivity.

As happened with $\mathtt{f}_{sp}$, the main restriction of $\mathtt{f}_c$ is that it does not always guarantee strong persistence. Note that this was expected, given the already commented result on the impossibility of arbitrary forgetting by just producing an HT formula. To check whether forgetting a in Π is possible, we can use semantic conditions like Theorem 1(v) or the Ω-condition, but these imply inspecting the models of Π. If we want to keep the method at a purely syntactic level, however, we can at best enumerate sufficient conditions for forgettability. For instance, [4] proved that a can be forgotten under strong persistence in any program Π that satisfies any of the following syntactic conditions:

Definition 6 (Definition 4 from [4]). *An extended logic program Π is* a-forgettable *if, at least one of the following conditions is satisfied:*

1. *Π contains the fact 'a' as a rule.*
2. *Π does not contain a-choices.*
3. *All rules in Π in which a occurs are a-choices.*

It is not difficult to see that Condition 2 above is equivalent to requiring that atom a does not occur in $NES(\Pi, a, \neg a)$, since the only possibility for a to occur in that formula is that there is a rule in Π of the form $\neg\neg a \wedge \varphi \rightarrow a \vee \psi$. In fact, as we prove below, Definition 6 is a quite general, sufficient syntactic condition for the applicability of $\mathtt{f}_c$.

[8] In most cases, after unfolding $\mathtt{f}_c$ as a logic program, we usually obtain not only a result strongly equivalent to $\mathtt{f}_{sp}$ but also the same or a very close syntactic representation.

[9] In fact, the as-dual set from [4] can be seen as an effect of the (CUT) rule. Moreover, our use of the latter was inspired by this as-dual construction.

Theorem 2. *Let Π be a logic program for signature $\mathcal{AT}$, let $V \subseteq \mathcal{AT}$ and $a \in \mathcal{AT} \setminus V$ and let $\Pi' = behead^a(\Pi)$. If Π' is a-forgettable, then: $\Pi \cong_V \mathtt{f}_c(\Pi, a)$.* □

In our example, it is easy to see that this condition is satisfied because $behead^a(\Pi_1) = \Pi_1$ and this program does not contain a-choices.

4 Forgetting into forks: the *unfolding* operator

As we have seen, syntactic forgetting is limited to a family of transformation operators whose applicability can be analysed in terms of sufficient syntactic conditions. This method is incomplete in the sense that forgetting a in Π may be possible, but still the syntactic conditions we use for applicability may not be satisfied. Consider the following example.

Example 3. Take the following logic program Π_3:

$$\neg\neg a \to a \tag{20}$$
$$\neg a \to b \tag{21}$$
$$a \to c \tag{22}$$
$$b \to c \tag{23}$$
$$c \to b \tag{24}$$

This program does not fit into the a-forgettable syntactic form, but in fact we can forget a under strong persistence to obtain $b \wedge c$, as we will see later. □

If we look for a complete forgetting method, one interesting possibility is allowing the result to contain the fork operator. As proved in [1], forgettability as a fork is always guaranteed: that is, it is always possible to forget any atom if we allow the result to be in the general form of a fork. The method provided in [1] to obtain such a fork, however, was based on synthesis from the fork denotation, which deals with sets of sets of HT models. We propose next an always applicable syntactic method to obtain a fork as the result of forgetting any atom.

In the context of propositional logic, forgetting an atom a in a formula φ corresponds to the quantified Boolean formula $\exists a\, \varphi$ which, in turn, is equivalent to the unfolding $\varphi[a/\bot] \vee \varphi[a/\top]$. In the case of Equilibrium Logic, we will apply a similar unfolding but, instead of disjunction, we will use the fork connective, and rather than $\bot$ and $\top$ we will have to divide the cases into $\neg a$ and $\neg\neg a$, since $(\neg a \mid \neg\neg a) \equiv \top$. More precisely, using (6) and (7) from Proposition 1 we can build the chain of equivalences $\Pi \cong \Pi \wedge \top \cong \Pi \wedge (\neg a \mid \neg\neg a) \cong (\Pi \wedge \neg a \mid \Pi \wedge \neg\neg a)$. Then, by Proposition 2, we separate the task of forgetting a in Π into forgetting a in each one of these two branches, leading to:

Definition 7 (Unfolding operator, $\mathtt{f}_|$). *For any logic program Π and atom a we define:* $\mathtt{f}_|(\Pi, a) \stackrel{\text{def}}{=} (\ \mathtt{f}_c(\Pi \wedge \neg a, a)\ \mid\ \mathtt{f}_c(\Pi \wedge \neg\neg a, a)\ )$ □

Theorem 3. *Let Π be a logic program for signature $\mathcal{AT}$, let $V \subseteq \mathcal{AT}$ and $a \in \mathcal{AT} \setminus V$. Then, $\Pi \cong_V \mathtt{f}_|(\Pi, a)$.* □

Corollary 1. *If $a \notin V$ and Π is a-forgettable then $\mathtt{f}_{|}(\Pi, a) \cong_V \mathtt{f}_c(\Pi, a)$, and so, $\mathtt{f}_{|}(\Pi, a) \cong \mathtt{f}_c(\Pi, a)$.* □

Using (2) and (3), it is easy to prove:

Theorem 4. *For any logic program Π and atom a:*

$$\begin{aligned}\mathtt{f}_{|}(\Pi, a) &\cong (\ \mathtt{f}_c(\Pi[a/\bot] \wedge \neg a, a)\ \mid\ \mathtt{f}_c(\Pi[\neg a/\bot] \wedge \neg\neg a, a)\ ) \\ &\cong (\ \Pi[a/\bot]\ \mid\ \mathtt{f}_c(\Pi[\neg a/\bot] \wedge \neg\neg a, a)\ )\end{aligned}$$

This theorem provides a simpler application of the unfolding operator: the left branch, for instance, is now the result of replacing a by $\bot$. The right branch applies the cut operator, but introducing a prior step: we add the formula $\neg\neg a$ and replace all occurrences of $\neg a$ by $\bot$. It is easy to see that, in this previous step, any occurrence of a in the scope of negation is removed in favour of truth constants[10]. This means that the result has no a-choices since a will only occur in the scope of negation in the rule $\neg\neg a = (\neg a \to \bot)$. Therefore, the use of $\mathtt{f}_c$ in $\mathtt{f}_{|}$ is always applicable. Moreover, in many cases, we can use elementary HT transformations to simplify the programs $\Pi[a/\bot]$ and $\Pi[\neg a/\bot] \wedge \neg\neg a$, to look for a simpler application of $\mathtt{f}_c$, or to apply properties about the obtained fork.

As an illustration, consider again forgetting a in Π_3 and let us use the transformation in Theorem 4. We can observe that $\Pi_3[a/\bot]$ replaces (20), (21) and (22) respectively by $(\neg\neg\bot \to \bot)$ (a tautology), $(\neg\bot \to b) \equiv b$ and $(\bot \to c)$ (again, a tautology), leaving (23)-(24) untouched. To sum up, $\Pi_3[a/\bot] \equiv b \wedge (b \to c) \wedge (c \to b) \equiv (b \wedge c)$. On the other hand, $\Pi_3[\neg a/\bot]$ replaces (20) and (21) respectively by $(\neg\bot \to a) \equiv a$ and $(\bot \to b)$ (a tautology), so that $\Pi_3[\neg a/\bot] \wedge \neg\neg a$ amounts to the formula $a \wedge (a \to b) \wedge (b \to c) \wedge (c \to b) \wedge \neg\neg a$ which is equivalent to $a \wedge b \wedge c$ and, trivially, $\mathtt{f}_c(a \wedge b \wedge c, a) = (b \wedge c)$. Putting everything together, we get $\mathtt{f}_{|}(\Pi_3, a) \cong (\ b \wedge c \mid b \wedge c\ ) \cong (b \wedge c)$ since forks satisfy the idempotence property for '|' – see (11) from Proposition 12 in [1]. In this way, we have *syntactically* proved that a was indeed forgettable in Π_3 leading to $b \wedge c$ even though this program was not a-forgettable. We claim that the $\mathtt{f}_{|}$ operator plus the use of properties about forks (like the idempotence used above) opens a wider range of syntactic conditions under which forks can be reduced into formulas, and so, under which an atom can be forgotten in ASP.

An important advantage of the unfolding operator is that, since it is always applicable, it can be used to forget a set of atoms by forgetting them one by one. We illustrate this with another example.

Example 4. Suppose we want to forget atoms $\{a, b\}$ in the program $\Pi_4 \stackrel{\text{def}}{=} (20) \wedge (21) \wedge (22)$ where we simply removed (23) and (24) from Π_3.

This program is not a-forgettable, but nevertheless let us assume that we start forgetting a with the application of the unfolding $\mathtt{f}_{|}(\Pi_4, a)$. For the left hand side, we get that $\Pi_4[a/\bot] \equiv (\neg\neg\bot \to a) \wedge (\neg\bot \to b) \wedge (\bot \to c) \equiv b$ as we had seen before. Similarly, for the right hand side:

$$\Pi_4[\neg a/\bot] \wedge \neg\neg a = (\neg\bot \to a) \wedge (\bot \to b) \wedge (a \to c) \wedge \neg\neg a \cong a \wedge c$$

[10] Truth constants can be removed using trivial HT simplifications.

so the application of $\mathtt{f}_c$ becomes trivially $\mathtt{f}_c(a \wedge c, a) = c$ and the final result amounts to $\mathtt{f}_|(\Pi_4, a) = (b \mid c)$ that is, a fork of two atoms, which as discussed in [1], is (possibly the simplest case of) a fork that *cannot be reduced to a formula*. Still, we can use Proposition 2 to continue forgetting b in each of the two branches of $(b \mid c)$. As none of them contains b-choices, we can just apply $\mathtt{f}_c$ to obtain the fork $(\, \mathtt{f}_c(b, b) \mid \mathtt{f}_c(b, c)\,) = (\top \mid c)$ which, by (4), is equivalent to the formula $(\neg\neg c \rightarrow c)$. We end up with one more example.

Example 5. Suppose we want to forget q in the following program Π_5:

$$\neg\neg q \rightarrow q \qquad q \rightarrow u \qquad q \rightarrow s \qquad \neg q \rightarrow t$$

Although this program is not q-forgettable, it was included as Example 7 in [4] to illustrate the application of operator $\mathtt{f}_{sp}$. If we use $\mathtt{f}_|(\Pi_5, q)$, it is very easy to see that $\Pi_5[q/\bot] \cong t$ whereas $\Pi_5[\neg q/\bot] \wedge \neg\neg q \cong q \wedge u \wedge s$ so that we get $\mathtt{f}_|(\Pi_5, q) = (\, t \mid \mathtt{f}_c(q \wedge u \wedge s, q)\,) = (\, t \mid (u \wedge s)\,)$. This fork *cannot* be represented as a formula, since t and $u \wedge s$ have no logical relation and the fork is homomorphic to $(b \mid c)$ obtained before. In other words, atom q cannot be forgotten in Π_5 as a formula, and so, $\mathtt{f}_{sp}(\Pi_5, q)$ from [4] does not satisfy strong persistence.

5 Conclusions

We have presented a syntactic transformation, we called *unfolding*, that is always applicable on any logic program and allows forgetting an atom (under strong persistence), producing an expression that may combine the fork operator and propositional formulas. Unfolding relies on another syntactic transformation, we called the *cut* operator (close to $\mathtt{f}_{sp}$ from [4]), that can be applied on any program that does not contain choice rules for the forgotten atom and, unlike unfolding, it returns a propositional formula without forks. Although, in general, the forks we obtain by unfolding cannot be reduced to propositional formulas, we have also illustrated how the use of general properties of forks makes this possible sometimes, even in conditions where previous syntactic methods were not known to be applicable. Future work will be focused on extending the syntactic conditions under which forks can be reduced to formulas – we claim that this is an analogous situation to finding conditions under which second order quantifiers can be removed in second order logic. We will also study the extension of the unfolding operator to sets of atoms, instead of proceeding one by one.

Acknowledgements We want to thank the anonymous reviewers for their suggestions that helped to improve this paper. Partially funded by Xunta de Galicia and the European Union, grants CITIC (ED431G 2019/01) and GPC ED431B 2022/33, and by the Spanish Ministry of Science and Innovation (grant PID2020-116201GB-I00).

References

1. Aguado, F., Cabalar, P., Fandinno, J., Pearce, D., Pérez, G., Vidal, C.: Forgetting auxiliary atoms in forks. Artificial Intelligence 275, 575–601 (2019)

2. Aguado, F., Cabalar, P., Fandinno, J., Pérez, G., Vidal, C.: A logic program transformation for strongly persistent forgetting – extended abstract. In: Proc. of the 37th Intl. Conf. on Logic Programming (ICLP'21), Porto, Portugal (virtual event), Electronic Proceedings in Theoretical Computer Science (EPTCS), vol. 345, pp. 11–13 (2021)
3. Aguado, F., Cabalar, P., Pearce, D., Pérez, G., Vidal, C.: A denotational semantics for equilibrium logic. Theory and Practice of Logic Programming 15(4-5), 620–634 (2015)
4. Berthold, M., Gonçalves, R., Knorr, M., Leite, J.: A syntactic operator for forgetting that satisfies strong persistence. Theory and Practice of Logic Programming 19(5-6), 1038–1055 (2019)
5. Cabalar, P., Ferraris, P.: Propositional theories are strongly equivalent to logic programs. Theory and Practice of Logic Programming 7(6), 745–759 (2007)
6. Cabalar, P., Pearce, D., Valverde, A.: Reducing propositional theories in equilibrium logic to logic programs. In: Bento, C., Cardoso, A., Dias, G. (eds.) Proceedings of the 12th Portuguese Conference on Progress in Artificial Intelligence (EPIA'05). Lecture Notes in Computer Science, vol. 3808, pp. 4–17. Springer (2005)
7. Cabalar, P., Pearce, D., Valverde, A.: Minimal logic programs. In: Dahl, V., Niemelä, I. (eds.) Proceedings of the 23rd International Conference on Logic Programming, (ICLP'07). pp. 104–118. Springer (2007)
8. Ferraris, P., Lee, J., Lifschitz, V.: A generalization of the Lin-Zhao theorem. Annals of Mathematics and Artificial Intelligence 47(1-2), 79–101 (2006)
9. Gonçalves, R., Knorr, M., Leite, J.: The ultimate guide to forgetting in answer set programming. In: KR. pp. 135–144. AAAI Press (2016)
10. Gonçalves, R., Knorr, M., Leite, J.: You can't always forget what you want: On the limits of forgetting in answer set programming. In: Kaminka, G.A., Fox, M., Bouquet, P., Hüllermeier, E., Dignum, V., Dignum, F., van Harmelen, F. (eds.) Proceedings of 22nd European Conference on Artificial Intelligence (ECAI'16). Frontiers in Artificial Intelligence and Applications, vol. 285, pp. 957–965. IOS Press (2016)
11. Heyting, A.: Die formalen Regeln der intuitionistischen Logik. In: Sitzungsberichte der Preussischen Akademie der Wissenschaften, pp. 42–56. Deutsche Akademie der Wissenschaften zu Berlin (1930), reprint in Logik-Texte: Kommentierte Auswahl zur Geschichte der Modernen Logik, Akademie-Verlag, 1986.
12. Knorr, M., Alferes, J.J.: Preserving strong equivalence while forgetting. In: Fermé, E., Leite, J. (eds.) Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8761, pp. 412–425. Springer (2014)
13. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective. pp. 169–181. Springer-Verlag (1999)
14. Mints, G.: Cut-free formulations for a quantified logic of here and there. Annals of Pure and Applied Logic 162(3), 237–242 (2010)
15. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25, 241–273 (1999)
16. Pearce, D.: A new logical characterisation of stable models and answer sets. In: Dix, J., Pereira, L.M., Przymusinski, T.C. (eds.) Selected Papers from the Non-Monotonic Extensions of Logic Programming (NMELP'96). Lecture Notes in Artificial Intelligence, vol. 1216, pp. 57–70. Springer-Verlag (1996)