A Stronger Notion of Equivalence for Logic Programs

Ka-Shu Wong

University of New South Wales
and National ICT Australia
Sydney, Australia
kswong@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-0713
May 2007

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Several different notions of equivalence have been proposed for logic programs with answer set semantics, most notably strong equivalence. However, strong equivalence is not preserved by certain logic program operators such as the strong and weak forgetting operators of Zhang and Foo, in the sense that two programs which are strongly equivalent may no longer be strongly equivalent after the same operator is applied to both. We propose the stronger notion of T-equivalence which is designed to be preserved by logic program operators such as strong and weak forgetting. We give a syntactic definition of T-equivalence and provide a model-theoretic characterisation of T-equivalence using what we call T-models. We show that strong and weak forgetting does preserve T-equivalence and using this, arrive at a model-theoretic definition of the strong and weak forgetting operators using T-models.

1 Introduction

The answer set semantics for logic programs is a dialect of logic programming with negation-by-failure, which allows it to handle both defaults as well as incomplete information [4]. It has been used in applications as diverse as planning, nonmonotonic reasoning and game theory.

Several different notions of equivalence have been proposed for logic programs with answer set semantics. One way of doing this is to say that two logic programs P and Q are equivalent if P and Q have the same answer sets. However this is insufficient to capture, for example, the difference between the programs $\{a \leftarrow not\ b\}$ and $\{a\}$: both have the single answer set $\{a\}$, but behave differently when additional rules, for example $\{b\}$, are added. A stronger notion of equivalence which distinguishes these two programs is *strong equivalence*, first proposed by Lifschitz, Pearce and Valverde [5]: Two logic programs P and Q are strongly equivalent, if by adding any set of rules R to both P and Q, the resulting programs $P \cup R$ and $Q \cup R$ have the same answer sets. Strong equivalence naturally arises when we consider logic programs in a modular way: If we have a program P with a subprogram Q which is strongly equivalent to Q', then we may replace Q by Q' without affecting the answer sets of the resulting program.

However, for some applications strong equivalence is not strong enough. One case of this is with the strong and weak forgetting operators of Zhang and Foo [8], which are described in Section 4 of this paper. There are strongly equivalent logic programs which do not remain strongly equivalent after the same forgetting operators are applied to both: For example, the programs $P = \{a \leftarrow b; b \leftarrow a; c \leftarrow not\ a\}$ and $Q = \{a \leftarrow b; b \leftarrow a; c \leftarrow not\ b\}$ are strongly equivalent, however P after strong forgetting by a gives $P' = \{\}$ while the same operation on Q gives $Q' = \{c \leftarrow not\ b\}$. It is clear that P' and Q' are not strongly equivalent. In such cases we need a finer notion of equivalence.

In this paper we address this problem by introducing a stronger notion of equivalence which is preserved by the strong and weak forgetting operators. We call this relation *T-equivalence*, since it is preserved under logic program transformations such as forgetting. T-equivalence is defined syntactically using three inference rules which define a consequence relation. Two logic programs P and Q are then said to be T-equivalent if the set of consequences of P and Q are the same.

We give some background definitions in Section 2, followed by an overview of equivalence relations on logic programs in Section 3, and an introduction to forgetting in Section 4. In Section 5 we define T-equivalence, for which we also give a model-theoretic characterisation. Section 6 discusses the properties of T-equivalence in relation to the strong and weak forgetting operators, through which we arrive at a model-theoretic definition of these operators. Finally, we conclude in Section 7 with some discussion on future research directions.

2 Preliminaries

In this paper we deal with normal logic programs with negation-by-failure, where each rule is of the form

$$a \leftarrow b_1, b_2, \cdots, b_m,\ not\ c_1, not\ c_2, \cdots, not\ c_n$$

where $b_1, \cdots, b_m$ and $c_1, \cdots, c_n$ are from a set $\mathcal{A}$ of atoms, and a is either an element of $\mathcal{A}$ or $\bot$. We assume that the set of atoms $\mathcal{A}$ is fixed. Given a rule r in this form, we denote $H(r) = a$ (*head* of r), $B^+(r) = \{b_1, \cdots, b_m\}$ (*positive part* of r), and $B^-(r) = \{c_1, \cdots, c_n\}$ (*negative part* of r). A *positive logic program* is a logic program consisting of *positive rules*, which are rules with no negative part (i.e., $B^-(r)$ is empty). Note that we allow infinite sets of rules as logic programs, but each rule itself must be finite.

We adopt the convention of representing single atoms with lowercase letters (e.g. a, b, c) and sets of atoms with uppercase letters (e.g. A, B, X, Y). We use the following notation for writing rules with sets of atoms. Suppose $X = \{a, b\}$ and $Y = \{c, d\}$. Then the rule $p \leftarrow a, b, not\ c, not\ d$ may be written as $p \leftarrow X, not\ Y$.

For a set X of atoms and a logic program P, we use the notation $X \models P$ to mean that X is a model of P in the classical sense: For each $r \in P$, if $B^+(r) \subseteq X$ and $B^-(r) \cap X = \emptyset$, then $H(r) \in X$. Observe that under this definition, if $H(r)$ is $\bot$, then $X \models r$ if either $B^+(r) \not\subseteq X$ or $B^-(r) \cap X \neq \emptyset$. We say that X is a *minimal model* of P if X is minimal by set inclusion among all the models of P, i.e. $X \models P$ and there is no X' such that $X' \subset X$ and $X' \models P$.

The *Gelfond-Lifschitz reduct* [4] P^X of a program P with respect to a set of atoms X is defined by $P^X = \{H(r) \leftarrow B^+(r) \mid r \in P \text{ and } X \cap B^-(r) = \emptyset\}$. We say that X is an *answer set* of P if X is a minimal model of P^X.

Example 1. *Let P be the program $\{a \leftarrow not\ b; b\}$. $\{b\}$ is an answer set for P since $P^{\{b\}} = \{b\}$, which has the minimal model $\{b\}$. However, $\{b\}$ is not an answer set for $Q = \{a \leftarrow not\ b; b \leftarrow not\ b\}$ since $Q^{\{b\}} = \emptyset$, and although $\{b\}$ is a model for this program, it is not minimal.*

For a set of atoms $S \subseteq \mathcal{A}$, write S^c for the complement of S in $\mathcal{A}$.

3 Equivalences on Logic Programs

A number of different notions of equivalence on logic programs with answer set semantics have been studied in the literature. The most basic notion of equivalence says that two logic programs are equivalent if they have the same answer sets. We call this *ordinary equivalence*.

Ordinary equivalence does not distinguish between programs which have the same answer sets, but have different answer sets when additional rules are added. This can be a problem when we consider logic programs in a modular way by splitting them into components, as substituting a component program for an equivalent program may not give the intended results.

Example 2. *Consider the programs $P = \{a \leftarrow not\ b\}$ and $Q = \{a\}$. Both P and Q have the single answer set $\{a\}$. However after adding rules $R = \{b\}$, the program $P \cup R$ has answer set $\{b\}$, while $Q \cup R$ has answer set $\{a, b\}$.*

To capture the kind of equivalence that distinguishes between these programs, the notion of *strong equivalence* [5] was introduced. Strong equivalence describes the property that two programs remain equivalent regardless of what additional rules are added, and is defined as follows:

Definition 1. *Logic programs P and Q are strongly equivalent, iff for all sets of rules R, the programs $P \cup R$ and $Q \cup R$ have the same answer sets.*

Strong equivalence has been studied extensively in the literature [7, 5, 2, 6]. There is a model-theoretic characterisation of strong equivalence by Turner [7], who defined the notion of *SE-models* for logic programs and showed that two logic programs are strongly equivalent iff they have the same SE-models. The definition of SE-models is as follows:

Definition 2. *Let P be a logic program, and let X, Y be sets of atoms. We say the pair (X, Y) is a SE-model of P if $X \subseteq Y$, $Y \models P$ and $X \models P^Y$.*

Eiter and Fink [3] introduced the weaker notion of *uniform equivalence*. This is defined in a similar way to strong equivalence, but with the added set of rules R being restricted to sets of *facts*, which are rules with empty bodies.

Definition 3. *Logic programs P and Q are uniformly equivalent, if for all sets of facts F, the programs $P \cup F$ and $Q \cup F$ have the same answer sets.*

Example 3. *This example is taken from the paper by Eiter and Fink [3]. Let $P = \{a \leftarrow not\ b; a \leftarrow b\}$ and $Q = \{a \leftarrow not\ c; a \leftarrow c\}$. It can be verified that P and Q are uniformly equivalent. However P and Q are not strongly equivalent, since $\{a, b\}$ is an answer set of $Q \cup \{b \leftarrow a\}$ but not of $P \cup \{b \leftarrow a\}$.*

It was shown that uniform equivalence can be characterised using *UE-models*, which are defined as a refinement of SE-models:

Definition 4. *Let P be a logic program and (X, Y) be an SE-model of P. We say (X, Y) is a UE-model of P iff for every SE-model (X', Y), $X' \subset X$ implies $X' = Y$.*

4 Forgetting on Logic Programs

In applying logic programs to knowledge representation, there arises a need to allow updating of logic programs with conflicting information. One way of addressing conflicts is to weaken the logic program to remove the conflicting atom; the new information may then be added without creating inconsistencies.

Zhang and Foo [8] presented a pair of operators for removing an atom from a logic program with answer set semantics, called *strong forgetting* and *weak forgetting*. The idea behind the strong and weak forgetting operators is to remove an atom from the program while preserving as much information as possible. The difference between strong and weak forgetting is the way that negation is treated. The intuition is that in strong forgetting, the negation of the atom to be forgotten is treated as false, while in weak forgetting it is treated as true.

Definition 5. *Given logic program P and atom a, define $Reduct(P, a)$ to be the program consisting of rules in P plus rules which can be derived by unfolding on the atom a:*

$$Reduct(P, a) = P \cup \{H(r) \leftarrow B^+(r) \setminus a, B^+(s), not\ B^-(r), not\ B^-(s) \mid r, s \in P,\ a \in B^+(r),\ H(s) = a\}$$

We then define the strong forgetting operator $SForgetLP(P, a)$ by taking the program $Reduct(P, a)$ and then removing rules r for which r is valid, and rules r where $H(r) = a$ or $a \in B^+(r)$ or $a \in B^-(r)$:

Definition 6. *The operator $SForgetLP$ is defined as follows:*

$$\begin{aligned} SForgetLP(P,a) \;=\; & Reduct(P,a)\backslash \\ & (\;\{r \mid r \textit{ is valid}\} \;\cup\; \{r \mid H(r) = a\} \;\cup \\ & \;\{r \mid a \in B^+(r)\} \;\cup\; \{r \mid a \in B^-(r)\}\;) \end{aligned}$$

To construct the weak forgetting operator $WForgetLP(P,a)$, we take the program $Reduct(P,a)$ and remove rules r for which r is valid, and rules r where $H(r) = a$ or $a \in B^+(r)$. Finally each rule r where $a \in B^-(r)$ is replaced by a rule r' with a removed, i.e. $H(r') = H(r)$, $B^+(r') = B^+(r)$, and $B^-(r') = B^-(r) \setminus \{a\}$.

Definition 7. *$WForgetLP$ is defined in two steps. First we define $WForgetLP'(P,a)$:*

$$\begin{aligned} WForgetLP'(P,a) \;=\; & Reduct(P,a)\backslash \\ & (\;\{r \mid r \textit{ is valid}\} \;\cup\; \{r \mid H(r) = a\} \;\cup \\ & \;\{r \mid a \in B^+(r)\}\;) \end{aligned}$$

Then $WForgetLP(P,a)$ is given by:

$$\begin{aligned} WForgetLP(P,a) = \{H(r) \leftarrow B^+(r), not\ \{B^-(r) \setminus a\} \mid \\ r \in WForgetLP'(P,a)\} \end{aligned}$$

So far, we have defined strong and weak forgetting for single atoms. Forgetting with multiple atoms is defined as follows:

Definition 8. *Let $S = (s_1, \cdots, s_n)$ be an ordered sequence of atoms, and $()$ represent the empty sequence. $SForgetLP(P,S)$ is defined recursively as follows:*

$$\begin{aligned} SForgetLP(P,()) &= P \\ SForgetLP(P,(s_1,\cdots,s_n)) &= SForgetLP(SForgetLP(P,(s_1,\cdots,s_{n-1})),s_n) \end{aligned}$$

$WForgetLP(P,S)$ is defined in the same way.

Example 4. *Let P be the logic program $\{a \leftarrow b; b \leftarrow c; d \leftarrow c, not\ b\}$.*

$SForgetFP(P,b)$ gives the result $\{a \leftarrow c\}$. Notice that we have obtained a new rule $a \leftarrow c$ via "unfolding" of $a \leftarrow b$ and $b \leftarrow c$.

$WForgetLP(P,b)$ gives the result $\{a \leftarrow c; d \leftarrow c\}$. Here we see that strong forgetting and weak forgetting treats the rule $d \leftarrow c, not\ b$ differently. In strong forgetting, the rule is eliminated, however in weak forgetting, not b is removed from the rule instead.

The example below shows that strong equivalence is not preserved by strong forgetting or weak forgetting.

Example 5. *Consider the programs $P = \{a \leftarrow b; b \leftarrow a; c \leftarrow not\ a\}$ and $Q = \{a \leftarrow b; b \leftarrow a; c \leftarrow not\ b\}$, which we saw in the introduction. These two programs are strongly equivalent. However, $SForgetLP(P,a) = \{\}$ and $SForgetLP(Q,a) = \{c \leftarrow not\ b\}$ which are not strongly equivalent. In addition, $WForgetLP(P,a) = \{c\}$ and $WForgetLP(Q,a) = \{c \leftarrow not\ b\}$ which are also not strongly equivalent.*

5 T-Equivalence

In this section we introduce a new equivalence relation, which we call T-equivalence. This equivalence is a refinement of strong equivalence with the property that it is preserved by both strong and weak forgetting, which we show in Section 6.2.

5.1 Definition

We define T-equivalence by first constructing a consequence relation $\vdash$ on logic programs. We then say that two logic programs are equivalent if they have the same set of consequences. The formal definitions are given below:

Definition 9. *Define the consequence relation $\vdash$ on logic programs by the following inference rules:*

(C0) $P \vdash r$ *for every* $r \in P$ *and every* r *of the form* $a \leftarrow a$ *where* a *is an atom.*

(C1) *If* $P \vdash a \leftarrow B, not\ C$ *and* X, Y *are sets of atoms, then* $P \vdash a \leftarrow B, X, not\ C, not\ Y$.

(C2) *If* $P \vdash a \leftarrow B_1, x, not\ C_1$ *and* $P \vdash x \leftarrow B_2, not\ C_2$, *then* $P \vdash a \leftarrow B_1, B_2, not\ C_1, not\ C_2$.

The inference rule (C0) states that rules in P and valid rules of the form $a \leftarrow a$ can be immediately derived. (C1) allows for rules to be weakened by adding additional atoms to the body, while (C2) is essentially an unfolding [1] operation. Note that any valid rule (rules where the same atom appears in both the head and in the positive part) can be derived by using (C0) followed by (C1).

We say that a rule r is a *consequence* of P, written $P \vdash r$, if it can be derived using a finite number of applications of (C0)–(C2). Let $Cn(P)$ denote the set of consequences of P, defined by $Cn(P) = \{r \mid P \vdash r\}$. T-equivalence is defined as follows:

Definition 10. *Logic programs P, Q are T-equivalent if $Cn(P) = Cn(Q)$.*

Note that if P and Q are T-equivalent then P and Q are strongly equivalent. This follows from the results of Section 5.2.

To simplify the proofs in this paper, we now introduce a notation for representing the steps taken to derive a rule r using the inference rules:

Definition 11. *Let P be a logic program, and define* C1 *and* C2 *as follows:*

$$\mathrm{C1}(r, X, Y) = H(r) \leftarrow B^+(r), X, not\ B^-(r), not\ Y$$

for rule r and sets of atoms X and Y, and

$$\mathrm{C2}(r, s) = H(r) \leftarrow B^+(r) \setminus H(s), B^+(s), not\ B^-(r), not\ B^-(s)$$

for rules r, s with $H(s) \in B^+(r)$.

$\mathrm{C1}(r, X, Y)$ is the result of applying (C1) to the rule r using the sets of atoms X and Y, and $\mathrm{C2}(r, s)$ is the result of applying (C2) to the rules r and s. Therefore every consequence of P can be written as an expression using the operators C1 and C2 and rules satisfying (C0). We call such an expression for r a *derivation* of r from P. The derivations of r from P with the smallest number of occurrences of C1 and C2 are called the *minimum derivations* of r from P.

Example 6. *Consider the program* $P = \{r_1, r_2, r_3\}$ *where* $r_1 = a \leftarrow d$, $r_2 = a \leftarrow b, c$ *and* $r_3 = b \leftarrow d, not\ e$. *Then the rule* $r_4 = a \leftarrow c, d, f, not\ e$ *is in* $Cn(P)$ *and can be written as the derivation* $\mathrm{C1}(\mathrm{C2}(r_2, r_3), \{f\}, \emptyset)$. *However this is not the minimum derivation for* r_4, *as it can also be written as* $\mathrm{C1}(r_1, \{c, f\}, \{e\})$.

5.2 Model-Theoretic Characterisation

Here we introduce a characterisation of T-equivalence in model-theoretic terms. Recall from Section 3 that strong equivalence is characterised by SE-models [7] and uniform equivalence is characterised by UE-models [3]. It turns out that T-equivalence can also be characterised in a similar way:

Definition 12. *Let* P *be a logic program, and let* X, Y *be sets of atoms. We say the pair* (X, Y) *is a T-model of* P *if* $X \models P^Y$. *Let* $M(P)$ *denote the set of all T-models of* P.

Notice the similarity of this to the definition for SE-models. In particular, observe that (X, Y) is an SE-model iff it is a T-model that satisfies the additional properties $X \subseteq Y$ and $Y \models P$. A consequence of this is that P is strongly equivalent to $Cn(P)$. This is because P and $Cn(P)$ have the same T-models as shown below, and since $Y \models P$ iff $Y \models Cn(P)$, we see that P and $Cn(P)$ must have the same SE-models.

Example 7. *Let* P *be the logic program* $\{a \leftarrow not\ b\}$. *Pairs of the form* $(\{a\} \cup X, Y)$ *or* $(X, \{b\} \cup Y)$ *where* $X, Y \subseteq \{a, b\}$ *are T-models of* P. $(\{b\}, \{a\})$, $(\{\}, \{a\})$, $(\{b\}, \{\})$, *and* $(\{\}, \{\})$ *are not T-models of* P.

The following result shows that T-models capture the notion of T-equivalence, in the same way that SE-models capture strong equivalence and UE-models capture uniform equivalence:

Theorem 1. *Let* P, Q *be logic programs. Then* P *and* Q *are T-equivalent iff* $M(P) = M(Q)$.

For the proof of Theorem 1, we introduce a new notation to denote the set of consequences restricted to positive rules. For positive programs P, we define Cn^+ as follows:

$$Cn^+(P) = \{r \mid r \in Cn(P), B^-(r) = \emptyset\}$$

Lemma 1. *Let* P *be a logic program, and* X, Y *be sets of atoms. Then*

1. $X \models P$ *iff* $X \models Cn(P)$. *If* P *is positive,* $X \models P$ *iff* $X \models Cn^+(P)$.
2. $Cn^+(P^Y) = Cn(P)^Y$
3. $M(P) = M(Cn(P))$

Lemma 2. *Suppose* P, Q *are positive logic programs. Then* $Cn^+(P) = Cn^+(Q)$ *iff* P *and* Q *have the same classical models.*

Proof of Lemma 1.

1. We first show that $X \models P$ iff $X \models Cn(P)$. The if case follows immediately from $P \subseteq Cn(P)$. For the other direction, suppose $X \models P$. To show that $X \models Cn(P)$, it suffices to show that the inference rules preserve $\models$. Thus it is enough to show the following:

   (0) $X \models x \leftarrow x$ for any atom x.

   (1) If $X \models r$, then $X \models \mathrm{C1}(r, A, B)$ for any sets of atoms A, B.

   (2) If $X \models \{r, s\}$, then $X \models \mathrm{C2}(r, s)$.

   The proofs are straightforward and are omitted.

   For the positive case, observe that $Cn(Cn^+(P)) = Cn(P)$, so $X \models P$ iff $X \models Cn(P)$ iff $X \models Cn^+(P)$.

2. We first show that $Cn^+(P^Y) \subseteq Cn(P)^Y$. Suppose $r \in Cn^+(P^Y)$. We show $r \in Cn(P)^Y$ by induction on the number of occurrences of C1 and C2 in the expression for r. The base case is where r satisfies (C0). Then either r is of the form $a \leftarrow a$, in which case it is in $Cn(P)^Y$, or $r \in P^Y$, in which case it is the reduct of some rule r' in P, and since $r' \in Cn(P)$ we have $r \in Cn(P)^Y$. For the inductive step, r is produced by either (C1) or (C2). If (C1), then $r = \mathrm{C1}(s, A, B)$ for some rule $s \in Cn(P^Y)$, and B and $B^-(s)$ are both empty, since $B^-(r)$ is empty. Therefore $s \in Cn^+(P^Y)$ and so $s \in Cn(P)^Y$ by the induction hypothesis. Now s is the reduct of some rule $s' \in Cn(P)$. Therefore r is the reduct of $\mathrm{C1}(s', A, \emptyset)$ which is in $Cn(P)$ and hence $r \in Cn(P)^Y$. If (C2), then $r = \mathrm{C2}(s, t)$ for some rules $s, t \in Cn(P^Y)$, then $B^-(s)$ and $B^-(t)$ are both empty and so the induction hypothesis gives us $s, t \in Cn(P)^Y$. Let s be the reduct of $s' \in Cn(P)$ and t the reduct of $t' \in Cn(P)$. Then $r' = \mathrm{C2}(s', t')$ has the properties $H(r') = H(r)$, $B^+(r') = B^+(r)$, and $B^-(r') = B^-(s') \cup B^-(t')$. Since both $B^-(s') \cap Y$ and $B^-(t') \cap Y$ are empty, the reduct of r' exists and is r.

   Now we show that $Cn(P)^Y \subseteq Cn^+(P^Y)$. Suppose $r \in Cn(P)^Y$, i.e. r is the reduct of a rule r' in $Cn(P)$. We show $r \in Cn^+(P^Y)$ by induction on the number of occurrences of C1 and C2 in the expression for r'. The base case is where r' satisfies (C0). Then either r' is of the form $a \leftarrow a$, in which case it is the same as r and is in $Cn^+(P^Y)$, or $r' \in P$, in which case its reduct r is in P^Y and hence in $Cn^+(P^Y)$. For the inductive step, r' is produced by either (C1) or (C2). If (C1), then $r' = \mathrm{C1}(s', A, B)$ for some $s' \in Cn(P)$. Since the reduct of r' exists and $B^-(s') \subseteq B^-(r')$, the reduct s of s' also exists and is in $Cn(P)^Y$. By the inductive hypothesis, $s \in Cn^+(P^Y)$. Now $r = \mathrm{C1}(s, A, \emptyset)$, which shows $r \in Cn^+(P^Y)$. If (C2), then $r' = \mathrm{C2}(s', t')$ for some $s', t' \in Cn(P)$. Since the reduct of r' exists, $B^-(r') \cap Y$ is empty and so $B^-(s') \cap Y$ and $B^-(t') \cap Y$ are also empty. Therefore the reducts s of s' and t of t' both exist and are in $Cn(P)^Y$, and by the induction hypothesis, are also in $Cn^+(P^Y)$. We have $r = \mathrm{C2}(r, s)$ and so $r \in Cn^+(P^Y)$.

3. $(X, Y) \in M(P)$ iff $X \models P^Y$ iff $X \models Cn^+(P^Y)$ iff $X \models Cn(P)^Y$ iff $(X, Y) \in M(Cn(P))$. □

Proof of Lemma 2 (sketch). The only if case is trivial. For the if case, we note that the classical consequences of a set of clauses are fully determined by the set

of classical models. Therefore the result follows from the following fact: If the rule $a \leftarrow b_1, \cdots, b_n$ is a classical consequence of P, then this rule is in $Cn^+(P)$.

To show this, consider the program consisting of P plus the rules $b_1, \cdots, b_n$, and $\bot \leftarrow a$. This is an inconsistent program. Because resolution is complete for programs of this form, we can produce a refutation tree. This tree can be thought of as a sequence of applications of (C2) on the rules of P plus the added rules to produce a rule with empty head and body. We then remove the added rules from the tree. The resulting tree describes a sequence of applications of (C2) on the rules of P to produce a rule with a as the head and a subset of $b_1, \cdots, b_n$ as the body. An application of (C1) gives us the rule that we need.

The case where $a = b_i$ for some i needs to be consider separately, as the tree produced becomes degenerate and does not actually represent a sequence of applications of (C2); but (C0) followed by (C1) gives us the rule needed. □

Proof of Theorem 1. The only if case follows from Lemma 1. For the if case, let P, Q be logic programs and suppose $M(P) = M(Q)$. We show that $Cn(P) = Cn(Q)$ by partitioning $Cn(P)$ according to the negative parts of the rules.

For any logic program P, define $[P]_S = \{H(r) \leftarrow B^+(r) \mid r \in P, B^-(r) = S\}$. Thus $[P]_S$ is made by taking the rules from P with negative part S, and removing the negative parts from these rules to form a positive logic program. This is related to the reduct operation via the following fact: $P^X = \bigcup_{S \subseteq X^c} [P]_S$.

$[Cn(P)]_S$ has the property that $[Cn(P)]_S \subseteq [Cn(P)]_T$ if $S \subseteq T$. This is true because (C1) allows us to add arbitrary atoms to the body of rules, and in particular to the negative part of rules. This shows that $[Cn(P)]_S = Cn(P)^{S_c}$, and hence the classical models of $[Cn(P)]_S$ are exactly $\{X \mid (X, S^c) \in M(P)\}$.

Using Lemma 2 and the fact that $[Cn(P)]_S$ is closed under Cn^+, this shows that $[Cn(P)]_S = [Cn(Q)]_S$ for every set of atoms S. Therefore $Cn(P) = Cn(Q)$, which proves the result. □

Example 8. *Reconsider the programs $P = \{a \leftarrow b; b \leftarrow a; c \leftarrow not\ a\}$ and $Q = \{a \leftarrow b; b \leftarrow a; c \leftarrow not\ b\}$. In Section 1 we saw that P and Q are strongly equivalent. However, P and Q are not T-equivalent because $(\emptyset, \{a\})$ is a T-model of P but not of Q. $(\emptyset, \{a\})$ is not a SE-model of P because $\{a\} \not\models P$.*

Now consider the program $P = \{p \leftarrow q, not\ q\}$. It can be easily verified that this program is strongly equivalent to the empty program. However, it is not T-equivalent to the empty program, since $(\{q\}, \emptyset)$ is a T-model of the empty program but not of P. $(\{q\}, \emptyset)$ is not a SE-model of the empty program because $\{q\} \not\subseteq \emptyset$.

6 Forgetting and T-Equivalence

It has been argued that strong equivalence should be the canonical notion of equivalence for logic programs, and hence the fact that strong equivalence is not preserved under strong and weak forgetting indicates a problem with the forgetting operators rather than a need for a new notion of equivalence. We believe that studying the properties of T-equivalence under strong and weak forgetting will provide some insights to how a forgetting operator that preserves strong equivalence can be constructed. Furthermore, strong and weak forgetting are currently used as mechanisms in modelling negotiation between agents with

intentions represented as logic programs. T-equivalence may thereby provide insights into canonical representations of agent intentions.

In this section, we consider the properties of T-equivalence under the strong and weak forgetting operators described in Section 4. Notice that strong and weak forgetting operators are defined syntactically. A model-theoretic definition of strong and weak forgetting would give rise to an equivalence relation on logic programs, defined by P and Q being equivalent iff they have the same models. Such an equivalence relation would satisfy certain properties in relation to strong and weak forgetting, in particular that the equivalence relation is preserved by the forgetting operators. Here we show that T-equivalence does satisfy these properties, and that the model-theoretic characterisation of T-equivalence gives us a way of defining strong and weak forgetting in model-theoretic terms.

6.1 Equivalence Relations Preserved by Forgetting

We define a number of postulates which encode desirable properties for an equivalence relation on logic programs in relation to a forgetting operator.

Let F denote either the strong or weak forgetting operator. Let P be a logic program, and S an ordered sequence of atoms. We write $F(P, S)$ for the program obtained from P by applying the forgetting operation with each element of S in order. If S is the empty sequence, then $F(P, S)$ is P. Let $\sim$ be an equivalence relation on logic programs, and consider the following postulates:

(1) $F(P, S) \sim F(P, \pi(S))$ for any permutation π

(2) $P \sim Q \Rightarrow F(P, S) \sim F(Q, S)$

(3) $P \sim Q \Rightarrow P$ strongly equivalent to Q

The intuition behind (1) is that we want the result of forgetting to be independent of the order in which atoms are forgotten. (2) expresses the property that equivalent programs remain equivalent after the same forgetting operation is applied to both. We also want the equivalence relation to be a strengthening of strong equivalence; this is expressed in (3).

6.2 T-Equivalence

T-equivalence satisfies the postulates listed in Section 6.1, as shown below.

Theorem 2. *For both strong forgetting and weak forgetting, T-equivalence satisfies postulates (1)–(3).*

Lemma 3.

- $r \in SForgetLP(Cn(P), a)$ *iff* $r \in Cn(P)$, r *is not a valid rule, and* r *does not contain* a.
- $r \in WForgetLP(Cn(P), a)$ *iff* r *does not contain* a, r *is not a valid rule, and there exists* $r' \in Cn(P)$ *such that* $H(r) = H(r')$, $B^+(r) = B^+(r')$ *and* $B^-(r) = B^-(r') \setminus \{a\}$.

Proof of Lemma 3. We first show that $Reduct(Cn(P), a) = Cn(P)$: Since $s, t \in Cn(P)$ implies $\mathrm{C2}(s, t) \in Cn(P)$, the rules added by the reduct operation are in fact already in $Cn(P)$, and so $Reduct(Cn(P), a) = Cn(P)$. The remainder of the proof is straightforward from the definitions of $SForgetLP$ and $WForgetLP$. □

Proof of Theorem 2. Let $\sim$ denote T-equivalence. We will prove that postulates (1)–(3) holds for the strong forgetting case. The proof for the weak forgetting case is similar.

In the following proof, F denotes the strong forgetting operator $SForgetLP$.

(1) It is sufficient to show that if $P \sim P'$, then $F(P, (a, b)) \sim F(P', (b, a))$ since it is possible to obtain any permutation of a sequence by swapping adjacent pairs. In fact, if we assume postulate (2), it is enough to prove

$$F(P, (a, b)) \sim F(P, (b, a))$$

Suppose $r \in F(P, (a, b))$. We need to show that $F(P, (b, a)) \vdash r$.

Since F is the strong forgetting operator, r must be in P or be derived from P via the *Reduct* operator. We consider separate cases based on how r is derived from P:

- r is in P. It is clear that r cannot contain a or b. Therefore $r \in F(P, (b, a))$.
- $r = \mathrm{C2}(r_1, r_2)$ for some r_1, r_2 in P where $H(r_2) = a$. Since r cannot contain b, r_1 and r_2 both do not contain b. hence $r_1, r_2 \in F(P, b)$ and so $r \in F(P, (b, a))$
- $r = \mathrm{C2}(r_1, r_2)$ for some r_1, r_2 in P where $H(r_2) = b$. In this case, $r \in F(P, b)$. Since r does not contain a, we get $r \in F(P, (b, a))$.
- $r = \mathrm{C2}(r_1, \mathrm{C2}(r_2, r_3))$ for some r_1, r_2, r_3 in P where $H(r_3) = a$ and $H(r_2) = b$. Now $r = \mathrm{C2}(\mathrm{C2}(r_1, r_2), r_3))$. We have $\mathrm{C2}(r_1, r_2) \in F(P, b)$ and r_3 is also in $F(P, b)$, so $r \in F(P, (b, a))$ as required.
- $r = \mathrm{C2}(\mathrm{C2}(r_1, r_2), r_3)$ for some r_1, r_2, r_3 in P where $H(r_2) = a$ and $H(r_3) = b$. If r_1 contains b but not r_2, then $r = \mathrm{C2}(\mathrm{C2}(r_1, r_3), r_2)$, and $\mathrm{C2}(r_1, r_3), r_2$ are both in $F(P, b)$, so $r \in F(P, (b, a))$. Similarly, if r_2 contains b but not r_1, then $r = \mathrm{C2}(r_1, \mathrm{C2}(r_2, r_3))$, and $\mathrm{C2}(r_2, r_3), r_1$ are both in $F(P, b)$, so $r \in F(P, (b, a))$. Finally, if r_1 and r_2 both contain b, then $r = \mathrm{C2}(\mathrm{C2}(r_1, r_3), \mathrm{C2}(r_2, r_3))$, $\mathrm{C2}(r_1, r_3), \mathrm{C2}(r_2, r_3)$ are both in $F(P, b)$ and so $r \in F(P, (b, a))$.

(2) Suppose $F(Cn(P), a) \subseteq Cn(F(P, a))$. Then $Cn(F(Cn(P), a)) = Cn(F(P, a))$ and so $F(Cn(P), a) \sim F(P, a)$, from which postulate (2) follows. Therefore it suffices to prove $F(Cn(P), a) \subseteq Cn(F(P, a))$.

Suppose $r \in F(Cn(P), a)$. Then $r \in Cn(P)$ by Lemma 3, so there is a way of deriving r from P. We show $r \in Cn(F(P, a))$ by induction on the size of the minimum derivation for r from P in terms of the number of applications of C1 and C2. The proof works by examining the minimum derivation of r from P. We show that some of the subexpressions of the minimum derivation are in $Cn(F(P, a))$, by showing that the rule given by the subexpression is in

$F(Cn(P,a))$ and then applying the induction hypothesis. Note that if the rule happens to be a valid rule, it is not in $F(Cn(P),a)$, but it is in $Cn(F(P,a))$ because it contains all valid rules. Therefore to show that it is in $Cn(F(P,a))$, we only need to check that the subexpression does not contain a.

We consider separate cases based on the form of the minimum derivation of r from P:

- If r is derived using (C0), the minimum derivation of r is r itself and does not contain C1 or C2. Since r cannot be of the form $x \leftarrow x$, as the forgetting operator removes valid rules, we know that r must be in P. We also know that r does not contain a. Therefore $r \in F(P,a)$, and hence $r \in Cn(F(P,a))$.

- $r = \mathrm{C1}(s, A, B)$

  We need to check that s does not contain a. Since r cannot contain a, s does not contain a either. Therefore $r \in Cn(F(P,a))$.

- $r = \mathrm{C2}(s,t)$

  If neither s nor t contain a, then $s, t \in Cn(F(P,a))$, and so $r \in Cn(F(P,a))$.

  On the other hand, if either s or t or both contain a, then since the resulting rule does not contain a, $H(t)$ must be a and $B^+(s)$ must contain a.

  Furthermore, if s,t are both in P, then $\mathrm{C2}(s,t)$ is in $\mathit{Reduct}(P,a)$ and hence in $Cn(F(P,a))$.

  In the cases below we assume $H(t) = a$ and $a \in B^+(s)$, and that one of s or t is derived and not in P. Note that minimality prevents either s or t from being a rule of the form $x \leftarrow x$ derived using (C0).

- $r = \mathrm{C2}(\mathrm{C1}(x, A, B), t)$

  We may assume that $A \cap B^+(x)$ and $B \cap B^-(x)$ are empty.

  If $B^+(x)$ contains a, then

  $$\mathrm{C2}(\mathrm{C1}(x, A, B), t) = \mathrm{C1}(\mathrm{C2}(x,t), A, B)$$

  Since $\mathrm{C2}(x,t)$ does not contain a, therefore $\mathrm{C2}(x,t) \in Cn(F(P,a))$, showing that $r \in Cn(F(P,a))$.

  If $B^+(x)$ does not contain a, then we must have $a \in A$. Now x does not contain a, so $x \in Cn(F(P,a))$. It is easy to see that

  $$\mathrm{C2}(\mathrm{C1}(x, A, B), t) = \mathrm{C1}(x, A \setminus \{a\} \cup B^+(t), B \cup B^-(t))$$

  thus showing r is in $Cn(F(P,a))$.

- $r = \mathrm{C2}(\mathrm{C2}(x,y), t)$

  The body of x or y, or both, must contain a.

  If $a \in B^+(x)$ and $a \notin B^+(y)$, then $\mathrm{C2}(x,t)$ exists and does not contain a. Therefore $\mathrm{C2}(x,t)$ and y are both in $Cn(F(P,a))$. If $H(y) \notin B^+(t)$, then

  $$\mathrm{C2}(\mathrm{C2}(x,y), t) = \mathrm{C2}(\mathrm{C2}(x,t), y)$$

which shows $r \in Cn(F(P,a))$. On the other hand, if $H(y) \in B^+(t)$, then $\mathrm{C2}(\mathrm{C2}(x,t),y)$ differs from r in that it does not have $H(y)$ in its body, but r does. In this case we have

$$\mathrm{C2}(\mathrm{C2}(x,y),t) = \mathrm{C1}(\mathrm{C2}(\mathrm{C2}(x,t),y),\{H(y)\},\emptyset)$$

If $a \notin B^+(x)$ and $a \in B^+(y)$, then $\mathrm{C2}(y,t)$ exists and does not contain a. Therefore both $\mathrm{C2}(y,t)$ and x are in $Cn(F(P,a))$. We have

$$\mathrm{C2}(\mathrm{C2}(x,y),t)) = \mathrm{C2}(x,\mathrm{C2}(y,t))$$

If $a \in B^+(x)$ and $a \in B^+(y)$, then $\mathrm{C2}(x,t)$ and $\mathrm{C2}(y,t)$ both exist and do not contain a. Hence both $\mathrm{C2}(x,t)$ and $\mathrm{C2}(y,t)$ are in $Cn(F(P,a))$. In this case

$$\mathrm{C2}(\mathrm{C2}(x,y),t) = \mathrm{C2}(\mathrm{C2}(x,t),\mathrm{C2}(y,t))$$

- $r = \mathrm{C2}(s,\mathrm{C1}(x,A,B))$

  We know that a is the head of $\mathrm{C1}(x,A,B)$. So $H(x) = a$ and $a \in B^+(s)$, and hence $\mathrm{C2}(s,x)$ exists. We have

  $$\mathrm{C2}(s,\mathrm{C1}(x,A,B)) = \mathrm{C1}(\mathrm{C2}(s,x),A,B)$$

  It is clear that $\mathrm{C2}(s,x)$ does not contain a, since r does not contain a, therefore $\mathrm{C2}(s,x) \in Cn(F(P,a))$, and so $r \in Cn(F(P,a))$.

- $r = \mathrm{C2}(s,\mathrm{C2}(x,y))$

  The head of $\mathrm{C2}(x,y)$ is a, so $H(x) = a$.

  If $B^+(x)$ does not contain a, then $\mathrm{C2}(s,x)$ does not contain a, so $\mathrm{C2}(s,x)$ is in $Cn(F(P,a))$, by the induction hypothesis. In addition, y does not contain a, since $a \notin B^+(x)$ which implies that the a is not the head of y, and a is not in r, showing that a is not in the body of y. Therefore y is also in $Cn(F(P,a))$. If $H(y) \notin B^+(s)$, then

  $$\mathrm{C2}(s,\mathrm{C2}(x,y)) = \mathrm{C2}(\mathrm{C2}(s,x),y)$$

  However if $H(y) \in B^+(s)$, then $\mathrm{C2}(\mathrm{C2}(s,x),y)$ differs from r in that $H(y)$ is missing from the body. In this case

  $$\mathrm{C2}(s,\mathrm{C2}(x,y)) = \mathrm{C1}(\mathrm{C2}(\mathrm{C2}(s,x),y),\{H(y)\},\emptyset)$$

  If $B^+(x)$ contains a, then $H(y)$ must be a, since $B^+(r)$ does not contain a. Now we know that $H(y) = a$ and $a \in B^+(s)$, so $\mathrm{C2}(s,y)$ exists. Furthermore, it does not contain a since $B^+(y)$ does not contain a, so we can conclude that $\mathrm{C2}(s,y)$ is in $Cn(F(P,a))$. Therefore

  $$\mathrm{C2}(s,\mathrm{C2}(x,y)) = \mathrm{C1}(\mathrm{C2}(s,y),B^+(x) \setminus \{a\},B^-(x))$$

(3) Suppose P and Q are T-equivalent. P and Q have the same T-models, so they have the same SE-models, and hence they are strongly equivalent. □

In addition to postulates (1)–(3), we may also want to add a postulate that limits how strong the equivalence relation can be:

(4) $P \not\sim Q \Rightarrow \exists S$ such that $F(P,S)$ not strongly equivalent to $F(Q,S)$

Postulate (4) expresses the notion that the equivalence relation should not be stronger than necessary; if two programs are not equivalent then there should be some chain of forgettings after which they are not strongly equivalent. Notice that (2) and (3) together implies the converse of (4). T-equivalence does not satisfy this postulate; however this postulate is not necessary for the conclusions below.

6.3 Forgetting on T-Models

Using the results of the previous section, we can construct a definition of strong and weak forgetting in model-theoretic terms. We want to define a pair of operators, $SForgetM$ and $WForgetM$, on sets of models, with the property that the models of a logic program P after strong (resp. weak) forgetting is given by applying $SForgetM$ (resp. $WForgetM$) to the models of P.

Definition 13. *The operators $SForgetM$ and $WForgetM$ are defined by:*

$$\begin{aligned} SForgetM(M,a) &= \{(X,Y) \mid (X \setminus \{a\}, Y \cup \{a\}) \in M \text{ or} \\ &\qquad (X \cup \{a\}, Y \cup \{a\}) \in M\} \\ WForgetM(M,a) &= \{(X,Y) \mid (X \setminus \{a\}, Y \setminus \{a\}) \in M \text{ or} \\ &\qquad (X \cup \{a\}, Y \setminus \{a\}) \in M\} \end{aligned}$$

where $M = M(P)$ for some logic program P.

We claim that this definition of $SForgetM$ and $WForgetM$ has the required properties, which can be stated formally as follows:

Proposition 1. *Let P be a logic program. The $SForgetM$ and $WForgetM$ operators satisfy:*

$$\begin{aligned} SForgetM(M(P),a) &= M(SForgetLP(P,a)) \\ WForgetM(M(P),a) &= M(WForgetLP(P,a)) \end{aligned}$$

Proof. To show that $SForgetM$ and $WForgetM$ satisfy Proposition 1 we need to show

$$\begin{aligned} (X,Y) \models SForgetLP(P,a) \quad \text{iff} \quad & (X \setminus \{a\}, Y \cup \{a\}) \models P \text{ or} \\ & (X \cup \{a\}, Y \cup \{a\}) \models P \\ (X,Y) \models WForgetLP(P,a) \quad \text{iff} \quad & (X \setminus \{a\}, Y \setminus \{a\}) \models P \text{ or} \\ & (X \cup \{a\}, Y \setminus \{a\}) \models P \end{aligned}$$

for all logic programs P. Since P is T-equivalent to $Cn(P)$, postulate (2) tells us that $M(SForgetLP(P,a)) = M(SForgetLP(Cn(P),a))$ and $M(WForgetLP(P,a)) = M(WForgetLP(Cn(P),a))$. Thus we may assume that P is closed under Cn.

(Strong forgetting case) Since P is closed under Cn, $SForgetLP(P, a)$ is the set of all rules $r \in P$ which do not contain a.

For the only-if case, suppose $(X, Y) \models SForgetLP(P, a)$. Firstly let us consider the case where there is no rule of the form $a \leftarrow B, not\ C$ in P such that $B \subseteq X \setminus \{a\}$ and $C \cap (Y \cup \{a\})$ is empty. In this case we show that $(X \setminus \{a\}, Y \cup \{a\}) \models P$. For each rule $r \in P$,

- If r does not contain a, then r is satisfied because $(X, Y) \models r$.
- If $a = H(r)$ then the body cannot be satisfied because of our assumption.
- If $a \in B^-(r)$ then r is eliminated in the reduct, hence it is satisfied.
- If $a \in B^+(r)$ then the body is satisfied because a is not in the model.

Now let us consider the case where there are rules of the form $a \leftarrow B, not\ C$ in P such that $B \subseteq X \setminus \{a\}$ and $C \cap (Y \cup \{a\})$ is empty. In this case we show that $(X \cup \{a\}, Y \cup \{a\})$ satisfies every rule $r \in P$.

- If r does not contain a, then r is satisfied because $(X, Y) \models r$.
- If $a = H(r)$ then r is satisfied since a is included in the model.
- If $a \in B^-(r)$, then r is eliminated in the reduct, hence it is satisfied.
- Suppose $a \in B^+(r)$. We will show that r is satisfied using contradiction. Suppose r is not satisfied, i.e. $H(r) \notin X \cup \{a\}$, $B^+(r) \subseteq X \cup \{a\}$, and $B^-(r) \cap (X \cup \{a\})$ is empty. From our initial assumption there is a rule $a \leftarrow B, not\ C$ such that $B \subseteq X \setminus \{a\}$ and $C \cap (Y \cup \{a\})$ is empty; call this rule s. We have $\mathrm{C2}(r, s) \in P$ since P is closed under Cn. This rule does not contain a, so $(X, Y) \models \mathrm{C2}(r, s)$. But this contradicts $H(r) \notin X \cup \{a\}$.

Therefore we have shown the only-if case. For the other direction, suppose either $(X \setminus \{a\}, Y \cup \{a\}) \models P$ or $(X \cup \{a\}, Y \cup \{a\}) \models P$. Then for each $r \in SForgetLP(P, a)$, since $r \in P$ and r does not contain a, we get $(X, Y) \models r$.

(Weak forgetting case) $WForgetLP(P, a)$ consists of rules $r \in P$ which do not contain a, plus the rule $H(r) \leftarrow B^+(r), not\ (B^-(r) \setminus \{a\})$ for each rule $r \in P$ which only contain a in the negative part.

For the only-if case, suppose $(X, Y) \models WForgetLP(P, a)$. We first consider the case where there is no rule of the form $a \leftarrow B, not\ C$ in P such that $B \subseteq X \setminus \{a\}$ and $C \cap (Y \setminus \{a\})$ is empty. For this case we show that $(X \setminus \{a\}, Y \setminus \{a\})$ satisfies P, by showing that each $r \in P$ is satisfied.

- If r does not contain a then r is satisfied because $(X, Y) \models r$.
- If $a = H(r)$, then r is satisfied because the body is not satisfied.
- If $a \in B^+(r)$ then r is satisfied because the model does not contain a.
- If $a \in B^-(r)$ but not $a \in B^+(r)$ or $H(r) = a$, then the rule $H(r) \leftarrow B^+(r), not\ (B^-(r) \setminus \{a\})$ is in $WForgetLP(P, a)$, which is satisfied by (X, Y), hence r is satisfied.

Next we consider the case where there is some rule of the form $a \leftarrow B, not\ C$ in P such that $B \subseteq X \setminus \{a\}$ and $C \cap (Y \setminus \{a\})$ is empty. For this case we show that $(X \cup \{a\}, Y \setminus \{a\})$ satisfies P, by showing that each rule $r \in P$ is satisfied.

- If r does not contain a, then r is satisfied because $(X, Y) \models r$.
- If $a = H(r)$, then r is satisfied since a is included in the model.
- If $a \in B^-(r)$ but not $a \in B^+(r)$ or $a = H(r)$, then the rule $H(r) \leftarrow B^+(r), not\ (B^-(r) \setminus \{a\})$ is in $WForgetLP(P, a)$ and hence r is satisfied.
- Suppose $a \in B^+(r)$ but not $a = H(r)$. We show that r is satisfied using contradiction. Suppose r is not satisfied, i.e. $H(r) \notin X \setminus \{a\}$, $B^+(r) \subseteq X \setminus \{a\}$, and $B^-(r) \cap (X \setminus \{a\})$ is empty. Let s be a rule $a \leftarrow B, not\ C$ in P such that $B \subseteq X \setminus \{a\}$ and $C \cap (Y \setminus \{a\})$ is empty; this rule exists by our initial assumption. The rule $\mathrm{C2}(r, s)$ is in P since P is closed under Cn. Now $\mathrm{C2}(r, s)$ can only contain a in the negative part. If it does not contain a, then $\mathrm{C2}(r, s) \in WForgetLP(P, a)$, so $(X, Y) \models \mathrm{C2}(r, s)$ which contradicts $H(r) \notin X \setminus \{a\}$. If it does contain a, then the rule $\mathrm{C2}(r, s)$ with *not* a removed is in $WForgetLP(P, a)$, and (X, Y) satisfies this rule, which contradicts $H(r) \notin X \setminus \{a\}$.

For the if case, suppose either suppose either $(X \setminus \{a\}, Y \setminus \{a\}) \models P$ or $(X \cup \{a\}, Y \setminus \{a\}) \models P$. Then it is easy to see that each $r \in WForgetLP(P, a)$ is satisfied by (X, Y). □

7 Conclusion

We have presented a new equivalence relation on logic programs, which we call T-equivalence. In Section 5 we gave a syntactic definition of T-equivalence, and showed that it can be characterised in terms of T-models. In Section 6 we showed that T-equivalence satisfies a number of properties in relation to the strong and weak forgetting operators of Zhang and Foo, and using this result we arrived at a definition of strong and weak forgetting in terms of T-models.

In this paper we have only considered normal logic programs. It should be possible to extend these results in this paper to more general classes of logic programs, such as nested logic programs. In follow-up work, we have extended the results to disjunctive logic programs.

Further research can be done in exploring new notions of forgetting "between" strong and weak forgetting. There is an opinion that strong equivalence should be in some sense the canonical semantics for logic programs, and hence forgetting operators should preserve strong equivalence. It would be interesting to see whether looking at forgetting in terms of T-models can provide fresh insights in constructing new forgetting operators, in particular in constructing operators that respect strong equivalence.

The properties of T-equivalence and the forgetting operators when restricted to certain syntactic subclasses of logic programs can also be considered. In particular, it would be interesting to look for subclasses where T-equivalence coincides with strong equivalence, as this would result in subclasses of logic programs where strong equivalence is preserved under strong and weak forgetting.

Bibliography

[1] Chandrabose Aravindan and Phan Minh Dung. On the correctness of unfold/fold transformation of normal and extended logic programs. *Journal of Logic Programming*, 24(3):201–217, 1995.

[2] Pedro Cabalar. A three-valued characterization for strong equivalence of logic programs. In *Proceedings of AAAI-2002*, pages 106–111, 2002.

[3] Thomas Eiter and Michael Fink. Uniform equivalence of logic programs under the stable model semantics. In *Proceedings of ICLP-2003*, pages 224–238, 2003.

[4] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, 1988.

[5] Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *Computational Logic*, 2(4):526–541, 2001.

[6] Fangzhen Lin. Reducing strong equivalence of logic programs to entailment in classical propositional logic. In *Proceedings of KR-02*, pages 170–176, 2002.

[7] Hudson Turner. Strong equivalence made easy: nested expressions and weight constraints. *Theory and Practice of Logic Programming*, 3(4-5):609–622, 2003.

[8] Yan Zhang and Norman Y. Foo. Solving logic program conflict through strong and weak forgettings. *Articifial Intelligence*, 170(8-9):739–778, 2006.