International Journal of Innovative Computing, Information and Control
Volume **11**, Number **1**, February **2015**

ICIC International ©2015 ISSN 1349-4198
pp. 327–347

AN ALGORITHM FOR CHECKING INCORRECTNESS OF A RULE IN EQUIVALENT TRANSFORMATION PROGRAMS

Hiroshi Mabuchi[1] and Shinya Miyajima[2]

[1]Faculty of Software and Information Science
Iwate Prefectural University
152-52 Sugo, Takizawa, Iwate 020-0693, Japan
mabu@iwate-pu.ac.jp

[2]Faculty of Engineering
Gifu University
1-1, Yanagido, Gifu-shi, Gifu 501-1193, Japan
miyajima@gifu-u.ac.jp

Received March 2014; revised July 2014

Abstract. *This paper proposes an algorithm for checking incorrectness of a rule in equivalent transformation programs. Incorrect rules in programs can be detected without having to execute the program by applying the proposed algorithm to each rule one by one. Incorrectness of a rule can be shown by this algorithm without having to consider interrelations with other rules. Programmers can know if a rule needs to be corrected prior to trying to solve large-scale problems. This leads to a remarkable cost reduction in the construction of correct programs.*

Keywords: Incorrectness, Equivalent transformation rule, Program correctness, Algorithmic debugging

1. **Introduction.** In many cases, whether a part of a program is correct or not is checked by programmers themselves. If the correctness of a program is defined, then this will be checked automatically. However, the correctness of a program is not defined in most conventional computational frameworks and, therefore, checking it automatically is a difficult task in most programs.

In this paper, our interest is with the automatic check of incorrectness in equivalent transformation (ET) programs, the programs found in the ET model [1, 9]. An ET program is a set of prioritized rewriting rules for meaning-preserving transformation of problems, and the problem solving process consists of successive rule applications (see Figure 1). Therefore, programming in the ET model is creation of a set of ET rules.

Detecting bug rules in ET programs are discussed in [12, 13, 14]. Similar to the debugging algorithms [12, 13, 14], those [2, 3, 4, 5, 7, 10, 16, 17, 18, 19, 20, 21] in the logic programming (LP) model [8, 19] are applied when an incorrect answer is obtained as the result of program execution.

This paper gives consideration to automatically check whether a given rule is incorrect or not. In the ET model, the correctness of a program is defined as follows. If each rule in a program transforms each computational state, then each transformation preserves meaning of each computational state. From this, the correctness of each rule can be checked, which is one of the superior features of the ET model. From the correctness, we can conclude that a rule applied to a transformation which does not preserve the meaning is incorrect. Based on this principle, checking incorrectness of a given rule is made possible.

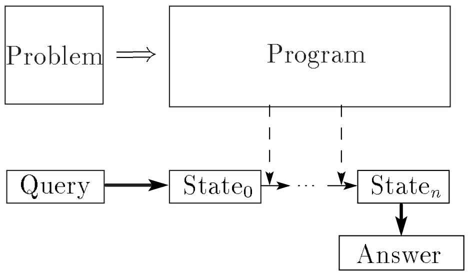

FIGURE 1. Computation in the ET model, where $0 \leq n < \infty$

The purpose of this paper is to propose an algorithm for checking incorrectness of a rule by utilizing a set of correct rules. Its incorrectness can be shown by the proposed algorithm without taking into consideration interrelations with other rules. Hence, an incorrect rule in a program can be detected without having to execute the program by applying the proposed algorithm to each rule one by one. Therefore, programmers can know if a rule needs to be corrected prior to trying to solve large-scale problems. This leads to a remarkable cost reduction in the construction of correct programs.

The main feature of the proposed algorithm is checking the incorrectness of each rule by creating many instances such that

1. the rule can be applied to, and
2. only finite computations are required.

An essential difference between the proposed algorithm and the algorithms in [12, 13, 14] are as follows. In the algorithms in [12, 13, 14], incorrect rules are detected by exploiting program execution processes when an incorrect answer is obtained as the result of program execution. That is, even though a program may contain incorrect rules, these algorithms cannot be applied until the program is complete. In the proposed algorithm, on the other hand, the incorrectness of a rule can be checked without execution of programs. That is, this algorithm can be applied even when a program is not complete. Therefore, in cases where the cost of creating and executing a program is significant, the proposed algorithm, which does not require programs to be complete, is superior because its cost is much lower than the algorithms in [12, 13, 14] for checking the incorrectness of a rule.

2. ET Model.

This section introduces the ET model and describes the approach under consideration in this study.

2.1. Problem setting.

An alphabet $\Delta = \langle C, F_S, V, P \rangle$ is assumed, where C is a set of constants, F_S a set of function symbols, V a set of variables, P a set of predicate symbols. Let $\mathbb{A}$ and $\mathbb{G}$ denote the set of all atoms on P and that of all ground atoms on P, respectively. Let $\mathbb{S}$ denote the set of all substitutions on Δ.

The definite clause c is an expression of the form $a \leftarrow b_1, \ldots, b_m$, where $a, b_1, \ldots, b_m \in \mathbb{A}$ and $m \geq 0$. The atom a is called the head of c, denoted by $head(c)$. The set $\{b_1, \ldots, b_m\}$ is called the body of c, denoted by $body(c)$. When $m = 0$, c is called a unit clause.

Let ϕ and $\mathbb{C}$ denote an empty set and the set of all definite clauses, respectively. For $K \subseteq \mathbb{C}$, a mapping T_K on the power set of $\mathbb{G}$ is defined by

$$\begin{aligned} T_K(G) = \{&head(c\theta) | (c \in K) \,\&\, (\theta \in \mathbb{S}) \\ &\&\, (body(c\theta) \subseteq G) \,\&\, (head(c\theta) \in \mathbb{G})\} \end{aligned}$$

for each $G \subseteq \mathbb{G}$. We define the meaning of K.

Definition 2.1. *For $K \subseteq \mathbb{C}$, the meaning of K, denoted by $\mathcal{M}(K)$, is defined by*

$$\mathcal{M}(K) = \bigcup_{m=1}^{\infty} T_K^m(\phi)$$

where

$$T_K^1 = T_K(\phi)$$
$$T_K^m(\phi) = T_K(T_K^{m-1}(\phi)) \quad (m \geq 2).$$

Let $D \subseteq \mathbb{C}$ represent a problem of interest. Let $\mathbb{Q}$ be a set of atoms which represents the set of all queries for D. Then a problem is given in the form of the pair $\langle D, q\rangle$ where $q \in \mathbb{Q}$.

For any $\alpha \in \mathbb{A}$, let $\text{rep}(\alpha)$ denote the set of all ground instances of α. Then the solution of a problem $\langle D, q\rangle$ is defined as the set $\mathcal{M}(D) \cap \text{rep}(q)$.

2.2. **Creating rules.** Let $\mathbb{U}$ denote the set of all unit clauses, respectively. For an arbitrarily set X, $\text{pow}(X)$ means the power set of X.

A rule r is defined as a relation on $\text{pow}(\mathbb{C})$, i.e., $r \subseteq \text{pow}(\mathbb{C}) \times \text{pow}(\mathbb{C})$. Moreover, for S, $S' \subseteq \mathbb{C}$, a rule r is an ET rule on D, if

$$(S,\ S') \in r \Rightarrow \mathcal{M}(D \cup S) = \mathcal{M}(D \cup S'). \tag{1}$$

A program in the ET model is defined as a set of rules for computing the solution set $\mathcal{M}(D) \cap \text{rep}(q)$. A program on D means a set of rules on D. Therefore, programming in the ET model (ET programming) is defined as the creation of a set of rules.

2.3. **Program execution.** Let R be a program on D. Given D and R, consider computation of the solution set $\mathcal{M}(D) \cap \text{rep}(q)$ for a query $q \in \mathbb{Q}$. In the ET model, the set is computed as follows.

Procedure 1 ([1])**.** *Let $D \subseteq \mathbb{C}$ represent a problem of interest and R be a program on D. This procedure computes the answer set A for D, R and $q \in \mathbb{Q}$.*

Step 1: *Let $Q_0 = \{ans(q) \leftarrow q\}$.*
Step 2: *Initialize i as $i = 1$.*
Step 3: *Obtain Q_i where $\langle Q_{i-1}, Q_i\rangle \in r$ and $r \in R$.*
Step 4: *If $Q_i \subseteq \mathbb{U}$, go to Step 5. Otherwise, renew i as $i = i + 1$ and go back to Step 3.*
Step 5: *Let $A = \bigcup_{(ans(a)\leftarrow)\in Q_i} \text{rep}(a)$. Terminate.*

Figure 2 illustrates the behavior of Procedure 1.

The notation $D : q \xrightarrow{R} A$ means that under domain knowledge D, the answer set A is obtained for a query q using transformation rules in the ET program R. We cite Theorem 2.1 to show that Procedure 1 is correct if R consists of ET rules.

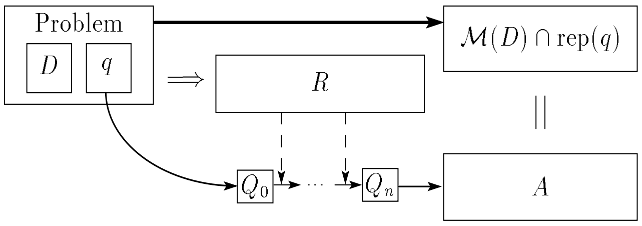

FIGURE 2. The behavior of Procedure 1

Theorem 2.1 ([1]). *Let $D \subseteq \mathbb{C}$ represent a problem of interest and $q \in \mathbb{Q}$. If there exists a set of ET rules R on D such that $D : q \xrightarrow{R} A$, then $\mathcal{M}(D) \cap \mathrm{rep}(q) = A$.*

We can define a correct rule from the above.

Definition 2.2. *A correct rule is an ET rule, i.e., a rule r satisfying (1) for any S, $S' \subseteq \mathbb{C}$.*

2.4. **Situation considered.** From Definition 2.2, an incorrect rule can be defined.

Definition 2.3. *An incorrect rule is a non-ET rule. Namely a rule r is incorrect if there exist at least one pair of S and S' such that*

$$(S,\ S') \in r \quad \text{and} \quad \mathcal{M}(D \cup S) \neq \mathcal{M}(D \cup S'). \tag{2}$$

We show an example of an incorrect rule. Consider the case that r is as follows.

$$\begin{aligned} r : &(avr\ \ {*x}\ \ {*y}) \\ &\Rightarrow (:=\ \ {*y}\ (d\ (\times\ \ {*x}\ \ {*z})\ 2)), (:=\ \ {*z}\ (+\ \ {*x}\ 2)). \end{aligned}$$

$:=$, d, $\times$ and $+$ denote arithmetic-substitution, division, multiplication and addition, respectively. The semantic of $(avr\ \ {*x}\ {*y})$ is ${*y} = ({*x} + ({*x} + 2))/2$, which corresponds to D. Let S be as follows.

$$S : \{(ans\ (avr\ 2\ \ {*y})) \leftarrow (avr\ 2\ \ {*y}).\}$$

S' is obtained by transforming S applying r.

$$\begin{aligned} S' : &\{(ans\ (avr\ 2\ \ {*y})) \\ &\leftarrow (:=\ \ {*y}\ (d\ (\times\ 2\ \ {*z})\ 2)), (:=\ \ {*z}\ (+\ 2\ 2)).\} \end{aligned}$$

Since

$$\begin{aligned} \mathcal{M}(D \cup S) &= \mathcal{M}(D) \cup \{(ans\ (avr\ 2\ 3))\} \\ \mathcal{M}(D \cup S') &= \mathcal{M}(D) \cup \{(ans\ (avr\ 2\ 4))\}, \end{aligned}$$

then $\mathcal{M}(D \cup S) \neq \mathcal{M}(D \cup S')$. Since $\mathcal{M}(D) \cup (ans\ (avr\ 2\ 3)) \neq \mathcal{M}(D) \cup (ans\ (avr\ 2\ 4))$, r is an incorrect rule. In fact, $(ans\ (avr\ 2\ 4))$ contradicts to the semantic, although $(ans\ (avr\ 2\ 3))$ does not contradict to it.

Consider the situation that programmers immediately check a created (or revised) rule for incorrectness in the middle of program creation. In this situation, programs cannot be executed since program creation is not completed. If incorrectness of a rule r can be checked without execution of the program, then programmers can know if there is a need to correct r before trying to solve the problems. By repeating this check and correction, ET rules are piled up so that correct programs are developed. This process leads to a remarkable cost reduction in the construction of correct programs.

3. **Proposed Algorithm.**

This section proposes an algorithm for checking incorrectness of r discussed in Section 2.4. Let $D \subseteq \mathbb{C}$ represent a problem of interest and R_c be a set of ET rules.

3.1. **Checking incorrectness using instances.** In this paper, we consider checking incorrectness of r by creating instances, transforming the instances using r, and checking whether meaning is preserved or not following this transformation. If the meaning is not preserved, then r is a non-ET rule. Therefore, instances are required to show that the meaning is not preserved within the transformation. We consider creating many instances, applying r to these instances, and checking whether the meaning is preserved or not. The reason for creating many instances is that we do not know which instance will detect transformations in which the meaning is not preserved.

The instance is given in the form of $Q \subseteq \mathbb{C}$. If r transforms Q into Q' and $\mathcal{M}(D \cup Q) \neq \mathcal{M}(D \cup Q')$, then r is incorrect. Thus we compute A and A' such that $D \cup Q \xrightarrow{R_c} A$ and $D \cup Q' \xrightarrow{R_c} A'$, respectively, and check whether $A \neq A'$ holds or not.

Q must satisfy the condition to which r can be applied. If $D \cup Q \xrightarrow{R_c} A$ and $D \cup Q' \xrightarrow{R_c} A'$ require infinite computation, we cannot find A and A'. Thus it is necessary that A and A' can only be obtained by finite computations. We define the property of Q such that A and A' must be obtained by finite computations.

Definition 3.1. *$Q \subseteq \mathbb{C}$ is finite if*

1. *Q is not a pair and not variable, or*
2. *Q is a pair whose rest is finite.*

For example, if Q includes lists, then for Q to be finite it is necessary that the length of these lists is finite. Even though Q is finite, it is not necessarily so that $D \cup Q \xrightarrow{R_c} A$ and $D \cup Q' \xrightarrow{R_c} A'$ require finite computation. However, this finiteness can be expected in many programs. We construct an algorithm for automatically generating many finite Q such that r is applicable from head and conditional part of r.

3.2. Overall process.

From Section 3.1, we create an instance Q such that

- r is applicable to Q, and
- Q is finite.

In the proposed algorithm, we use many instances for increasing the possibility that $A \neq A'$ holds. Based on the above discussions, we demonstrate the complete process of the proposed algorithm in Algorithm 1.

Algorithm 1. *This algorithm checks incorrectness of a rule r when R_c is given.*

Step 1: *Generate $Q^{(j)} \subseteq \mathbb{C}$, $j = 1, \ldots, m$ such that r can be applied to $Q^{(j)}$ and $Q^{(j)}$ is finite. Initialize $\boldsymbol{Q}$ as $\boldsymbol{Q} = \{Q^{(1)}, \ldots, Q^{(m)}\}$.*

Step 2: *If $\boldsymbol{Q} = \phi$, then it cannot be checked whether r is correct or incorrect. Terminate. Otherwise choose an element from $\boldsymbol{Q}$. We name the chosen element Q. Eliminate Q from $\boldsymbol{Q}$.*

Step 3: *Compute an answer set A such that $D \cup Q \xrightarrow{R_c} A$.*

Step 4: *Transform Q by applying r. We name the obtained set Q'.*

Step 5: *Compute an answer set A' such that $D \cup Q' \xrightarrow{R_c} A'$.*

Step 6: *If $A \neq A'$, then r is incorrect. Terminate. Otherwise go back to Step 2.*

Figure 3 illustrates the first cycle of Algorithm 1.

We present Theorem 3.1 with respect to the correctness of Algorithm 1.

Theorem 3.1. *If Algorithm 1 asserts that r is incorrect, then r is a non-ET rule.*

Proof: Let D, R_c, Q, Q', A and A' be defined as in Algorithm 1. Since all rules in R_c are ET rules, we have $\mathcal{M}(D \cup Q) = A$ and $\mathcal{M}(D \cup Q') = A'$. This and $A \neq A'$ yield $\mathcal{M}(D \cup Q) \neq \mathcal{M}(D \cup Q')$. □

3.3. Creation of instances – examples.

In the practical execution of Algorithm 1, we need to consider how to execute Step 1, i.e., how to generate $Q^{(1)}, \ldots, Q^{(m)}$. $Q^{(1)}, \ldots, Q^{(m)}$ which must satisfy the condition to which r can be applied. Moreover, to make the computations of A and A' finite, it is necessary that $Q^{(1)}, \ldots, Q^{(m)}$ are finite. We illustrate two examples for creating finite $Q^{(1)}, \ldots, Q^{(m)}$ such that r can be applied to.

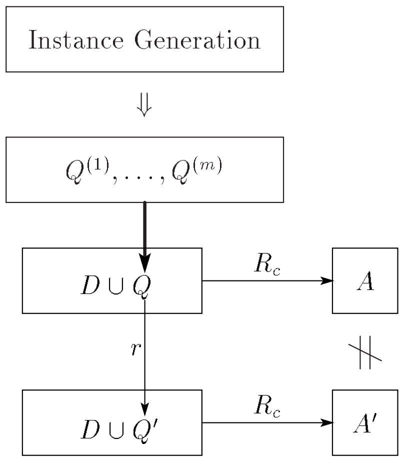

FIGURE 3. The first cycle of Algorithm 1

3.3.1. *Example 1.* Consider the case that r is given as follows.

$$\begin{aligned}&(app\ (*a|\ *X)\ \ *Y\ \ *Z)\\ &\quad \Rightarrow \{(=\ *Z\ (*Z1))\},\ (app\ \ *X\ \ *Y\ \ *Z1).\end{aligned}$$

app predicate means that the combined elements of the first and second arguments will now form the third argument's elements. The left-hand side of $\Rightarrow$ is called a "head". It stands for the proposition before replacement. The right-hand side of $\Rightarrow$ is called a "body". It stands for the proposition obtained by replacing the proposition in the head. The description surrounded by the most external pair of brackets is one atom. In the list $(*a|\ *A)$, $*a$ and $*A$ mean the first element and the set of elements after the second, respectively. An atom in $\{\}$ in body denotes execution of procedure and $=$ means unification. In this case, clauses included in $Q^{(1)}, \dots, Q^{(m)}$ must have at least one body atom which matches with the atom $(app\ (*a|\ *X)\ \ *Y\ \ *Z)$. Moreover, computations of A and A' are made easier if the length of the list $(*a|\ *X)$ is finite. Thus we introduce the following rule.

$$\begin{aligned}(rst\ \ *T)\ \ &\Rightarrow\ \ (=\ *T\ ());\\ &\Rightarrow\ \ (=\ *T\ (?));\\ &\Rightarrow\ \ (=\ *T\ (?\ ?)).\end{aligned} \tag{3}$$

This rule represents the splitting of one head into three bodies. We create the following rule to apply (3).

$$(chk\ ((app\ (*a|\ *X)\ \ *Y\ \ *Z))) \Rightarrow (rst\ \ *X). \tag{4}$$

Then the length of the list $(*a|*X)$ becomes one, two or three. By increasing the variety of the list length, i.e., by increasing the number of bodies in (3), m increases so that we can obtain many elements in $\boldsymbol{Q}$. This increases the possibility that Algorithm 1 terminates with $A \neq A'$. On the other hand, the computational cost also increases since A and A' must be computed for many elements in $\boldsymbol{Q}$. By giving a query

$$(chk\ ((app\ (*a|\ *X)\ \ *Y\ \ *Z))) \tag{5}$$

to the set of (3) and (4), we obtain the following answers.

$$\begin{aligned}&(ans\ (chk\ ((app\ (*A\ \ *B\ \ *C)\ \ *D\ \ *E)))) \leftarrow .\\ &(ans\ (chk\ ((app\ (*F\ \ *G)\ \ *H\ \ *I)))) \leftarrow .\\ &(ans\ (chk\ ((app\ (*J)\ \ *K\ \ *L)))) \leftarrow .\end{aligned} \tag{6}$$

From these answers, we can obtain initial $\boldsymbol{Q}$ as

$$\begin{aligned}\boldsymbol{Q} = \{&\{(app\ (*A\ \ *B\ \ *C)\ \ *D\ \ *E)\\ &\quad \leftarrow (app\ (*A\ \ *B\ \ *C)\ \ *D\ \ *E).\}\\ &\{(app\ (*F\ \ *G)\ \ *H\ \ *I)\\ &\quad \leftarrow (app\ (*F\ \ *G)\ \ *H\ \ *I).\}\\ &\{(app\ (*J)\ \ *K\ \ *L)\\ &\quad \leftarrow (app\ (*J)\ \ *K\ \ *L).\}\}\end{aligned} \tag{7}$$

3.3.2. *Example 2.* Consider the case that r is given as follows:

$$\begin{aligned}&(app\ \ *H\ (*b|*T)\ (*a|*Z)), \{(not\ (test{=}\ \ *b\ \ *a))\}\\ &\Rightarrow \{(=\ \ *H\ (*a))\},\ (app\ \ *h\ (*b|*T)\ \ *Z).\end{aligned}$$

An atom in $\{\}$ in the head denotes a condition for application of a rule. *not* is negation. $(test{=}\ \ *b\ *a)$ means that $*a$ can be unified with $*b$. In this case, clauses included in $Q^{(1)}, \ldots, Q^{(m)}$ must satisfy the condition $(not\ (test{=}\ \ *b\ \ *a))$ adding that they must have at least one body atom which matches with the atom $(app\ \ *H\ (*b|*T)\ (*a|*Z))$. We introduce the following rule to make clauses included in $Q^{(1)}, \ldots, Q^{(m)}$ satisfy this condition:

$$(not\ (test{=}\ \ *b\ \ *a)) \Rightarrow \{(=\ \ *b\ b), (=\ \ *a\ a)\}. \tag{8}$$

Moreover, we use (3) for making the length of $(*b|*T)$ and $(*a|*Z)$ finite. We create the following rule to apply (3) and (8).

$$\begin{aligned}&(chk\ ((app\ \ *H\ (*b|*T)\ (*a|*Z))))\\ &\Rightarrow (not\ (test{=}\ \ *b\ \ *a)), (rst\ \ *T),\ (rst\ \ *Z).\end{aligned} \tag{9}$$

By giving a query

$$(chk\ ((app\ \ *H\ (*b|*T)\ (*a|*Z)))) \tag{10}$$

to the set of (3), (8) and (9), we obtain the following answers.

$$\begin{aligned}&(ans\ (chk\ ((app\ \ *A\ (b)\ (a\ \ *B\ \ *C))))) \leftarrow .\\ &(ans\ (chk\ ((app\ \ *D\ (b)\ (a\ \ *E))))) \leftarrow .\\ &(ans\ (chk\ ((app\ \ *F\ (b)\ (a))))) \leftarrow .\\ &\qquad\vdots\\ &(ans\ (chk\ ((app\ \ *Y\ (b\ \ *A\ \ *B)\ (a))))) \leftarrow .\end{aligned} \tag{11}$$

From these answers, we can obtain initial $\boldsymbol{Q}$ as

$$\begin{aligned}\boldsymbol{Q} = \{&\{(app\ \ *A\ (b)\ (a\ \ *B\ \ *C))\\ &\quad \leftarrow (app\ \ *A\ (b)\ (a\ \ *B\ \ *C)).\}\\ &\{(app\ \ *D\ (b)\ (a\ \ *E))\\ &\quad \leftarrow (app\ \ *D\ (b)\ (a\ \ *E)).\}\\ &\{(app\ \ *F\ (b)\ (a)) \leftarrow (app\ \ *F\ (b)\ (a)).\}\\ &\qquad\vdots\\ &\{(app\ \ *Y\ (b\ \ *A1\ \ *B1)\ (a))\\ &\quad \leftarrow (app\ \ *Y\ (b\ \ *A1\ \ *B1)\ (a)).\}\}.\end{aligned} \tag{12}$$

3.4. **Creation of instances – general algorithm.** The core of the techniques introduced in Section 3.3 are

- Creating definite clauses to which r can be applied by utilizing the patterns of atoms in the head of r,
- Creating finite sets of definite clauses by restricting list length,
- Creating definite clauses satisfying the condition in r by preparing concrete instances.

We obtain Algorithm 2 by systematizing these core elements.

Algorithm 2. *Let R_c be a set of ET rules. This algorithm computes $Q^{(1)}, \ldots, Q^{(m)}$ in Algorithm 1.*

Step 1: *If some variables in r are lists, then create a set R_f of rules for making the list length finite. For example, (3) is a rule in R_f. Otherwise $R_f = \phi$.*
Step 2: *If r includes conditional parts, then create a set R_s of rules for making clauses in $Q^{(1)}, \ldots, Q^{(m)}$ satisfy the conditions. For example, (8) is a rule in R_s. See (17) for more examples. Otherwise $R_s = \phi$.*
Step 3: *Create a rule r_Q for obtaining head and body atoms of clauses in $Q^{(1)}, \ldots, Q^{(m)}$ utilizing rules in R_f and R_s. For example, (4) and (9) are r_Q.*
Step 4: *Create a query q_Q from the head of r_Q. For example, (5) and (10) are q_Q.*
Step 5: *Compute an answer set A_Q by giving q_Q to the set $R_Q = \{r_Q\} \cup R_f \cup R_s$ of rules. For example, (6) and (11) are A_Q.*
Step 6: *Create $Q^{(1)}, \ldots, Q^{(m)}$ from A_Q. See (7) and (12) for example.*

Note that some rules in R_f and R_s can be reused once created. (17) in Appendix are the examples of rules in R_s for built-in atoms frequently used in conditional parts. For further information on built-in atoms, please refer to [22] in REFERENCES.

4. Examples and Verification.

In this section, we verify the effectiveness of Algorithm 1 through some examples. In all examples, the test rules are incorrect rules. We adopted Algorithm 2 for executing Step 1 in Algorithm 1 in all examples. In Step 1 of Algorithm 2, we created R_f including (3) only for all examples except Sections 4.6 and 4.7.

4.1. Example 1.

Consider the case that R_c in Algorithm 1 is given and we check incorrectness of the rule r_1, where R_c is the following set of rules:

$$
\begin{array}{l}
(app\ *H\ *T\ ()) \Rightarrow \{(=\ *H\ ()),\ (=\ *T\ ())\}.\\
(app\ *H\ ()\ *Z) \Rightarrow \{(=\ *H\ *Z)\}.\\
(app\ ()\ *Y\ *Z) \Rightarrow \{(=\ *Y\ *Z)\}.\\
(app\ (*a|\ *X)\ \ *Y\ \ *Z)\\
\quad \Rightarrow \{(=\ *Z\ (*a|\ *Z1))\},\ (app\ *X\ *Y\ *Z1).\\
(app\ *H\ (*b|\ *T)\ (*a|\ *Z))\\
\quad \Rightarrow \{(=\ *H\ (*a|\ *h))\},\ (app\ *h\ (*b|\ *T)\ \ *Z);\\
\quad \Rightarrow \{(=\ *H\ ()),\ (=\ (*b|\ *T)\ (*a|\ *Z))\}.\\
(zeros\ ()) \Rightarrow \{(false)\}.\\
(zeros\ (*a)) \Rightarrow \{(=\ *a\ 0)\}.\\
(zeros\ (*a\ \ *b|\ *X))\\
\quad \Rightarrow \{(=\ *a\ 0)\},\ (zeros\ (*b|\ *X)).\\
(zeros\ *X),\ \{(pvar\ *X)\}\\
\quad \Rightarrow \{(=\ *X\ (0|\ *x))\},\ (zeros\ *X)
\end{array}
$$

and r_1 is given as follows:

$$r_1 : (zeros\ (*a|\ *K)),\ (app\ \ *K\ (1|\ *M)\ (1|\ *R))$$
$$\Rightarrow \{(=\ *K\ ()),\ (=\ (1|\ *M)\ (1|\ *R))\}.$$

zeros predicate will succeed if there are elements in the first argument which consist of a list containing more than one 0. It fails (false) when the list is empty. In Step 2 of Algorithm 2, $R_s = \phi$ since r_1 does not include a conditional part. In Step 3 of Algorithm 2, r_Q is as follows.

$$(chk\ ((zeros\ (*a|\ *K))\ (app\ \ *K\ (1|\ *M)\ (1|\ *R))))$$
$$\Rightarrow (rst\ \ *K), (rst\ \ *M), (rst\ \ *R).$$

In Step 4 of Algorithm 2, q_Q is as follows.

$$(chk\ ((zeros\ (*a|\ *K))\ (app\ \ *K\ (1|\ *M)\ (1|\ *R)))).$$

In Step 5 of Algorithm 2, A_Q is computed as the set of following answers.

$$(ans\ (chk\ ((zeros\ (*A))$$
$$(app\ ()\ (1)\ (1\ \ *B\ \ *C))))) \leftarrow .$$
$$(ans\ (chk\ ((zeros\ (*D))\ (app\ ()\ (1)\ (1\ \ *E))))) \leftarrow .$$
$$(ans\ (chk\ ((zeros\ (*F))\ (app\ ()\ (1)\ (1))))) \leftarrow .$$
$$\vdots$$
$$(ans\ (chk\ ((zeros\ (*D\ \ *E\ \ *F))$$
$$(app\ (*E\ \ *F)\ (1\ \ *G\ \ *H)\ (1))))) \leftarrow .$$

In Step 6 of Algorithm 2, $\boldsymbol{Q}$ is created as the set of following sets.

$$\{(zeros\ (*A)),\ (app\ ()\ (1)\ (1\ \ *B\ \ *C))$$
$$\leftarrow (zeros\ (*A)),\ (app\ ()\ (1)\ (1\ \ *B\ \ *C)).\}$$
$$\{(zeros\ (*D)),\ (app\ ()\ (1)\ (1\ \ *E))$$
$$\leftarrow (zeros\ (*D)),\ (app\ ()\ (1)\ (1\ \ *E)).\}$$
$$\{(zeros\ (*F)),\ (app\ ()\ (1)\ (1))$$
$$\leftarrow (zeros\ (*F)),\ (app\ ()\ (1)\ (1)).\}$$
$$\vdots$$
$$\{(zeros\ (*D\ \ *E\ \ *F)),$$
$$(app\ (*E\ \ *F)\ (1\ \ *G\ \ *H)\ (1))$$
$$\leftarrow (zeros\ (*D\ \ *E\ \ *F)),$$
$$(app\ (*E\ \ *F)\ (1\ \ *G\ \ *H)\ (1)).\}$$

We proceed to Step 2 in Algorithm 1. In Step 2 of Algorithm 1, we choose Q as the third set in $\boldsymbol{Q}$, i.e.,

$$Q = \{(zeros\ (*F)),\ (app\ ()\ (1)\ (1))$$
$$\leftarrow (zeros\ (*F)),\ (app\ ()\ (1)\ (1)).\}$$

In Step 3 of Algorithm 1, A is computed as

$$A = \{(ans\ ((zeros\ (0))\ (app\ ()\ (1)\ (1)))) \leftarrow .\}$$

In Step 4 of Algorithm 1, Q' is obtained as

$$Q' = \{(ans\ ((zeros\ (*A))\ (app\ ()\ (1)\ (1)))) \leftarrow .\}$$

In Step 5 of Algorithm 1, A' is obtained as

$$A' = \{(ans\ ((zeros\ (*A))\ (app\ ()\ (1)\ (1)))) \leftarrow .\}$$

In this case, no transformation is needed for computing A', since $Q' \subseteq \mathbb{U}$. Algorithm 1 terminates with $A \neq A'$ in Step 6. Thus incorrectness of r_1 is successfully checked by Algorithms 1 and 2. On the other hand, if we choose Q as the first, second and last set in $\boldsymbol{Q}$, then $A \neq A'$ does not hold.

4.2. **Example 2.** Consider the case that R_c is given and we check incorrectness of the rule r_2, where R_c is the following set of rules:

$$\begin{aligned}
&(app\ \ {*H}\ \ {*T}\ ()) \Rightarrow \{(=\ \ {*H}\ ()),\ (=\ \ {*T}\ ())\}.\\
&(app\ \ {*H}\ ()\ \ {*Z}) \Rightarrow \{(=\ \ {*H}\ \ {*Z})\}.\\
&(app\ ()\ \ {*Y}\ \ {*Z}) \Rightarrow \{(=\ \ {*Y}\ \ {*Z})\}.\\
&(app\ ({*a}|\ {*X})\ \ {*Y}\ \ {*Z})\\
&\quad \Rightarrow \{(=\ \ {*Z}\ ({*a}|\ {*Z1}))\},\ (app\ \ {*X}\ \ {*Y}\ \ {*Z1}).\\
&(app\ \ {*H}\ ({*b}|\ {*T})\ ({*a}|\ {*Z}))\\
&\quad \Rightarrow \{(=\ \ {*H}\ ({*a}|\ {*h}))\},\ (app\ \ {*h}\ ({*b}|\ {*T})\ \ {*Z});\\
&\quad \Rightarrow \{(=\ \ {*H}\ ()),\ (=\ \ ({*b}|\ {*T})\ ({*a}|\ {*Z}))\}
\end{aligned}$$

and r_2 is given as follows:

$$\begin{aligned}
&r_2 : (app\ \ {*H}\ ({*a}\ \ {*b}|\ {*T})\ ({*c}\ \ {*d}|\ {*Z})),\\
&\quad \{(not\ (test{=}\ \ ({*a}\ \ {*b})\ ({*c}\ \ {*d})))\}\\
&\quad\ \Rightarrow \{(=\ \ {*H}\ ({*c}|\ {*h}))\}, (app\ \ {*h}\ ({*a}\ \ {*b})\ ({*d}|\ {*Z})).
\end{aligned}$$

In Step 2 of Algorithm 2, we create R_s as a set of the following rules.

$$\begin{aligned}
&(not\ (test{=}\ \ ({*a}\ \ {*b})\ ({*c}\ \ {*d})))\\
&\quad \Rightarrow (not\ (test{=}\ \ {*a}\ \ {*c}));\\
&\quad \Rightarrow (not\ (test{=}\ \ {*b}\ \ {*d})).\\
&(not\ (test{=}\ \ {*b}\ \ {*a})) \Rightarrow \{(=\ \ {*b}\ b), (=\ \ {*a}\ a)\}.
\end{aligned} \tag{13}$$

In Step 3 of Algorithm 2, r_Q is as follows.

$$\begin{aligned}
&(chk\ ((app\ \ {*H}\ ({*a}\ \ {*b}|\ {*T})\ ({*c}\ \ {*d}|\ {*Z}))))\\
&\quad \Rightarrow (not\ (test{=}\ \ ({*a}\ \ {*b})\ ({*c}\ \ {*d}))),\\
&\qquad (rst\ \ {*T}),\ (rst\ \ {*Z}).
\end{aligned}$$

In Step 4 of Algorithm 2, q_Q is as follows.

$$(chk\ ((app\ \ {*H}\ ({*a}\ \ {*b}|\ {*T})\ ({*c}\ \ {*d}|\ {*Z})))).$$

In Step 5 of Algorithm 2, A_Q is computed as the set of following answers.

$$\begin{aligned}
&(ans\ (chk\ ((app\ \ {*A}\ (b\ \ {*B})\ (a\ \ {*C}))))) \leftarrow .\\
&(ans\ (chk\ ((app\ \ {*D}\ (b\ \ {*E})\ (a\ \ {*F}\ \ {*G}))))) \leftarrow .\\
&(ans\ (chk\ ((app\ \ {*V}\ (b\ \ {*W}\ \ {*X})\\
&\ (a\ \ {*Y}\ \ {*A}\ \ {*B}))))) \leftarrow .\\
&\qquad\qquad \vdots\\
&(ans\ (chk\ ((app\ \ {*I}\ ({*J}\ b\ \ {*K}\ \ {*L})\\
&\ ({*M}\ a\ \ {*N}\ \ {*O}))))) \leftarrow .
\end{aligned}$$

In Step 6 of Algorithm 2, $\boldsymbol{Q}$ is created as the set of following sets.

$$
\begin{aligned}
&\{(app\ *A\ (b\ *B)\ (a\ *C)))\\
&\quad \leftarrow (app\ *A\ (b\ *B)\ (a\ *C))).\}\\
&\{(app\ *D\ (b\ *E)\ (a\ *F\ *G))\\
&\quad \leftarrow (app\ *D\ (b\ *E)\ (a\ *F\ *G)).\}\\
&\{(app\ *V\ (b\ *W\ *X)\ (a\ *Y\ *A\ *B))\\
&\quad \leftarrow (app\ *V\ (b\ *W\ *X)\ (a\ *Y\ *A\ *B)).\}\\
&\qquad\qquad \vdots\\
&\{(app\ *I\ (*J\ b\ *K\ *L)\ (*M\ a\ *N\ *O))\\
&\quad \leftarrow (app\ *I\ (*J\ b\ *K\ *L)\ (*M\ a\ *N\ *O)).\}.
\end{aligned}
$$

We proceed to Step 2 in Algorithm 1. In Step 2 of Algorithm 1, we choose Q as the third set in $\boldsymbol{Q}$, i.e.,

$$
\begin{aligned}
Q = \{&(app\ *V\ (b\ *W\ *X)\ (a\ *Y\ *A\ *B))\\
&\leftarrow (app\ *V\ (b\ *W\ *X)\ (a\ *Y\ *A\ *B)).\}.
\end{aligned}
$$

In Step 3 of Algorithm 1, A is computed as

$$
\begin{aligned}
A = \{&(ans\ ((app\ (a)\ (b\ *A\ *B)\\
&(a\ b\ *A\ *B)))) \leftarrow .\}.
\end{aligned}
$$

In Step 4 of Algorithm 1, Q' is obtained as

$$
\begin{aligned}
Q' = \{&(app\ (a \mid *A)\ (b\ *B\ *C)\ (a\ *D\ *E\ *F))\\
&\leftarrow (app\ *A\ (b\ *B)\ (*D\ *E\ *F)).\}.
\end{aligned}
$$

In Step 5 of Algorithm 1, A' is obtained as

$$
\begin{aligned}
A' = \{&(ans\ ((app\ (a\ *A)\ (b\ *B\ *C)\\
&(a\ *A\ b\ *B)))) \leftarrow .\}.
\end{aligned}
$$

Algorithm 1 terminates with $A \neq A'$ in Step 6. Thus incorrectness of r_2 is successfully checked by Algorithms 1 and 2. On the other hand, if we choose Q as the first, second and last set in $\boldsymbol{Q}$, then $A \neq A'$ does not hold.

For Section 4.3 to Section 4.10, because incorrectness of a rule is checked in the same manner as Section 4.1 and Section 4.2, detailed explanation of their processes is omitted.

4.3. **Example 3.** Consider the case that we check incorrectness of the following rule.

$$
\begin{aligned}
r_3 : &(app\ *H\ (*b \mid *T)\ (*a \mid *Z)),\\
&\{(not\ (test{=}\ *b\ *a))\}\\
&\quad \Rightarrow \{(=\ *H\ (*a))\},\ (app\ *h\ (*b \mid *T)\ *Z).
\end{aligned}
$$

Note that this rule was presented in Section 3.3.2. In Sections 4.3, 4.4 and 4.5, R_c is similar to that of Section 4.2. We set R_s similarly to Section 4.2. Then incorrectness of r_3 is successfully checked by Algorithms 1 and 2.

4.4. **Example 4.** Consider the case that we check incorrectness of the following rule.

$$\begin{aligned} r_4 : &(app\ \ *H\ (*a\ \ *b\ \ *c|*T)\ (*d\ \ *e\ \ *f|*Z)),\\ &\{(not\ (test{=}\ \ (*a\ \ *b\ \ *c)\ (*d\ \ *e\ \ *f)))\}\\ &\Rightarrow \{(=\ \ *H\ (*d|*h))\},\\ &(app\ *h\ (*a\ \ *b\ \ *c|*T)\ (*e|*Z)). \end{aligned}$$

We set R_s as the union of (13) and the following rule.

$$\begin{aligned} &(not\ (test{=}\ \ (*a\ \ *b\ \ *c)\ (*d\ \ *e\ \ *f)))\\ &\Rightarrow (not\ (test{=}\ \ *a\ \ *d));\\ &\Rightarrow (not\ (test{=}\ \ *b\ \ *e));\\ &\Rightarrow (not\ (test{=}\ \ *c\ \ *f)). \end{aligned}$$

Then incorrectness of r_4 is successfully checked by Algorithms 1 and 2.

4.5. **Example 5.** Consider the case that we check incorrectness of the following rule.

$$\begin{aligned} r_5 : &(app\ \ *H\ (*a\ \ *b\ \ *c\ \ *d|*T)\ (*e\ \ *f\ \ *g\ \ *h|*Z)),\\ &\{(not\ (test{=}\ \ (*a\ \ *b\ \ *c\ \ *d)\ (*e\ \ *f\ \ *g\ \ *h)))\}\\ &\Rightarrow \{(=\ \ *H\ (*e|*h))\},\\ &(app\ *h\ (*a\ \ *b\ \ *c|*T)\ (*f\ \ *g\ \ *h|*Z)). \end{aligned}$$

We set R_s as the union of (13) and the following rule.

$$\begin{aligned} &(not\ (test{=}\ \ (*a\ \ *b\ \ *c\ \ *d)\ (*e\ \ *f\ \ *g\ \ *h)))\\ &\Rightarrow (not\ (test{=}\ \ *a\ \ *e));\\ &\Rightarrow (not\ (test{=}\ \ *b\ \ *f));\\ &\Rightarrow (not\ (test{=}\ \ *c\ \ *g));\\ &\Rightarrow (not\ (test{=}\ \ *d\ \ *h)). \end{aligned}$$

Then incorrectness of r_5 is successfully checked by Algorithms 1 and 2.

4.6. **Example 6.** Consider the case that R_c is given and we check incorrectness of the rule r_6, where R_c is the following set of rules.

$$\begin{aligned} &(sum\ \ *A\ \ *B), \{(number\ \ *A)\}\\ &\Rightarrow \{(==\ \ *A\ 1), (=\ \ *B\ 1)\};\\ &\Rightarrow \{(>\ \ *A\ 1)\}, (mns1\ \ *A\ \ *C), (pls\ \ *A\ \ *D\ \ *B),\\ &\quad (sum\ \ *C\ \ *D).\\ &(mns1\ \ *A\ \ *B), \{(number\ \ *A)\}\\ &\Rightarrow \{(:=\ \ *B\ (-\ \ *A\ 1))\}.\\ &(pls\ \ *A\ \ *B\ \ *C), \{(number\ \ *A), (number\ \ *B)\}\\ &\Rightarrow \{(:=\ \ *C\ (+\ \ *A\ \ *B))\} \end{aligned} \tag{14}$$

and r_6 is given as follows.

$$\begin{aligned} r_6 : &(sum\ \ *A *B), \{(number\ \ *A)\}\\ &\Rightarrow \{(:= *B\ (d\ (+ *A\ 1)\ 2))\}. \end{aligned}$$

sum predicate indicates that when the first argument is a number, the sum of the first argument's number and all integers that exist between the first argument's number and the number 1 will form the second argument's value. For example, if the first argument's element is 3, its sum of $3 + 2 + 1$, which is 6, will form the second argument's value. When the first argument is a number, *mns*1 predicate indicates that the value obtained by subtracting 1 from the first argument's number will form the second argument's value. When the first and second arguments are a number, *pls* predicate indicates that the sum of these arguments' numbers will form the third argument's value. In this example, $R_f = \phi$ since all of variables in the head of r_6 are not in the list. We created R_s as a set of the following rule.

$$\begin{aligned}(number\ \ *A) &\Rightarrow \{(=\ \ *A\ 1)\};\\ &\Rightarrow \{(=\ \ *A\ 2)\};\\ &\Rightarrow \{(=\ \ *A\ 3)\}.\end{aligned} \tag{15}$$

Then incorrectness of r_6 is successfully checked by Algorithms 1 and 2.

4.7. **Example 7.** Consider the case that R_c is given and we check incorrectness of the rule r_7, where R_c is the union of (14) and the following rules.

$$\begin{aligned}&(sqsum\ \ *A\,B\ \ *C), \{(number\ \ *A), (number\ \ *B)\}\\ &\ \Rightarrow \{(==\ \ *A\ \ *B), (:=\ *C\ (\times *A\ \ *A))\};\\ &\ \Rightarrow \{(<\ \ *A\ \ *B)\}, (mns1\ \ *B\ \ *D),\\ &\quad (sqpls\ \ *B\ \ *E\ \ *C), (sqsum\ \ *A\ \ *D\ \ *E).\\ &(sqpls\ \ *A *B *C), \{(number\ \ *A), (number\ \ *B)\}\\ &\ \Rightarrow \{(:=\ \ *C\ (+\ (\times\ \ *A\ \ *A)\ \ *B))\}\end{aligned}$$

and r_7 is given as follows.

$$\begin{aligned}&r_7 : (sqsum\ \ *A\ \ *B\ \ *C),\\ &\{(number\ \ *A), (number\ \ *B), (<=\ \ *A\ \ *B)\}\\ &\ \Rightarrow \{(:=\ \ *D\ (+\ \ *A\ 1)), (:=\ \ *E\ (+\ (\times\ \ *A\ 2)\ 1)),\\ &\quad (:=\ \ *F\ (d\ (\times\ (\times\ \ *D\ \ *E)\ \ *A)\ 6)),\\ &\quad (:=\ \ *G\ (+\ \ *B\ 1)), (:=\ \ *H\ (+\ (\times\ \ *B\ 2)\ 1)),\\ &\quad (:=\ \ *I\ (d\ (\times\ (\times\ \ *G\ \ *H)\ \ *B)\ 6)),\\ &\quad (:=\ \ *C\ (-\ \ *I\ \ *F))\}.\end{aligned}$$

sqsum predicate is the sum of the squares. *sqpls* predicate indicates that when the first and second arguments are a number, the sum of the second argument's number and the square of the first argument's number will form the third argument's value.

Similarly to Section 4.6, $R_f = \phi$ since all of variables in the head of r_7 are not in the list. We create R_s as the union of (15) and the following rule.

$$\begin{aligned}&(<=\ \ *A\ \ *B), \{(number\ \ *A), (number\ \ *B)\}\\ &\quad \Rightarrow \{(<=\ \ *A\ \ *B)\}.\end{aligned}$$

Then incorrectness of r_7 is successfully checked by Algorithms 1 and 2.

4.8. **Example 8.** Consider the case that R_c is given and we check incorrectness of the rule r_8, where R_c is the following set of rules.

$$\begin{aligned}&(mbr\ \ *X\ ()) \Rightarrow \{(false)\}.\\ &(mbr\ \ *X\ (*A| *L)) \Rightarrow \{(=\ \ *X\ \ *A)\};\\ &\qquad\qquad\qquad\qquad\quad \Rightarrow (mbr\ \ *X\ \ *L)\end{aligned}$$

and r_8 is given as follows. *mbr* predicate means that elements in the first argument are contained in the element list of the second argument.

$$\begin{aligned}
&r_8 : (mbr\ \ *X\ \ *L),\\
&\{(rdc\ \ *X\ \ *L\ \ *R\ \ *F), (==\ \ *F\ on)\}\\
&\ \Rightarrow (mbr\ \ *X\ \ *R).\\
&(rdc\ \ *X\ (*A|*L)\ \ *R\ \ *F), \{(=\ \ *X\ \ *A)\}\\
&\ \rightarrow (=\ \ *R\ (*A|*S)), (rdc\ \ *X\ \ *L\ \ *S\ \ *F).\\
&(rdc\ \ *X\ (*A|*L)\ \ *R\ \ *F)\\
&\ \rightarrow (=\ \ *F\ on), (rdc\ \ *X\ \ *L\ \ *R\ \ *F).\\
&(rdc\ \ *X\ ()\ \ *R\ \ *F) \rightarrow (=\ \ *R\ ()).
\end{aligned} \tag{16}$$

rdc predicate indicates that unification of elements taken from the first and second arguments will form the third argument's elements. When there are elements in the first and second arguments that cannot be unified, the fourth argument's flag switches to "on". Note that the rule including $\rightarrow$ is added to execute the procedure in $\{\}$. We create R_s as a set of the rules falling under (18) in Appendix.

Then, the incorrectness of r_8 is successfully checked by Algorithms 1 and 2.

4.9. **Example 9.** Consider the case that R_c is given and we check incorrectness of the rule r_9, where R_c is the following set of rules. *count* predicate serves to find how many elements that belong to the first argument are also found within the second argument list. That number, when found, will form the third argument's value. *lgth* predicate indicates that a count of the number of elements in the first argument will form the second argument's value.

$$\begin{aligned}
&(count\ \ *n\ ()\ \ *x) \Rightarrow \{(=\ \ *x\ 0)\}.\\
&(count\ \ *n\ (*a|*A)\ \ *x)\\
&\ \Rightarrow \{(=\ \ *n\ \ *a)\}, (pls1\ \ *y\ \ *x), (count\ \ *n\ \ *A\ \ *y);\\
&\ \Rightarrow (not\ (=\ \ *n\ \ *a)), (count\ \ *n\ \ *A\ \ *x).\\
&(pls1\ \ *A\ \ *B), \{(number\ \ *A)\}\\
&\ \Rightarrow \{(:=\ \ *B\ (+\ \ *A\ 1))\}.\\
&(not\ (=\ \ *a\ \ *b)), \{(ground\ \ *a), (ground\ \ *b)\}\\
&\ \Rightarrow \{(not\ (=\ \ *a\ \ *b))\}.\\
&(ground\ \ *a), \{(var\ \ *a)\} \rightarrow (false).\\
&(ground\ \ *a) \rightarrow .\\
&(lgth\ ()\ \ *N) \Rightarrow \{(=\ \ *N\ 0)\}.\\
&(lgth\ (*L|*M)\ \ *N)\\
&\ \Rightarrow (pls1\ \ *P\ \ *N), (lgth\ \ *M\ \ *P).
\end{aligned}$$

$(var * a)$ means that $*a$ is a variable, and r_9 is given as follows. *grnds* predicate will succeed when all elements in a list are constants or symbols; it will fail when variables, even if only one, are found in a list.

$$\begin{aligned}
&r_9 : (count\ \ *A\ \ *L\ \ *C),\\
&\{(grnds\ \ *L), (rdc\ \ *A\ \ *L\ \ *N\ \ *F), (==\ \ *F\ on)\}
\end{aligned}$$

$$\begin{aligned}
&\Rightarrow (length\ \ *N\ \ *C).\\
&(grnds\ ()) \to .\\
&(grnds\ (*L|*M)), \{(var\ \ *L)\} \to (false).\\
&(grnds\ (*L|*M)) \to (grnd\ s *M).
\end{aligned}$$

$(rdc\ *A\ *L\ *N\ *F)$ is similar to (16). We created R_s as the union of (18) in Appendix and the following rules:

$$\begin{aligned}
&(grnds\ ()) \Rightarrow .\\
&(grnds\ (*a|*b)) \Rightarrow \{(=\ \ *a\ a)\}, (grnds\ \ *b);\\
&\qquad\qquad\qquad\ \ \Rightarrow \{(=\ \ *a\ b)\}, (grnds\ \ *b);\\
&\qquad\qquad\qquad\ \ \Rightarrow \{(=\ \ *a\ c)\}, (grnds\ \ *b).
\end{aligned}$$

Then $A \neq A'$ does not hold for all generated instances. Thus incorrectness of r_9 is not checked by Algorithms 1 and 2.

4.10. **Example 10.** Consider the case that R_c is given and we check incorrectness of the rule r_{10}, where R_c is the following set of rules. min predicate indicates that the smallest value in the first argument list will form the second argument's value.

$$\begin{aligned}
&(\min\ ()\ \ *B) \Rightarrow \{(false)\}.\\
&(\min\ (*A)\ \ *B) \Rightarrow \{(= *A\ \ *B)\}.\\
&(\min\ (*A\ \ *B|*X)\ \ *m)\\
&\ \Rightarrow (\min\ (*B|*X)\ \ *n), (\min 2\ \ *A\ \ *n\ \ *m).\\
&(\min 2\ \ *A\ \ *B\ \ *C)\\
&\ \Rightarrow \{(=\ \ *A\ \ *X), (=\ \ *B\ \ *Y), (=\ \ *C\ \ *X)\}, (<\ \ *X\ \ *Y);\\
&\ \Rightarrow \{(=\ \ *A\ \ *X), (=\ \ *B\ \ *Y), (=\ \ *C\ \ *Y)\}, (>=\ \ *X\ \ *Y).\\
&(<\ \ *X\ \ *Y), \{(number\ \ *X), (number\ \ *Y)\}\\
&\ \Rightarrow \{(<\ \ *X\ \ *Y)\}.\\
&(>=\ \ *X\ \ *Y), \{(number\ \ *X), (number\ \ *Y)\}\\
&\ \Rightarrow \{(>=\ \ *X\ \ *Y)\}
\end{aligned}$$

and r_{10} is given as follows.

$$\begin{aligned}
&r_{10} : (\min\ *L\ \ *m),\\
&\{(number\ \ *m), (del\ \ *m\ \ *L\ \ *N\ \ *F), (==\ \ *F\ on)\}\\
&\ \ \Rightarrow (\min\ *N\ \ *m)
\end{aligned}$$

where

$$\begin{aligned}
&(del\ \ *m\ ()\ \ *N\ \ *F) \to (=\ \ *N\ ()).\\
&(del\ \ *m\ (*L|*M)\ \ *N\ \ *F),\\
&\{(number\ \ *L), (<=\ \ *m\ \ *L)\}\\
&\ \to (=\ \ *F\ on), (del\ \ *m\ \ *M\ \ *N\ \ *F).\\
&(del\ \ *m\ (*L|*M)\ \ *N\ \ *F)\\
&\ \to (=\ \ *N\ (*L|*P)), (del\ \ *m\ \ *M\ \ *P\ \ *F).
\end{aligned}$$

del predicate indicates that elements which are smaller than the first argument's number are taken from the second argument and used to form the third argument's elements.

When the second argument contains elements that are not smaller than the first argument's number, the fourth argument's flag switches to "on". We created R_s as the union of (15), (18) in Appendix and the following rules.

$$
\begin{aligned}
&(del\ \ {*m}\ ()\ \ {*L}\ \ {*F}) \Rightarrow \{(false)\}.\\
&(del\ \ {*m}\ ({*A})\ \ {*L}\ \ {*F}), \{(number\ \ {*m})\}\\
&\ \Rightarrow \{(:=\ {*A}\ (+\ {*m}\ 1)), (=\ {*L}\ ()), (=\ {*F}\ on)\}.\\
&(del\ \ {*m}\ ({*A}\ \ {*B}|\,{*C})\ \ {*L}\ \ {*F}), \{(number\ \ {*m})\}\\
&\ \Rightarrow \{(:=\ {*A}\ (+\ {*m}\ 1)), (:=\ {*B}\ (+\ {*m}\ 1)), (=\ {*F}\ on)\},\\
&\quad (mins\ \ {*m}\ \ {*C}), (=\ {*L}\ \ {*C});\\
&\ \Rightarrow \{(:=\ {*A}\ (+\ {*m}\ 1)), (=\ {*B}\ \ {*m}), (=\ {*F}\ on)\},\\
&\quad (mins\ \ {*m}\ \ {*C}), (=\ {*L}\ ({*B}|\,{*C}));\\
&\ \Rightarrow \{(=\ {*A}\ \ {*m}), (:=\ {*B}\ (+\,{*m}\ 1)), (=\ {*F}\ on)\},\\
&\quad (mins\ \ {*m}\ \ {*C}), (=\ {*L}\ ({*A}|\,{*C}));\\
&\ \Rightarrow \{(=\ {*A}\ \ {*m}), (=\ {*B}\ \ {*m})\},\\
&\quad (=\ {*L}\ ({*A}\ \ {*B}|\,{*M})), (del\ \ {*m}\ \ {*C}\ \ {*M}\ \ {*F}).\\
&(mins\ \ {*m}\ ()) \Rightarrow .\\
&(mins\ \ {*m}\ ({*A}|\,{*B})), \{(number\ \ {*m})\}\\
&\ \Rightarrow \{(=\ {*A}\ \ {*m})\}, (mins\ \ {*m}\ \ {*B}).
\end{aligned}
$$

Then, the incorrectness of r_{10} is successfully checked by Algorithms 1 and 2. min predicate indicates that all elements in the second argument list are unified with the first argument's number.

We can confirm by the results of Sections 4.1 – 4.10 that the incorrectness of nine out of the ten test rules can be checked by Algorithms 1 and 2.

5. **Comparison to Other Debugging Algorithms.** This section compares Algorithm 1 with other debugging algorithms.

5.1. **Checking correctness or incorrectness of a rule.** Under the ET model, it is possible to check a rule r for correctness or incorrectness using the following methods.

Method 1: If the pattern of atoms in r is identical to that in one of the clauses in D, then r is an ET-rule. Therefore, check whether the pattern of atoms in r is identical to that in one of the clauses in D or not.

Method 2: For $Q_0, \ldots, Q_n \subseteq \mathbb{C}$ and $q \in \mathbb{Q}$, assume that the transformation

$$
\begin{aligned}
&Q_0 \to \cdots \to Q_n, \quad 0 \le n < \infty,\\
&Q_0 = \{(ans(q)) \leftarrow q\}, \quad Q_n \subseteq \mathbb{U}
\end{aligned}
$$

is executed within $D : q \xrightarrow{R} A$. For $i \in \{1, \ldots, n\}$, if r transforms Q_i into Q_{i+1} and $\mathcal{M}(D \cup Q_i) \neq \mathcal{M}(D \cup Q_{i+1})$, then r is incorrect. On the other hand, computations of $\mathcal{M}(D \cup Q_p)$, $p \in \{i, i+1\}$ require infinite procedures in general. Thus we compute $A_p = \mathcal{M}(D \cup Q_p) \cap \mathrm{rep}(ans(q_d))$ for $q_d \in \mathbb{Q}$ and check whether $A_i \neq A_{i+1}$, which is a sufficient condition of $\mathcal{M}(D \cup Q_i) \neq \mathcal{M}(D \cup Q_{i+1})$, holds or not. If $A_i \neq A_{i+1}$, then r is incorrect. A_p can be computed by specializing $D \cup Q_p$ using q_d and transforming the specialized set by applying rules in R_c.

Method 3: Manually prove, based on (1), that r is an ET rule.

Method 4: Create instances, use r to transform the instances, and check whether meaning is preserved or not following this transformation. If the meaning is not preserved, then r is a non-ET rule.

Methods 1 and 3 are suitable for showing that r is an ET rule. Methods 2 and 4 are suitable for showing that r is a non-ET rule. Method 1 can be executed with low cost because it only needs to check the patterns of the atoms. However, we cannot say that r is incorrect even if the pattern of atoms in r is not identical to those in all clauses in D. The reason is that there are ET rules which have patterns of atoms that are not identical to those in all clauses in D. In the execution of Method 2, we must execute the whole program and compute A_i for each i. A large cost is required for execution and computations when the given program is large. Method 3 requires the largest cost of all methods since programmers are required to manually prove (1) while the other methods can be executed automatically. Method 4 can be executed at lower cost if the instances are given as, with regards to Method 2, it requires far fewer transformations and, with regards to Method 4, it needs only finite instances while infinite instances must be considered under Method 3. Moreover, incorrectness can be checked even for the rules whose patterns of atoms are not identical to those in all clauses in D. Thus we proposed Algorithm 1 based on Method 4.

5.2. **Comparison to debugging algorithms in the LP model.** Similarly to the algorithms in [12, 13, 14], the debugging algorithms [2, 3, 4, 5, 7, 10, 16, 17, 18, 19, 20, 21] in the LP model [8, 19] are applied when an incorrect answer is obtained as the result of program execution. Then does an algorithm similar to Algorithm 1 also exist in the LP model? Namely does an algorithm for checking incorrectness of a program component also exist in the LP model? The answer seems to be NO. The reason is as follows. In the LP model, program is a set of clauses. Although correctness of a set of clauses is defined, correctness of an individual clause is not defined. Therefore, the criterion for checking incorrectness of an individual clause does not exist. In the ET model, on the other hand, correctness of an individual rule is defined which makes it possible to check incorrectness of a rule.

5.3. **Comparison to the debugging algorithms in the ET model.** In this section, we will demonstrate the essential advantage of the proposed method by providing a comparison with the debugging algorithms in [12, 13, 14]. To begin, their characteristics and problem areas are illustrated.

The characteristics and problem areas of the [12] method are as follows.
[Characteristics]
· Some of the rule sets it uses have high probability of being correct and some have low probability; incorrect rules are detected from the rule sets with low probability of being correct.
· An incorrect rule itself can be detected, not a set containing incorrect rules.
[Problem areas]
· Distinguishing between the rule sets with high probability of being correct and those with low probability is difficult.
· Since it does not say if a rule set is absolutely correct, it cannot be asserted that a detected rule is "absolutely incorrect".

The characteristics and problem areas of the [13] method are as follows.
[Characteristics]
· Incorrect rules are detected from the rule sets by inserting debugging rules created by

Oracle (correct rule generating system).
· In the program, it is not necessary to distinguish between the rule sets with high probability of being correct and those with low probability.
· An incorrect rule itself can be detected, not a set containing incorrect rules.
· As long as the rules created by Oracle are correct, it can be asserted that detected rule(s) are "absolutely incorrect."

[Problem areas]
· Oracle, which can create correct rules, is required.
· Rule creation through Oracle is more difficult than the "Yes-No answers" of LP algorithmic debugging.

The characteristics and problem areas of the [14] method are as follows.

[Characteristics]
· Partial rule sets (including incorrect rules) within the initial program can be computed.
· In the program, it is not necessary to distinguish between the rule sets with high probability of being correct and those with low probability.
· Existence of incorrect rules within the computed rule sets can be theoretically and definitely guaranteed.

[Problem areas]
· It is impossible to determine which rules within the computed rule sets are incorrect.
· The computed rule sets may correspond to the initial program. That is, results which have no meaning (which is with the initial program with incorrect rules) may be computed.

The studies which were conducted to solve the [12] method's problem areas are [13] and [14]. With the [13] method, however, it requires Oracle and therefore has associated large costs. At the same time, however, it has the highest accuracy of debugging among these three methods. With the [14] method, although existence of incorrect rules can be theoretically and definitely guaranteed, it is not capable of identifying which rules are incorrect. Therefore, each one of these three methods has both advantages and drawbacks.

The proposed method (algorithm) is an approach that is fundamentally different from these three methods. An essential difference between the proposed algorithm and the algorithms in [12, 13, 14] are as follows. In the methods in [12, 13, 14], incorrect rules are detected by exploiting program execution processes when an incorrect answer is obtained as the result of program execution. That is, even though a program may contain incorrect rules, these algorithms cannot be applied until the program is complete. In the proposed algorithm, on the other hand, the incorrectness of a rule can be checked without execution of programs. That is, this algorithm can be applied even when a program is not complete. Therefore, in cases where the cost of creating and executing a program is significant, the proposed algorithm, which does not require programs to be complete, is superior because its cost is much lower than the algorithms in [12, 13, 14] for checking the incorrectness of a rule.

6. Application of the Proposed Method.

As demonstrated from Section 4.1 to Section 4.10, one of fundamental applications of the proposed method is to check the incorrectness of rules in a program. This can be applied to a variety of programs ranging from small programs to large programs.

Although there are many other applications for the proposed method, the ones considered most important will be discussed.

It is possible to apply the proposed method to program generation. In program generation under the LE-based method [11], the correctness of well-formed formulas, referred to as logical equivalence (LE), is verified through the use of ET rule-based programs. Some of the ET rules required in the verification process must be manually created by programmers. To correctly verify LE, making sure that incorrect ET rules are not included in the manually created ET rules is crucial. However, having all ET rules checked by programmers exacts a high cost. Because the proposed method is able to eliminate incorrect ET rules at a low cost, it may make the implementation of LE verification programs more efficient.

Furthermore, the proposed method can be applied to programming learning via e-learning systems. Currently, in the most commonly used e-learning systems, such as Moodle, system developers provide e-learning system functions and learners and teachers use the functions [6, 15]. However, incorporating a function, through which the incorrectness of the program's rules is checked, to these systems is theoretically and structurally impossible. So, an e-learning system based on the theory of the equivalent transformation (ET) model is currently under development. Because this system can incorporate a function to check the incorrectness of the program's rules, it can provide a new, efficient approach to programming learning that is different from conventional e-learning systems.

7. Conclusions.

This paper proposed Algorithms 1 and 2 for checking incorrectness of a rule in ET programs and for creating a set of instances, respectively, presented Theorem 3.1 for justifying Algorithm 1, and verified the effectiveness of Algorithm 1 through some examples. The main feature of Algorithms 1 and 2 is checking incorrectness of r by creating finite $Q^{(1)}, \ldots, Q^{(m)}$ such that r can be applied to.

REFERENCES

[1] K. Akama, Y. Sigeta and E. Miyamoto, Solving logical problems by equivalent transformation (1) – A theoretical foundation, *J. Japanese Society for Artificial Intelligence*, vol.13, no.6, pp.929-935, 1998.

[2] E. Av-Ron, *Top-Down Diagnosis of Prolog Programs*, Ph.D. Thesis, Weizmanm Institute, 1984.

[3] D. Binks, *Declarative Debugging in Ӧodel*, Ph.D. Thesis, University of Bristol, 1995.

[4] R. Caballero, A declarative debugger of incorrect answers for constraint functional-logic programs, *Proc. of 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming*, pp.8-13, 2005.

[5] T. Davie and O. Chitil, Hat-delta: One right does make a wrong, *The 7th Symposium on Trends in Functional Programming*, 2006.

[6] H. Hideto, A blended learning environment using moodle, *Journal of Japan Society for Educational Technology*, vol.20, pp.241-242, 2004.

[7] V. Hirunkitti and C. J. Hogger, A generalised query minimisation for program debugging, *Lecture Notes in Comput. Sci.*, vol.749, pp.153-170, 1993.

[8] J. W. Lloyd, *Foundations of Logic Programming*, 2nd Edition, Springer-Verlag, 1987.

[9] H. Mabuchi, K. Akama and T. Wakatsuki, Equivalent transformation rules as components of programs, *International Journal of Innovative Computing, Information and Control*, vol.3, no.3, pp.685-696, 2007.

[10] M. Maeji and T. Kanamori, Top-down zooming diagnosis of logic programs, *ICOT Technical Report TR-290, ICOT*, Japan, 1987.

[11] K. Miura, K. Akama, H. Mabuchi and H. Koike, Theoretical basis for making equivalent transformation rules from logical equivalence for program synthesis, *International Journal of Innovative Computing, Information and Control*, vol.9, no.6, pp.2635-2650, 2013.

[12] S. Miyajima, K. Akama and H. Mabuchi, Algorithmic debugging of equivalent transformation programs based on differences in certainty of rules, *Proc. of the 9th International Conference on Intelligent Technologies*, pp.103-112, 2008.

[13] S. Miyajima, K. Akama and H. Mabuchi, Algorithmic debugging of equivalent transformation programs using oracle rules, *International Journal of Innovative Computing, Information and Control*, vol.7, no.8, pp.4703-4716, 2011.

[14] S. Miyajima, K. Akama, H. Mabuchi and Y. Wakamatsu, Automatic detection of incorrect rules in equivalent transformation programs, *International Journal of Innovative Computing, Information and Control*, vol.5, no.8, pp.2203-2218, 2009.
[15] Moodle.org, *https://moodle.org*.
[16] L. Naish, A declarative debugging scheme, *J. Functional and Logic Programming*, vol.1997, 1997.
[17] L. Naish, A three-valued declarative debugging scheme, *Proc. of Workshop on Logic Programming Environments*, pp.1-12, 1997.
[18] L. M. Pereira, Rational debugging in logic programming, *Lecture Notes in Comput. Sci.*, vol.225, pp.203-210, 1986.
[19] K. Pettorossi and M. Proietti, Transformation of logic programs: Foundations and techniques, *Journal of Logic Programming*, vol.19/20, pp.261-320, 1994.
[20] E. Y. Shapiro, *Algorithmic Program Debugging*, ACM Distinguished Dissertation Series, MIT Press, Cambridge, MA, 1982.
[21] J. Silva, A comparative study of algorithmic debugging strategies, *Lecture Notes in Comput. Sci.*, vol.4407, pp.143-159, 2007.
[22] *http://ext-web.edu.sgu.ac.jp/koike/eti/documents/eti_builtin/index.htm*.

Appendix.

$$\begin{aligned}
&(:=\ *X\ (+\ *A\ *B)), \{(number\ *A), (number\ *B)\}\\
&\quad \Rightarrow \{(:=\ *X\ (+\ *A\ *B))\}.\\
&(:=\ *X\ (-\ *A\ *B)), \{(number\ *A), (number\ *B)\}\\
&\quad \Rightarrow \{(:=\ *X\ (-\ *A\ *B))\}.\\
&(:=\ *X\ (\times\ *A\ *B)), \{(number\ *A), (number\ *B)\}\\
&\quad \Rightarrow \{(:=\ *X\ (\times\ *A\ *B))\}.\\
&(:=\ *X\ (d\ *A\ *B)), \{(number\ *A), (number\ *B)\}\\
&\quad \Rightarrow \{(:=\ *X\ (d\ *A\ *B))\}.\\
&(>\ *A\ *B), \{(number\ *A), (number\ *B)\}\\
&\quad \Rightarrow \{(>\ *A\ *B)\}.\\
&(<\ *A\ *B), \{(number\ *A), (number\ *B)\}\\
&\quad \Rightarrow \{(<\ *A\ *B)\}.\\
&(testMatch\ *A\ *B)\\
&\quad \Rightarrow \{(testMatch\ *A\ *B)\}.\\
&(==\ *A\ *B), \{(number\ *A), (number\ *B)\}\\
&\quad \Rightarrow \{(==\ *A\ *B)\}.\\
&(/==\ *A\ *B), \{(number\ *A), (number\ *B)\}\\
&\quad \Rightarrow \{(/==\ *A\ *B)\}.\\
&(list\ *A) \Rightarrow \{(=\ *A\ (*a))\};\ \Rightarrow \{(=\ *A\ (*a\ *b))\};\ \Rightarrow \{(=\ *A\ (*a\ *b\ *c))\}.\\
&(number\ *A) \Rightarrow \{(=\ *A\ 1)\};\ \Rightarrow \{(=\ *A\ 2)\};\ \Rightarrow \{(=\ *A\ 3)\}.\\
&(int\ *A) \Rightarrow \{(=\ *A\ -1)\};\ \Rightarrow \{(=\ *A\ 0)\};\ \Rightarrow \{(=\ *A\ 1)\}.\\
&(real\ *A) \Rightarrow \{(=\ *A\ 0.5)\};\ \Rightarrow \{(=\ *A\ 1.5)\};\ \Rightarrow \{(=\ *A\ 2.5)\}.\\
&(char\ *A) \Rightarrow \{(=\ *A\ a)\};\ \Rightarrow \{(=\ *A\ b)\};\ \Rightarrow \{(=\ *A\ c)\}.\\
&(string\ *A) \Rightarrow \{(=\ *A\ abc)\};\ \Rightarrow \{(=\ *A\ def)\};\ \Rightarrow \{(=\ *A\ ghi)\}.
\end{aligned} \tag{17}$$

"$-$" means subtraction. "$>$" and "$<$" are signs of inequalities. $(testMatch\ *A\ *B)$ mean that $*B$ can be matched with $*A$. "$==$" and "$/==$" denote equality and non-equality, respectively. $(list\ *A)$, $(number\ *A)$, $(int\ *A)$, $(real\ *A)$, $(char\ *A)$ and $(string\ *A)$

mean that $*A$ is a list, a number, an integer, a real number, a character and a string, respectively.

$$
\begin{aligned}
&(rdc\ *X\ ()\ *L\ *F) \Rightarrow \{(false)\}.\\
&(rdc\ *X\ (*A)\ *L\ *F), \{(=\ *X\ a)\}\\
&\ \Rightarrow \{(=\ *A\ b), (=\ *L\ ()), (=\ *F\ on)\}.\\
&(rdc\ *X\ (*c\ *d|*C)\ *L\ *F), \{(=\ *X\ a)\}\\
&\ \Rightarrow \{(=\ *c\ b), (=\ *d\ b), (=\ *F\ on)\},\\
&\quad (noRdc\ *X\ *C\ *L\ *F);\\
&\ \Rightarrow \{(=\ *c\ b), (=\ *d\ a), (=\ *F\ on)\},\\
&\quad (=\ *L\ (*d|*M)), (noRdc\ *X\ *C\ *M\ *F);\\
&\ \Rightarrow \{(=\ *c\ a), (=\ *d\ b), (=\ *F\ on)\},\\
&\quad (=\ *L\ (*c|*M)), (noRdc\ *X\ *C\ *M\ *F);\\
&\ \Rightarrow \{(=\ *c\ a), (=\ *d\ a)\},\\
&\quad (=\ *L\ (*c\ *d|*M)), (rdc\ *X\ *C\ *M\ *F);\\
&\ \Rightarrow \{(=\ *c\ b), (=\ *F\ on)\},\\
&\quad (=\ *L\ (*d|*M)), (noRdc\ *X\ *C\ *M\ *F);\\
&\ \Rightarrow \{(=\ *d\ b), (=\ *F\ on)\},\\
&\quad (=\ *L\ (*c|*M)), (noRdc\ *X\ *C\ *M\ *F);\\
&\ \Rightarrow \{(=\ *c\ a)\}, (=\ *L\ (*c\ *d|*M)),\\
&\quad (rdc\ *X\ *C\ *M\ *F);\\
&\ \Rightarrow \{(=\ *d\ a)\}, (=\ *L\ (*c\ *d|*M)),\\
&\quad (rdc\ *X\ *C\ *M\ *F);\\
&\ \Rightarrow (=\ *L\ (*c\ *d|*M)), (rdc\ *X\ *C\ *M\ *F).\\
&(noRdc\ *X\ *L\ *M\ *F) \Rightarrow \{(=\ *M\ *L)\}.\\
&(=\ *L\ *M) \Rightarrow \{(=\ *L\ *M)\}.\\
&(==\ *A\ *B) \Rightarrow \{(==\ *A\ *B)\}.
\end{aligned}
\tag{18}
$$