Equivalence of Logic Programs under Updates

Katsumi Inoue[1] and Chiaki Sakama[2]

[1] National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
ki@nii.ac.jp
[2] Department of Computer and Communication Sciences
Wakayama University, Sakaedani, Wakayama 640-8510, Japan
sakama@sys.wakayama-u.ac.jp

Abstract. This paper defines a general framework for testing equivalence of logic programs with respect to two parameters. Given two sets of rules $\mathcal{Q}$ and $\mathcal{R}$, two logic programs P_1 and P_2 are said to be *update equivalent with respect to* $(\mathcal{Q}, \mathcal{R})$ if $(P_1 \setminus Q) \cup R$ and $(P_2 \setminus Q) \cup R$ have the same answer sets for any two logic programs $Q \subseteq \mathcal{Q}$ and $R \subseteq \mathcal{R}$. The notion of update equivalence is suitable to take program updates into account when two logic programs are compared. That is, the notion of relativity stipulates the languages of updates, and two parameters $\mathcal{Q}$ and $\mathcal{R}$ correspond to the languages for deletion and addition, respectively. Clearly, the notion of strong equivalence is a special case of update equivalence where $\mathcal{Q}$ is empty and $\mathcal{R}$ is the set of all rules in the language. In fact, the notion of update equivalence is strong enough to capture many other notions such as weak equivalence, update equivalence on common rules, and uniform equivalence. We also discuss computation and complexity of update equivalence.

1 Introduction

The notion of equivalence in logic programming has recently become important. Because a logic program is used to represent knowledge of a problem domain, we often have to consider whether two logic programs P_1 and P_2 represent the same knowledge. For example, one logic program P_1 may be viewed as a specification of knowledge in some domain, and another representation P_2 may be expected to be a compact form of P_1 which can easily be computed.

Strong equivalence [11] is one of the most widely recognized criteria for equivalence of logic programs. Two logic programs P_1 and P_2 are said to be *strongly equivalent* if for any logic program R, $P_1 \cup R$ and $P_2 \cup R$ have the same answer sets. On the other hand, two programs are *weakly equivalent* if they agree with their answer sets. The notion of strong equivalence was introduced earlier by Maher [15] for definite programs under the name of *equivalence as program segments*. Recently, strong equivalence has been studied both logically and computationally for answer set programming [11, 18, 17, 14, 21, 2, 4].

In [11], it is argued that strong equivalence can be used to simplify a part of a logic program without looking at the other part. For example, $\{ p \leftarrow p \}$

and $\emptyset$ are strongly equivalent, so that the rule in the former set can always be eliminated from any program. On the other hand, the two weakly equivalent programs $\{ p \leftarrow not\, q \}$ and $\{ p \leftarrow \}$ are not strongly equivalent, so the rule in the former cannot be replaced by the rule in the latter. Hence, strong equivalence takes the influence of addition of a rule set to each program into account.

However, strong equivalence cannot capture the *negative* influence, i.e., deletion of a rule set from each program. For example, $P_1 = \{ p \leftarrow ,\ q \leftarrow not\, p \}$ and $P_2 = \{ p \leftarrow \}$ are strongly equivalent. According to the above discussion, the two rules in P_1 can be replaced by the rule of P_2, which implies that the rule $q \leftarrow not\, p$ can be eliminated from P_1. However, the process of program development is not always monotonic. We often revise and update our previous rules and delete some part of a program in exchange for additional new rules. In such a case, eliminating a rule like $q \leftarrow not\, p$ is harmful, and it should be kept for later uses because q should be derived whenever p is removed. Strong equivalence does not take such a possibility of removal into account.

In this paper, we consider a much stronger notion of equivalence which is tolerant of both addition and removal. Given two sets of rules $\mathcal{Q}$ and $\mathcal{R}$, two logic programs P_1 and P_2 are said to be *update equivalent with respect to* $(\mathcal{Q}, \mathcal{R})$ if $(P_1 \setminus Q) \cup R$ and $(P_2 \setminus Q) \cup R$ have the same answer sets for any two logic programs $Q \subseteq \mathcal{Q}$ and $R \subseteq \mathcal{R}$. Clearly, the notion of strong equivalence is a special case of update equivalence where $\mathcal{Q}$ is empty and $\mathcal{R}$ is the set of all rules in the language. In the above example, P_1 and P_2 are not update equivalent with respect to $(\mathcal{P}, \mathcal{P})$ where $\mathcal{P}$ is the set of all rules in the language of P_1 and P_2. This is because the removal of $p \leftarrow$ from both programs involves derivation of q in P_1. For another example, $P_3 = \{ p \leftarrow ,\ q \leftarrow p \}$ and $P_4 = \{ p \leftarrow ,\ q \leftarrow \}$ are strongly equivalent, but are not update equivalent with respect to $(\mathcal{P}, \mathcal{P})$ because the removal of $p \leftarrow$ differentiates their answer sets. Non-equivalence of P_3 and P_4 is explained as follows. While the reason why q is true in P_3 is justified by the rule $q \leftarrow p$ and the truth of p, both p and q hold as facts in P_4. So q depends on p in P_3, and the loss of p affects the loss of q, but no such dependency exists in P_4. This contrasts well with the case that the weakly equivalent programs $\{ p \leftarrow not\, q \}$ and $\{ p \leftarrow \}$ are not strongly equivalent, where the truth of p is factual in the latter but it is justified by the default rule in the former. In other words, strong equivalence distinguishes derivation of literals through negation as failure from derivation without involving negation as failure. This asymmetric property of strong equivalence is not always natural from the viewpoint of nonmonotonic reasoning.

Without any restriction on $\mathcal{Q}$ and $\mathcal{R}$, two logic programs are called *strongly update equivalent* if they are update equivalent with respect to $(\mathcal{P}, \mathcal{P})$. Surprisingly, only a small class of strongly equivalent logic programs becomes strongly update equivalent. In fact, we prove that two logic programs are strongly update equivalent only if they only differ in additional valid (or tautological) rules. However, a slightly modified definition assures that most modular transformations of rules proposed in the literature preserve *update equivalence on common rules*. As a related work, Eiter *et al.* [3] discuss another generalization of strong equiv-

alence in the context of updating logic programs. However, the effect of removal in equivalence of logic programs has never been analyzed so far. Leite [10] introduces another update equivalence in the context of dynamic logic programming, but it is not a generalization of strong equivalence.

For update equivalence, often we can restrict the languages of changing parts $(\mathcal{Q}, \mathcal{R})$ to some subsets of the whole language of programs. Such restriction is practicably interesting because logic programs and *deductive databases* are usually divided into invariable and variable parts such that only variable parts are changed in updates [19]. This notion of equivalence is partially considered in [14] as *relative equivalence* and in [2] as *uniform equivalence*.

The rest of this paper is organized as follows. Section 2 reviews the answer set semantics and previous definitions of equivalence. Section 3 defines relative update equivalence. Section 4 shows the necessary and sufficient condition of strong update equivalence, and also considers update equivalence on common rules. Section 5 discusses the application of relative update equivalence to database updates. Section 6 shows a translation from relative update equivalence into relative strong equivalence, and considers the time complexity of update equivalence. Section 7 concludes the paper.

2 Background

A *(logic) program* is represented in a *general extended disjunctive program* (GEDP) [13, 7], which consists of a finite number of *rules* of the form:

$$L_1; \cdots; L_k; \mathit{not}\, L_{k+1}; \cdots; \mathit{not}\, L_l \leftarrow L_{l+1}, \ldots, L_m, \mathit{not}\, L_{m+1}, \ldots, \mathit{not}\, L_n \qquad (1)$$

where each L_i is a literal ($n \geq m \geq l \geq k \geq 0$), and *not* is *negation as failure* (NAF). The symbol ; represents a disjunction. For any literal L, the literal complimentary to L is written as $\overline{L}$, that is, when A is an atom, $\overline{A} = \neg A$ and $\overline{\neg A} = A$. A rule with variables stands for the set of its ground instances. The left-hand side of the rule is the *head*, and the right-hand side is the *body*. For each rule r of the form (1), $\mathit{head}^+(r)$, $\mathit{head}^-(r)$, $\mathit{body}^+(r)$ and $\mathit{body}^-(r)$ denote the sets of literals $\{L_1, \ldots, L_k\}$, $\{L_{k+1}, \ldots, L_l\}$, $\{L_{l+1}, \ldots, L_m\}$, and $\{L_{m+1}, \ldots, L_n\}$, respectively. A rule r is an *integrity constraint* if $\mathit{head}^+(r) = \emptyset$. Any rule with the empty body $H \leftarrow$ is called a *fact* and is also written as H without the symbol $\leftarrow$ as long as no confusion arises. A GEDP is an *extended disjunctive program* (EDP) [5] if it contains no NAF in the head of any rule (i.e., $k = l$).

The semantics of a program is given by its *answer sets*. First, let P be a GEDP without NAF (i.e., $k = l$ and $m = n$) and $S \subseteq \mathit{Lit}$, where Lit is the set of all ground literals in the language of P. Then, S is an *answer set* of P if S is a minimal set satisfying the conditions:

1. For each ground rule r of the form $L_1; \cdots; L_l \leftarrow L_{l+1}, \ldots, L_m$ from P, $\mathit{body}^+(r) \subseteq S$ implies $\mathit{head}^+(r) \cap S \neq \emptyset$;
2. If S contains a pair of complementary literals L and $\overline{L}$, then $S = \mathit{Lit}$.

Second, given *any* GEDP P (with NAF) and $S \subseteq Lit$, consider the GEDP (without NAF) P^S obtained as follows: a rule $L_1; \cdots; L_k \leftarrow L_{l+1}, \ldots, L_m$ is in P^S if there is a ground rule r of the form (1) from P such that $head^-(r) \subseteq S$ and $body^-(r) \cap S = \emptyset$. Then, S is an *answer set* of P if S is an answer set of P^S.

An answer set is *consistent* if it is not Lit. A program is *consistent* if it has a consistent answer set. An answer set S of P is *minimal* if there is no other answer set S' of P such that $S' \subset S$. Every answer set of any EDP is minimal [5], but the minimality of answer sets no longer holds for GEDPs [13]. The set of all answer sets of P is written as $\mathcal{AS}(P)$.

The notions of weak and strong equivalence are defined as follows.

Definition 2.1 Let P_1, P_2, and R be programs.

(1) P_1 and P_2 are *(weakly) equivalent* if $\mathcal{AS}(P_1) = \mathcal{AS}(P_2)$.
(2) P_1 and P_2 are *equivalent relative to* R [8] if $\mathcal{AS}(P_1 \cup R) = \mathcal{AS}(P_2 \cup R)$.
(3) P_1 and P_2 are *strongly equivalent* [11] if P_1 and P_2 are equivalent relative to any program.

Obviously, two strongly equivalent programs are weakly equivalent, and in fact they are equivalent relative to $\emptyset$.

3 Relative Update Equivalence

Relative update equivalence of logic programs is an elaboration of strong equivalence of logic programs under the two additional concepts: (a) deletion of rules as well as addition, and (b) the restriction of languages for deletion and addition.

Definition 3.1 Suppose that P_1, P_2, $\mathcal{Q}$, and $\mathcal{R}$ are sets of rules with a common underlying language. P_1 and P_2 are *update equivalent with respect to* $(\mathcal{Q}, \mathcal{R})$ if $\mathcal{AS}((P_1 \setminus Q) \cup R) = \mathcal{AS}((P_2 \setminus Q) \cup R)$[3] for any programs $Q \subseteq \mathcal{Q}$ and $R \subseteq \mathcal{R}$.

Each rule in $\mathcal{Q}$ is called a *removable rule*, while each rule in $\mathcal{R}$ is called an *insertable rule*. A removable or insertable rule is called an *updatable rule*.

When two programs are update equivalent with respect to some pair $(\mathcal{Q}, \mathcal{R})$, they are called *relatively update equivalent*, or *update equivalent* for short. As it is seen by its name, (relative) update equivalence enables us to give the semantics for equivalence of logic programs with respect to updates. That is, P_1 and P_2 are regarded as equivalent programs in the sense that they are equivalent after any program $Q \subseteq \mathcal{Q}$ is deleted and then any program $R \subseteq \mathcal{R}$ is inserted. Consideration of such update equivalence is meaningful and important because

[3] For removing rules containing variables from a program, the set difference operation is semantically defined on ground programs as $P \setminus Q = ground(P) \setminus ground(Q)$, where $ground(P)$ is the ground instances of elements from P. For example, when x is a variable and a is a constant, $\{p(a)\} \setminus \{p(x)\} = \emptyset$, and $\{p(x)\} \setminus \{p(a)\} = \{p(y) \mid y \neq a\}$. Similarly, the union $P \cup Q$ and the intersection $P \cap Q$ are respectively defined as $ground(P) \cup ground(Q)$ and $ground(P) \cap ground(Q)$, e.g., $\{p(a)\} \cup \{p(x)\} = \{p(x)\}$.

it guarantees equivalence of two different programs dynamically in the face of any common change of these programs.[4]

In the special case, both $\mathcal{Q}$ and $\mathcal{R}$ are given as the set of all rules in the language of programs, that is, any rule can be either deleted or added in updates. This case is further investigated in Section 4. Often, however, we want to consider update equivalence with respect to $(\mathcal{Q}, \mathcal{R})$ in which $\mathcal{Q}$ and $\mathcal{R}$ are given as some distinguished sets of updatable rules. Such restriction to updates is practicably interesting because logic programs and *deductive databases* are usually divided into invariable and variable parts such that only variable parts are updatable [19]. *Abductive logic programming* [9, 6] is another example where $\mathcal{Q}$ and $\mathcal{R}$ are defined as distinguished sets of *abducibles*. Such applications of relative update equivalence are investigated in Section 5.

A similar notion using some distinguished set of insertable rules can also be defined for strong equivalence.[5]

Definition 3.2 Suppose that P_1 and P_2 are programs, and that $\mathcal{R}$ is a set of insertable rules. P_1 and P_2 are *strongly equivalent with respect to* $\mathcal{R}$ if $\mathcal{AS}(P_1 \cup R) = \mathcal{AS}(P_2 \cup R)$ holds for any program $R \subseteq \mathcal{R}$.

4 Strong Update Equivalence

In the general notion of relative update equivalence, we can restrict updatable rules to some distinguished rules in the language. In this section, we consider the special case of update equivalence where such restriction is not specified.

Definition 4.1 Two programs P_1 and P_2 are *strongly update equivalent* (*S-update equivalent*, for short) if $\mathcal{AS}((P_1 \setminus Q) \cup R) = \mathcal{AS}((P_2 \setminus Q) \cup R)$ holds for any programs Q and R.

For example, two programs $\{ p \leftarrow p \}$ and $\emptyset$ are S-update equivalent. Obviously, two S-update equivalent programs are strongly equivalent, and in fact they are equivalent for $Q = \emptyset$ and for any program R. By definition, two S-update equivalent programs are update equivalent with respect to any pair $(\mathcal{Q}, \mathcal{R})$ of updatable rules.

4.1 Characterization of S-Update Equivalence

One important problem is how two strongly update equivalent programs are different from each other. With regard to this problem, we can show that two

[4] Eiter *et al.* [3] have also discussed a notion of update equivalence. The focuses in [3] are different from ours in that the semantics of updates are captured by Kripke structures and that finitary characterization of updates are described.

[5] For strong equivalence, Lin [14] has also mentioned the idea of relative equivalence by defining equivalence between two logic programs with respect to a set of atoms. Our definition is more general than Lin's as a set of rules is taken into account.

S-update equivalent programs are almost identical; the difference between the two always consists of valid rules. Here, a valid rule is a rule that never changes the answer sets of any program if the rule is added to the program.

Definition 4.2 A rule r is *valid* if $\{r\}$ is strongly equivalent to $\emptyset$.

Lemma 4.1 *Let U be a program. If U and $\emptyset$ are strongly equivalent, then U is a set of valid rules.*

Lemma 4.2 *Let P be a program, and V a set of valid rules. Then, P and $P \cup V$ are S-update equivalent.*

The *symmetric difference* $P_1 \Delta P_2$ of two programs P_1 and P_2 is defined as $P_1 \Delta P_2 = (P_1 \setminus P_2) \cup (P_2 \setminus P_1)$. The next theorem shows that update equivalence of P_1 and P_2 is determined by the validity of $P_1 \Delta P_2$.

Theorem 4.3 *Two programs P_1 and P_2 are S-update equivalent if and only if $P_1 \Delta P_2$ is a set of valid rules.*

Proof. Suppose that P_1 and P_2 are S-update equivalent. Then, for any program R, $\mathcal{AS}((P_1 \setminus P_2) \cup R) = \mathcal{AS}((P_2 \setminus P_2) \cup R)$, that is, $\mathcal{AS}((P_1 \setminus P_2) \cup R) = \mathcal{AS}(R)$. Hence, $P_1 \setminus P_2$ is strongly equivalent to $\emptyset$. By Lemma 4.1, $P_1 \setminus P_2$ is a set of valid rules. The same argument can be applied to $P_2 \setminus P_1$.

Conversely, suppose that $P_1 \setminus P_2$ and $P_2 \setminus P_1$ are sets of valid rules. By Lemma 4.2, P_2 and $P_2 \cup (P_1 \setminus P_2)$ are S-update equivalent, that is, P_2 and $P_2 \cup P_1$ are S-update equivalent. Similarly, P_1 and $P_1 \cup P_2$ are S-update equivalent. Therefore, P_2 and P_1 are S-update equivalent. □

The next theorem completely characterizes valid rules by their syntax.

Theorem 4.4 *A rule r of the form* (1) *is valid if and only if it satisfies one of the following:*

(i) $head^+(r) \cap body^+(r) \neq \emptyset$.
(ii) $head^-(r) \cap body^-(r) \neq \emptyset$.
(iii) $body^+(r) \cap body^-(r) \neq \emptyset$.
(iv) $head^+(r) \cup body^-(r) \neq \emptyset$ *and there are two literals L_1 and L_2 in $head^-(r) \cup body^+(r)$ such that $\overline{L_1} = L_2$.*

Proof. Let R be any program, and S any answer set of R. If a rule r satisfies one of (i), (ii), and (iii), then it is easy to show that $\mathcal{AS}(R^S) = \mathcal{AS}((R \cup \{r\})^S)$. By $S \in \mathcal{AS}(R^S)$, S is also an answer set of $R \cup \{r\}$. Next, suppose that r satisfies (iv). Let L_1 and L_2 be a complementary pair of literals in the condition (iv).

(I) Firstly consider the case that S is consistent. (I-a) If $L_1, L_2 \in body^+(r)$, then $body^+(r) \not\subseteq S$ because S is consistent. So no literal is derived through the rule in $\{r\}^S$. (I-b) If $L_1, L_2 \in head^-(r)$, then $head^-(r) \not\subseteq S$ because S is consistent. Then $\{r\}^S = \emptyset$. (I-c) If $L_1 \in body^+(r)$ and $L_2 \in head^-(r)$, then either (c-1) $L_1 \notin S$ and $L_2 \notin S$, or (c-2) $L_1 \in S$ and $L_2 \notin S$, or (c-3) $L_1 \notin S$ and $L_2 \in S$. If (c-1) or (c-2), $head^-(r) \not\subseteq S$ and thus $\{r\}^S = \emptyset$. If (c-3),

$body^+(r) \not\subseteq S$ and hence no literal is derived through the rule in $\{r\}^S$. In either case, $\mathcal{AS}(R^S) = \mathcal{AS}((R \cup \{r\})^S)$ holds.

(II) Secondly suppose that $S = Lit$. In this case, $head^-(r) \cup body^+(r) \subseteq S$, that is, $head^-(r) \subseteq S$ and $body^+(r) \subseteq S$. On the other hand, $head^+(r) \cup body^-(r) \neq \emptyset$ implies that either (II-a) $head^+(r) \neq \emptyset$ or (II-b) $body^-(r) \neq \emptyset$ holds. If (II-a), r is satisfied by S, that is, $body^+(r) \subseteq S$ implies $head^+(r) \cap S \neq \emptyset$. If (II-b), $\{r\}^S = \emptyset$ holds. In either case, $\mathcal{AS}(R^S) = \mathcal{AS}((R \cup \{r\})^S)$ holds. Thus, $S = Lit$ is also an answer set of $R \cup \{r\}$. Hence, r is valid.

Conversely, suppose that r satisfies none of (i), (ii), (iii), and (iv). Then,

$$head^+(r) \cap body^+(r) = \emptyset, \tag{2}$$

$$head^-(r) \cap body^-(r) = \emptyset, \tag{3}$$

$$body^+(r) \cap body^-(r) = \emptyset, \tag{4}$$

and either

$$head^+(r) \cup body^-(r) = \emptyset \tag{5}$$

or there is no complementary pair of literals L_1 and L_2 in $head^-(r) \cup body^+(r)$. Let $S = head^-(r) \cup body^+(r)$.

(I) Suppose that S is consistent. In this case, there is no complementary literals L_1 and L_2 in $S = head^-(r) \cup body^+(r)$. Then, $body^-(r) \cap S = \emptyset$ by (3) and (4). Also by $head^-(r) \subseteq S$, $\{r\}^S \neq \emptyset$ holds. By $body^+(r) \subseteq S$, there is a literal $L \in head^+(r)$ such that $L \notin S$ by (2). Now, let R be the program exactly consisting of the facts of S, i.e., $R = S$. Obviously, $R^S = S$. Then, $L \in S'$ for some answer set $S' \in \mathcal{AS}(\{r\}^S \cup R)$, while $L \notin S$ for the answer set $S \in \mathcal{AS}(\emptyset \cup R)$. Hence, $\{r\}$ and $\emptyset$ are not strongly equivalent.

(II) Suppose that S is inconsistent. In this case, there is a pair of complementary literals L_1 and L_2 in $head^-(r) \cup body^+(r)$. Then, $head^+(r) \cup body^-(r) = \emptyset$ holds by (5), that is, $head^+(r) = body^-(r) = \emptyset$. Thus the rule r is an integrity constraint with no NAF in the body. Again, let $R = S$. Then, the unique answer set of R is Lit. However, Lit is not an answer set of $R \cup \{r\}^{Lit}$ because Lit cannot satisfy the rule of $\{r\}^{Lit}$ so that there is no answer set of $R \cup \{r\}$. Hence, $\{r\}$ and $\emptyset$ are not strongly equivalent. By (I) and (II), r is not valid. □

The conditions (i) and (iii) in Theorem 4.4 are considered in the context of EDPs without classical negation in [1], in which rules satisfying (i) and (iii) are called *tautologies* and *contradictions*, respectively. The condition (ii) is similar to (i) and is meaningful only for the class of GEDPs with NAF in heads. The condition (iv) is necessary for extended programs with classical negation. In other words, (i), (ii) or (iii) is the necessary and sufficient condition for a rule in a program without classical negation to be valid. According to Theorem 4.4, the following rules are all valid:

$$p \leftarrow p,\ q, \qquad q \leftarrow p,\ not\ p, \qquad q;\ not\ p \leftarrow \neg p, \qquad \leftarrow p,\ \neg p,\ not\ q.$$

However, the following conditions are excluded from the definition of valid rules.

(v) $head^+(r) \cap head^-(r) \neq \emptyset$.

(vi) there are two literals L_1 and L_2 in $head^+(r)$ such that $\overline{L_1} = L_2$.
(vii) $head^+(r) = body^-(r) = \emptyset$ and there are two literals L_1 and L_2 in $head^-(r) \cup body^+(r)$ such that $\overline{L_1} = L_2$.

An example for the case (v) is $p; not\, p \leftarrow$, which has two answer sets $\emptyset$ and $\{p\}$. Similarly, for the case (vi), $p; \neg p \leftarrow$ has two answer sets $\{p\}$ and $\{\neg p\}$. For the case (vii), the integrity constraint $\leftarrow p, \neg p$ eliminates the answer set Lit if it is added to $\{\, p \leftarrow , \neg p \leftarrow \}$ as in the proof of Theorem 4.4.

4.2 Update Equivalence on Common Rules

Theorem 4.3 implies that only valid rules can be safely eliminated from a program under the situation that any update can occur. This is a rather unexpected result. In fact, we cannot even show S-update equivalence of $\{\, p; p \leftarrow q \,\}$ and $\{\, p \leftarrow q \,\}$, although the latter rule is just a *merged* form of the former. The main reason why such two programs P_1 and P_2 are not S-update equivalent is that, when a rule r_1 in $P_1 \setminus P_2$ is removed, r_1 is syntactically different from the semantically equivalent rule r_2 in $P_2 \setminus P_1$. Thus, removing $Q = \{r_1\}$ from P_1 and P_2 results in elimination of r_1 from P_1 while r_2 remains in P_2.

A rational solution is to exclude removal of rules from $P_1 \Delta P_2$ in testing update equivalence. In other words, a removal-addition pair (Q, R) is considered for any program R and any set Q of rules from $P_1 \cap P_2$.

Definition 4.3 Two programs P_1 and P_2 are *update equivalent on common rules* (*C-update equivalent*, for short) if $\mathcal{AS}((P_1 \setminus Q) \cup R) = \mathcal{AS}((P_2 \setminus Q) \cup R)$ holds for any pair (Q, R) of programs such that Q consists of rules from $P_1 \cap P_2$ and R is any set of rules.

Update equivalence on common rules is a restricted version of S-update equivalence, and hence update equivalence implies C-update equivalence. Many important program transformations proposed in the literature preserve C-update equivalence. For example, the previous program $\{\, p; p \leftarrow q \,\}$ is C-update equivalent to its merged form $\{\, p \leftarrow q \,\}$. In [7], it is shown that any rule of the form
$not\, L_1; \cdots; not\, L_l \leftarrow L_{l+1}, \ldots, L_m, not\, L_{m+1}, \ldots, not\, L_n$
can be transformed to an integrity constraint of the form
$\leftarrow L_1, \ldots, L_l, L_{l+1}, \ldots, L_m, not\, L_{m+1}, \ldots, not\, L_n$
without changing the answer sets. Such a *modular* transformation preserves C-update equivalence as well as strong equivalence [11]. In fact, whenever $P_1 \cap P_2$ is empty, P_1 and P_2 are C-update equivalent if and only if P_1 and P_2 are strongly equivalent. The next theorem generalizes this fact.

Theorem 4.5 *Two programs P_1 and P_2 are C-update equivalent if and only if $P_1 \setminus P_2$ and $P_2 \setminus P_1$ are strongly equivalent.*

Proof. Let $P = P_1 \cap P_2$. If P_1 and P_2 are C-update equivalent, then $\mathcal{AS}((P_1 \setminus P) \cup R) = \mathcal{AS}((P_2 \setminus P) \cup R)$ holds for any program R. Then, $P_1 \setminus P$ and $P_2 \setminus P$ are strongly equivalent. That is, $P_1 \setminus P_2$ and $P_2 \setminus P_1$ are strongly equivalent.

Conversely, suppose that $P_1 \backslash P_2$ and $P_2 \backslash P_1$ are strongly equivalent. Then, $P_1 \backslash P$ and $P_2 \backslash P$ are strongly equivalent. That is, $\mathcal{AS}((P_1 \backslash P) \cup R) = \mathcal{AS}((P_2 \backslash P) \cup R)$ holds for any program R. Here, $R = (R \cap P) \cup (R \backslash P)$. Let Q be the program such that $R \cap P = P \backslash Q$, and R' be the program $R \backslash P$. Then, $(P_i \backslash P) \cup R = (P_i \backslash Q) \cup R'$ holds for $i = 1, 2$. Hence, $\mathcal{AS}((P_1 \setminus Q) \cup R') = \mathcal{AS}((P_2 \setminus Q) \cup R')$ holds. Since R is any program, Q can be any subset of P and R' can also be any program. This implies that P_1 and P_2 are C-update equivalent. □

On the other hand, an *unfold/fold transformation* [20] does not preserve C-update equivalence. For example, $\{ p \leftarrow q, \quad q \leftarrow r \}$ and $\{ p \leftarrow r, \quad q \leftarrow r \}$ are weakly equivalent, but are not even strongly equivalent because the addition of q causes the truth of p in the former only. In Section 1, we have seen that $\{ p \leftarrow , \ q \leftarrow not\, p \}$ is strongly equivalent to $\{ p \leftarrow \}$. However, these two programs are not C-update equivalent because removing the common $p \leftarrow$ derives q in the former. Although it is claimed that strong equivalence allows us to replace the former rules with the latter, we regard that such a transformation is not tolerant of program updates.[6] Hence, C-update equivalence gives us a better criterion of program transformation than strong equivalence under the situation that updates may occur.

The relationship between several notions of equivalence in logic programs can be summarized in the form of relative update equivalence as follows.

Proposition 4.6 *Let P_1 and P_2 be programs such that $P_1 \subset \mathcal{P}$ and $P_2 \subset \mathcal{P}$ where $\mathcal{P}$ is the set of all rules in the language of P_1 and P_2.*

(1) *P_1 and P_2 are S-update equivalent iff they are update equivalent wrt $(\mathcal{P}, \mathcal{P})$. iff they are update equivalent wrt $(P_1 \cup P_2, \mathcal{P})$.*
(2) *P_1 and P_2 are C-update equivalent iff they are update equivalent wrt $(P_1 \cap P_2, \mathcal{P})$.*
(3) *P_1 and P_2 are strongly equivalent iff they are update equivalent wrt $(\emptyset, \mathcal{P})$.*
(4) *P_1 and P_2 are weakly equivalent iff they are update equivalent wrt $(\emptyset, \emptyset)$.*

Note in Proposition 4.6 (1) that the removal rules in S-update equivalence can be set to the union of two given programs, $P_1 \cup P_2$, rather than the set of all rules $\mathcal{P}$ in the language. This is because any rule in $\mathcal{P} \setminus (P_1 \cup P_2)$ has no effect if it is removed from either P_1 or P_2, that is, both P_1 and P_2 are unchanged by such a removal.

5 Uniform Equivalence

In database updates, updates are permitted only on variable data. Representing a database as a logic program P, P is usually divided into two parts: $P = Int(P) \cup Ext(P)$, where $Int(P) \cap Ext(P) = \emptyset$. Here, $Ext(P)$ denotes the set of facts in P called an *extensional database*, and the set of non-facts $Int(P) = P \setminus Ext(P)$

[6] Unlike strong equivalence, RED$^-$, NONMIN, WGPPE and S-IMP in [4] fail to preserve C-update equivalence (thereby, S-update equivalence).

is called an *intensional database*. In databases, $Ext(P)$ can be considered as variable data while $Int(P)$ is regarded as invariable knowledge. Similarly, the set of all literals in the language is divided into the *extensional literals* $\mathcal{E}$ and the *intensional literals* $\mathcal{I}$ as: $Lit = \mathcal{I} \cup \mathcal{E}$, where $\mathcal{I} \cap \mathcal{E} = \emptyset$. Here, $\mathcal{I}$ is the set of all literals with the predicates appearing in heads of $Int(P)$, and $\mathcal{E}$ is the set of all other literals. Then, two databases P_1 and P_2 are *equivalent* in the sense of Sagiv [19] if P_1 and P_2 are strongly equivalent with respect to $\mathcal{E}$.

Sagiv [19] also considers *uniform equivalence* of two Datalog programs which can be defined as follows. Two programs P_1 and P_2 are *uniformly equivalent* if the output of P_1 agrees with that of P_2 for any input from $Lit = \mathcal{I} \cup \mathcal{E}$, where the output of P is defined as $\{\, S \cap \mathcal{I} \mid S \in \mathcal{AS}(P \cup R),\ R \subseteq Lit \,\}$.

Uniform equivalence implies Sagiv's equivalence. In fact, uniform equivalence takes an input literal set R not only from the extensional part $\mathcal{E}$ but also from the intensional one $\mathcal{I}$. Since it is obvious that the extensional part in each answer set S, i.e., $\mathcal{E} \cap S$, is always the same between the two, it turns out that two programs are uniformly equivalent if and only if they are strongly equivalent with respect to Lit. Sagiv uses the notion of uniform equivalence for minimizing Datalog programs. Eiter and Fink [2] consider uniform equivalence for normal and extended disjunctive programs. The notion of (uniform) equivalence can also be generalized to update equivalence as follows.

Definition 5.1 Let P_1 and P_2 be programs. Suppose that $\mathcal{I}$ and $\mathcal{E}$ are the sets of intentional and extensional literals, respectively, which are common to both P_1 and P_2. Then, P_1 and P_2 are *extensionally update equivalent* if they are update equivalent with respect to $(\mathcal{E}, \mathcal{E})$. On the other hand, P_1 and P_2 are *uniformly update equivalent* if they are update equivalent with respect to (Lit, Lit).

Example 5.1 Suppose that two databases P_1 and P_2 are given as

$$P_1 = \{\, p \leftarrow a,\ q, \qquad q \leftarrow not\, b, \quad b \leftarrow \,\},$$
$$P_2 = \{\, p \leftarrow a,\ not\, b, \quad q \leftarrow not\, b, \quad b \leftarrow \,\},$$

where $\mathcal{E} = \{a, b\}$ and $\mathcal{I} = \{p, q\}$. Then, P_1 and P_2 are extensionally update equivalent, but are not uniformly update equivalent. In fact, P_1 and P_2 are update equivalent with respect to $(\mathcal{E}, \mathcal{E})$, but are not with respect to (Lit, Lit) because $\mathcal{AS}(P_1 \cup \{a, q\}) = \{\{a, b, p, q\}\}$ while $\mathcal{AS}(P_2 \cup \{a, q\}) = \{\{a, b, q\}\}$.

A database P is *disjunctive* if disjunctions appear in P. Usually, $Ext(P)$ also contains disjunctive facts in disjunctive databases. Let $\mathcal{D}(\mathcal{E})$ be the set of all disjunctions of literals from $\mathcal{E}$. Then, *disjunctively update equivalence* of two databases is defined as update equivalence with respect to $(\mathcal{D}(\mathcal{E}), \mathcal{D}(\mathcal{E}))$. This can also be represented by the notion of *disjunctive explanations* in [8].

Often, updates on the invariable part are translated into updates on the variable part in databases. This type of updates is called *view updates*. The view update problem in databases is concerned with the problem of translating an update request on intentional literals in $\mathcal{I}$ into updates on extensional literals $\mathcal{E}$. This problem can be characterized by *extended abduction* [6], and we will consider equivalence with respect to abductive updates in another paper.

6 Computation and Complexity

This section considers the computational aspects of update equivalence.

We first show a translation of relative update equivalence into relative strong equivalence, which is similar to the transformation proposed in [6, 8].

Given two tested programs P_1 and P_2 and the updatable rules $\mathcal{Q}$ and $\mathcal{R}$, we convert the update equivalence problem $(P_1, P_2, \mathcal{Q}, \mathcal{R})$ into the strong equivalence problem $(K_1, K_2, \mathcal{K}) = (\nu(P_1), \nu(P_2), \mu(\mathcal{Q}) \cup \mathcal{R})$, where K_1 and K_2 are programs and $\mathcal{K}$ is a set of insertable rules. To this end, any removable rule r in $\mathcal{Q}$ is associated with a unique literal δ_r (the *name* of r) through negation as failure as $not\, \delta_r$. In this way, the deletion of r is realized by the addition of δ_r to the program. Then, the translations ν and μ are defined as:

$$\nu(P_i) = (P_i \setminus \mathcal{Q}) \cup \{\, (H \leftarrow B,\, not\, \delta_r) \mid r = (H \leftarrow B) \in P_i \cap \mathcal{Q} \,\}, \quad (i = 1, 2)$$
$$\mu(\mathcal{Q}) = \{\, \delta_r \mid r \in \mathcal{Q} \,\}.$$

Theorem 6.1 *Suppose that $(P_1, P_2, \mathcal{Q}, \mathcal{R})$ is converted to $(K_1, K_2, \mathcal{K})$ as above. P_1 and P_2 are update equivalent with respect to $(\mathcal{Q}, \mathcal{R})$ if and only if K_1 and K_2 are strongly equivalent with respect to $\mathcal{K}$.*

Notice that the translation ν is modular. Without loss of generality, we can assume that the removable rules $\mathcal{Q}$ are finite and are included in $P_1 \cup P_2$; if $\mathcal{Q} \not\subseteq P_1 \cup P_2$, we can substitute $\mathcal{Q}$ with $\mathcal{Q} \cap (P_1 \cup P_2)$ without changing the result of equivalence testing. See also Proposition 4.6 (1). Then, the translations ν and μ can always be computed in linear time.

Example 6.1 Two programs $\{\, p \leftarrow not\, q,\ \leftarrow q \,\}$ and $\{\, p \leftarrow ,\ \leftarrow q \,\}$ are strongly equivalent. However, they are neither S-update equivalent nor C-update equivalent. In fact, removing the common constraint $\leftarrow q$ and adding $q \leftarrow$ cause the deletion of p in the former only. Let us verify this fact. The constraint $\leftarrow q$ is converted to $\leftarrow q,\, not\, \delta_{\leftarrow q}$, so the addition of $\{\, \delta_{\leftarrow q},\ q \,\}$ to both converted programs causes the same effect. On the other hand, if the constraint $\leftarrow q$ as well as other rules are not removable, no rule is converted, and hence they become update equivalent with respect to $(\emptyset, \mathcal{R})$ for any $\mathcal{R}$, that is, they are (relatively) strongly equivalent.

Theorem 6.1 reduces testing relative update equivalence to testing relative strong equivalence. At the moment, however, no sophisticated procedure is known for testing relative strong equivalence,[7] although some useful methods exist for testing non-relative strong equivalence [18, 14, 21]. In fact, we can show that update equivalence in general is harder than strong equivalence as follows.

To establish the computational complexity of relative update equivalence of propositional programs, we can use Proposition 4.6 (4) and the result by Turner [21] that deciding weak equivalence of two GEDPs is Π_2^P-hard.

[7] Woltran [22] recently showed that strong/uniform equivalence *wrt all rules over some alphabet* can be reduced to weak equivalence. We can also show that strong equivalence *wrt a finite set of rules* can be reduced to weak equivalence.

Theorem 6.2 *The problem of checking relative update equivalence of two propositional programs is Π_2^P-hard in general.*

By contrast, checking S-update equivalence of two programs P_1 and P_2 can be done in polynomial time by Theorems 4.3 and 4.4. That is, we check whether each rule in $P_1 \Delta P_2$ is valid or not. If every rule in $P_1 \Delta P_2$ is valid, P_1 and P_2 are S-update equivalent; otherwise, they are not S-update equivalent.

Theorem 6.3 *S-update equivalence of two propositional programs can be decided in polynomial time.*

Now, we compare the notions of weak, strong, and update equivalence in the non-relative versions from the complexity viewpoint. In [21], it is shown that deciding strong equivalence of two GEDPs is in coNP and that deciding weak equivalence of two GEDPs is Π_2^P-hard. Hence, the strength of S-update equivalence is reflected in the time complexity of the respective decision problems: unless the polynomial hierarchy collapses, deciding S-update equivalence is easier than deciding strong equivalence, which in turn is easier than weak equivalence.

Although the notion of S-update equivalence seems too strong to be practical, C-update equivalence is more attractive. In fact, Theorem 4.5 indicates that C-update equivalence is much closer to strong equivalence.

Theorem 6.4 *The problem of checking C-update equivalence of two propositional programs is coNP-complete.*

7 Conclusion

We have proposed update equivalence in logic programming, investigated its properties, considered several variants, and presented their applications. We have completely characterized each case of update equivalence. Although the condition for S-update equivalence is very strong, one for C-update equivalence is rather practical. We have also shown that most previously proposed notions of equivalence in logic programming and deductive databases can be characterized by relative update equivalence. The notion of update equivalence can thus be used to guarantee the correctness of a program transformation in a dynamic setting, and is helpful to optimize logic programs for various applications.

We can consider more general form of programs allowing *nested expressions* [12]. There are some formalizations of strong equivalence of two nested logic programs in non-standard logics [18, 17, 21]. While we have shown that relative update equivalence can be converted to relative strong equivalence, more direct connections between these logics and update equivalence are also worth investigating. Another future work is to characterize many transformation techniques in logic programming in terms of subclasses of relative update equivalence. New transformations preserving relative update equivalence should also be developed.

References

1. S. Brass and J. Dix. Characterization of the disjunctive stable semantics by partial evaluation. *Journal of Logic Programming*, 32(3):207–228, 1997.
2. T. Eiter and M. Fink. Uniform equivalence of logic programs under the stable model semantics. In: *Proc. of ICLP 2003*, LNCS 2916, pp. 224–238, Springer, 2003.
3. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Reasoning about evolving nonmonotonic knowledge bases. In: *Proc. of LPAR 2001*, LNAI 2250, pp. 407–421, Springer, 2001.
4. T. Eiter, M. Fink, H. Tompits, and S. Woltran. Simplifying logic programs under uniform and strong equivalence. In: *Proc. of LPNMR 2004*, LNAI 2923, pp. 87–99, Springer, 2004.
5. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
6. K. Inoue. A simple characterization of extended abduction. In *Proc. of the 1st International Conference on Computational Logic*, LNAI 1861, pp. 718–732, Springer, 2000.
7. K. Inoue and C. Sakama. Negation as failure in the head. *Journal of Logic Programming*, 35(1):39–78, 1998.
8. K. Inoue and C. Sakama. Disjunctive explanations. In: *Proc. of ICLP 2002*, LNCS 2401, pp. 317–332, Springer, 2002.
9. A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. In: D. M. Gabbay, C. J. Hogger and J. A. Robinson (eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pp. 235–324, Oxford University Press, 1998.
10. J. A. Leite. *Evolving Knowledge Bases*. IOS Press, 2003.
11. V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.
12. V. Lifschitz, L. R. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.
13. V. Lifschitz and T. Y. C. Woo. Answer sets in general nonmonotonic reasoning (preliminary report). In: *Proc. of KR '92*, pp. 603–614, Morgan Kaufmann, 1992.
14. F. Lin. Reducing strong equivalence of logic programs to entailment in classical propositional logic. In: *Proc. of KR 2002*, pp. 170–176, Morgan Kaufmann, 2002.
15. M. J. Maher. Equivalence of logic programs. In: [16], pp. 627–658, 1988.
16. J. Minker (ed.). *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988.
17. M. Osorio, J. A. Navarro, and J. Arrazola. Equivalence in answer set programming. In: *Proc. of LOPSTR 2001*, LNCS 2372, pp. 57–75, Springer, 2001.
18. D. Pearce, H. Tompits, and S. Woltran. Encodings for equilibrium logic and logic programs with nested expressions. In: *Proc. of EPIA 2001*, LNCS 2258, pp. 306–320, Springer, 2001.
19. Y. Sagiv. Optimizing Datalog programs. In: [16], pp. 659–668, 1988.
20. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In: *Proc. of the 2nd International Conference on Logic Programming*, pp. 127–138, 1984.
21. H. Turner. Strong equivalence made easy: nested expressions and weight constraints. *Theory and Practice of Logic Programming*, 3(4–5):609–622, 2003.
22. S. Woltran. Characterizations for relativized notions of equivalence in answer set programming. In: *Proc. of JELIA '04*, LNAI, this volume, 2004.