Generating Speq Rules based on Automatic Proof of Logical Equivalence

Katsunori Miura, Kiyoshi Akama, and Hiroshi Mabuchi

Abstract—In the Equivalent Transformation (ET) computation model, a program is constructed by the successive accumulation of ET rules. A method by meta-computation by which a correct ET rule is generated has been proposed. Although the method covers a broad range in the generation of ET rules, all important ET rules are not necessarily generated. Generation of more ET rules can be achieved by supplementing generation methods which are specialized for important ET rules. A Specialization-by-Equation (Speq) rule is one of those important rules. A Speq rule describes a procedure in which two variables included in an atom conjunction are equalized due to predicate constraints. In this paper, we propose an algorithm that systematically and recursively generate Speq rules and discuss its effectiveness in the synthesis of ET programs. A Speq rule is generated based on proof of a logical formula consisting of given atom set and dis-equality. The proof is carried out by utilizing some ET rules and the ultimately obtained rules in generating Speq rules.

Keywords—Equivalent transformation, ET rule, Equation of two variables, Rule generation, Specialization-by-Equation rule

I. INTRODUCTION

GENERATING efficient programs which are correct with respect to their specifications is very important. Many studies generate a low-level language program from a specification (Refer to Fig. 1). In their studies, a specification is strictly defined by a formal description approach to efficiently generate a program. An algebraic specification [4] is one of formal description approaches, where a specification is defined by an equality. A method for generating C or Java programs from algebraic specifications is proposed in [7], [11]. A method [3] for generating C++ programs from specifications defined by UML is also proposed.

A theory of program generation based on the Equivalent Transformation (ET) computation model [1] has also been proposed by our research group, where a specification is defined by a set of definite clauses, and a low-level language program is generated from their specification (Refer to Fig. 1). A crucial difference between our approach and others is that we have a space of ET rules for searching algorithms that is correct with respect to the given specification, and generate a program from a set of ET rules that is found by the search. A framework for generating C programs from ET rule sets has been proposed in [18].

An ET rule is a rewriting rule that describes a procedure to replace a clause set with another clause set preserving the equivalent declarative meaning. ET rules have abundant expressive powers that can describe various procedures, and each ET rule is completely independent of other ET rules. An algorithm is constructed by successively accumulating ET rules, and it is guaranteed that an algorithm constructed by correct ET rules is correct with respect to their specifications. ET rules have such good features for constructing algorithms. Generation of ET rules is the most important task in making programs.

A general method for generating correct ET rules has been proposed in [9]. In this method, an ET rule is generated by a finite synthesis of basic equivalent transformations using meta-computation. Since a method using meta-computation covers a broad range in the generation of ET rules, a system [14] based on this method can generate various ET rules. However, in the method using meta-computation, 1. it is not necessarily the case that all ET rules are generated, and 2. there are ET rules which cannot be generated in a short period of time.

In this paper, we consider that ET rule generation focusing on the following items is important.

1) Even if a generation range of ET rules is narrowed, ET rules derived in the range are important.
2) ET rules cannot be generated by a finite synthesis of basic equivalent transformations.
3) An algorithm which efficiently generate ET rules can be found.

One of the rules that satisfy the items mentioned above is a Specialization-by-Equation (Speq) rule. A Speq rule describes a procedure in which two variables included in an atom conjunction are equalized due to predicate constraints. Generating a Speq rule is essential because the rule solves problems...
more efficiently and is absolutely necessary to solve a certain problem within a finite period of time. When a conjunction of atoms is given as an input, the algorithm proposed in this paper outputs all Speq rules obtained from the input. An ET rule set is called a program in the rest of this paper.

This paper describes the syntax of ET rules, specifications, programs and computation on the ET computation model in Section 2 and discusses the importance of Speq rules in Section 3. It also proposes an algorithm that generates a Speq rule in Section 4 and presents the process of generating the rules by the proposed algorithm in Section 5. Furthermore, the effectiveness of the algorithm is discussed in Section 6.

The following notations will be used. Taking $A$ as an atom, let $\text{Var}(A)$ be the set of variables on $A$. Since an atom conjunction is treated as an atom set in this paper, it is called an atom set in the rest of this paper.

II. PROGRAM CONSTRUCTION IN THE ET COMPUTATION MODEL

A. Definition of ET rules

In the ET computation model, a procedure is described by a set of declarative meaning-preserving rewriting rules, each of which is called an ET rule. An ET rule replaces a clause set $C$ with another clause set $C'$ which has an equivalent declarative meaning as $C$.

[Syntax of ET rules]

An ET rule is composed of a head part ($Hs$), a condition part ($C$), an execution part ($E_n$) and a replacement part ($B_{sn}$); where $n$ is a nonnegative integer.

\[ Hs, [C] \Rightarrow [E_1], B_{s1}; \]
\[ \Rightarrow [E_2], B_{s2}; \]
\[ \vdots \]
\[ \Rightarrow [E_n], B_{sn}. \]

A head part consists of one or more atoms while all other parts consist of zero or more atoms. When a head is transformed into two or more bodies as shown above, ";" is used for dividing those bodies. If a head is transformed to only one body, such an ET rule is described as ($Hs$, $C$) $\Rightarrow$ $[E]$, $B_{sn}$). The proposed algorithm in this paper generates ET rules in which the number of bodies is one. However, ET rules with multiple bodies are used when generating ET rules by the proposed algorithm (See Section V).

B. Program and specification in the ET computation model

In the theory of program construction based on the ET computation model, a program is a set of ET rules. The computation of the program is represented as a finite or infinite sequence $\text{com} = [s_0, s_1, s_2, \ldots]$. For any two successive problems $s_i$ and $s_{i+1}$ in $\text{com}$, the program transforms $s_i$ into $s_{i+1}$ by one step. The transformation is carried out as long as there exists an ET rule that can be applied to a problem in $\text{com}$.

In the ET computation model, a specification is strictly defined by a set of definite clauses. Such a method where a set of definite clauses is taken as a specification is an essential paradigm used in Logic Programming (LP) [10], [15] and Constraint Logic Programming (CLP) [8], [17]. Strictly, a specification is a pair $\langle D, Q \rangle$, where $D$ is background knowledge (predicate definition) and $Q$ is a set of queries. Since ET rules have partial correctness with respect to $D$, it is guaranteed that a program consisting of only correct ET rules has partial correctness with respect to its specification.

C. Importance of ET rules

Since ET rules have complete independence with regard to partial correctness, adding new ET rules to an existing ET rule set does not impair the correctness of the existing ET rule set. This enables the function of a program to be flexibly expanded through modification of each ET rule or addition of new ET rules. In addition, the squeeze method [2] where a program is created by successively accumulating ET rules has been proposed.

Furthermore, since software systems become larger and more complex, the need for the development of cost-effective and reliable software in a short period of time increases accordingly. As a result, there has been increasing attention to component-wise program generation [5], [6], [16]. This programming method generates a program by accumulating program components one by one, each of which has passable independence. Since ET rules can be considered as components constituting a program, also when viewed from the perspective of component-wise program generation, they can confer considerable advantage [12].

III. APPROACHES FOR GENERATING ET RULES

A. Generating ET rules based on the meta-computation method

A general method for generating ET rules has been proposed in [9]. The method replaces an atom set with another atom set using a basic equivalent transformation and generates ET rules by a finite synthesis of replacements. This replacement is carried out using meta-computation, which replaces an atom set with another atom set using general ET rules (unfolding rules and build-in rules) that are obtained from the definition of predicates. Since the method using meta-computation can generate various ET rules, a system for constructing programs based on this method has achieved many successful results [14].

In the meta-computation method, paths for replacing an atom set are important. There exist many paths because of the following two items (Refer to Fig. 2).

1) There exist various ET rules applied to an atom set.
2) Number of rule applications

An ET rule is generated based on a certain evaluation from among the derived paths. Because there is a possibility that a search area may increase tremendously, the meta-computation method uses the following controls.

1) Limit the number of rule applications
2) Specify an application rule

In the meta-computation method, if the number of unfolding rule applications increases, then a search area grows correspondingly, increasing the cost of ET rule generation.
B. Proposing a new method focusing on specialized rule classes

In this paper, we consider that ET rule generation focusing on the following items is important.
1) Even if a search area of ET rules is narrowed, ET rules derived in the area are important (Refer to Fig. 2).
2) ET rules cannot be generated by a finite synthesis of basic equivalent transformations.
3) An algorithm which efficiently generate ET rules can be found.

We are not dismissing the meta-computation method but, instead, taking an approach where several generation methods are combined as shown in Fig. 3. In this paper, we propose an algorithm that generates ET rules with respect to a meshed area of Fig. 3. (The area includes ET rules which cannot be generated by meta-computation method.)

C. The target ET rules

A Specialization-by-Equation (Speq) rule is one of ET rules, each of which describes a procedure in which two variables included in an atom set are equalized due to predicate constraints. For example, we will now consider a Speq rule for an atom set \{copy(Vx, Vy), copy(Vx, Vv)\}. A predicate copy is defined by the following two clauses.

\[ c1 : \text{copy}([ ] [ ] \leftarrow c2 : \text{copy}([A] B), [A] [C] \leftarrow \text{copy}(B, C) \]

Since variables Vy and Vv are copies of a variable Vx, it is not difficult to deduce that Vy = Vv. This equation is defined by a Speq rule, and the following formula is a Speq rule with respect to \{copy(Vx, Vy), copy(Vx, Vv)\}. eq(\text{Arg}1, \text{Arg}2) means that \text{Arg}1 is equal to \text{Arg}2.

\[ r_{\text{copy}} : \text{copy}(Vx, Vy), \text{copy}(Vx, Vv) \Rightarrow eq(Vy, Vv), \text{copy}(Vx, Vv), \text{copy}(Vx, Vv). \]

[Importance of Speq rules]

A Speq rule has the following features:
1) an ET rule with single body, i.e., a Speq rule replaces a head with one body.
2) an ET rule where an eq atom is included in the body part. In the ET program, primarily applying the following makes the computation efficient: 1) ET rules with single body and 2) ET rules which are useful for specializing clauses. A eq atom in the body part of a Speq rule promotes the transformation of clauses by specializing a clause efficiently. Many Speq rules cannot be generated by meta-computation method; therefore, an algorithm for generating Speq rules is essential.

[Example of computation using Speq rule]

Speq rules promote the transformation of definite clauses. Therefore, by using a Speq rule, a set of unit clauses can be obtained from a query Q with fewer transformations than without the use of the Speq rule. We now show an example of replacing a clause using \( r_{\text{copy}} \) mentioned above. Let a definite clause set \( \text{ans}(Vc, Vd) \leftarrow \text{copy}(Va, Vb), Vc), \text{copy}(Va, Vb), Vd)) \) be a query Q. When \( r_{\text{copy}} \) is applied to Q, Q is replaced as follows:

\[ \text{ans}(Vc, Vd) \leftarrow eq(Vc, Vd), \text{copy}(Va, Vb), Vc), \text{copy}(Va, Vb), Vd) \]

When the eq atom is executed, the above clause set is replaced to \( \text{ans}(Vc, Vc) \leftarrow \text{copy}(Va, Vb), Vc), \text{copy}(Va, Vb), Vc) \). Since two atoms in the body part of the above clause are the same, the above clause is replaced to \( \text{ans}(Vc, Vc) \leftarrow \text{copy}(Va, Vb), Vc) \). Finally, when a unfolding rule is applied to \( \text{copy}(Va, Vb), Vc), \text{copy}(Va, Vb), Vc) \) is obtained from \( \text{ans}(Vc, Vc) \leftarrow \text{copy}(Va, Vb), Vc) \).

When the above computation is carried out without using Speq rules, unfolding rules are generally used. Because unfolding rules are applied separately to \( \text{copy}(Va, Vb), Vc) \) and \( \text{copy}(Va, Vb), Vd) \), they, as a result, require more transformations than Speq rules use.

IV. Algorithm for generating Speq rules

A. Syntax of Speq rules

A Speq rule describes a procedure in which two variables included in an atom set are equalized due to predicate constraints (Refer to Section III-C). In this section, we show the
syntax of Spec rules. Let A be a set of atoms. Let a ∈ A. Then, arguments of A take a variable or a constant. Let X ∈ Var(A) and Y ∈ Var(A), where X ≠ Y. A Spec rule is composed of an atom set A and an atom eq(X, Y). The syntax of Spec rules is defined by the following formula.

\[ A \Rightarrow eq(X, Y), A. \]

\[ \text{app}(Vx, [\ ], Vy) \] is one of examples including variables and a constant in an atom a. A Spec rule for \[ \text{app}(Vx, [\ ], Vy) \] has a procedure Vx = Vy as follows:

\[ r_{\text{app}} : \text{app}(Vx, [\ ], Vy) \Rightarrow eq(Vx, Vy), \text{app}(Vx, [\ ], Vy). \]

B. Outline of the proposed algorithm and its input and output

1) Outline of the algorithm: The proposed algorithm generates ET rules consisting of a head part and a replacement part. Let A be an atom set. Let X and Y be variables on Var(A), where X ≠ Y. The head part consists of an atom set A and the replacement part an atom set A \( \cup eq(X, Y) \). That an ET rule contains eq(X, Y) in its body part is important in embodying variables. If a logical formula \( \forall(A \Rightarrow eq(X, Y)) \) is true, the following logical equivalence between atom sets is guaranteed.

\[ \forall(A \leftrightarrow eq(X, Y), A) \]

Therefore, the following rewriting rule derived by the proposed algorithm is an ET rule [13].

\[ A \Rightarrow eq(X, Y), A. \]

Given A, the proposed algorithm proves X = Y, and consequently an ET rule is generated. The algorithm then derives the empty set by successively transforming a clause set \( \{yes \Leftarrow A. \text{noteq}(X, Y)\} \) using a rule set.

2) Input and output: We show the definition of an input and an output of the proposed algorithm. An input is A, and an output is S, which is a set of all Spec rules obtained from the input. A Spec rule is composed of A and eq(X, Y) and is defined as follows:

\[ S = \{(A \Rightarrow eq(X, Y), A) | (X \in \text{Var}(A)) \text{ & } (Y \in \text{Var}(A)) \text{ & } (Y \neq X) \text{ & } (\forall(A \Rightarrow eq(X, Y))\} \]

If all conditions for S are not satisfied, then S is the empty set.

C. The proposed algorithm

When A is given, the proposed algorithm outputs S obtained by from A (Refer to Fig. 4). The algorithm is composed of the following eight processes.

[Algorithm for generating Spec rules]

1) Derive A as an input (See Section IV-B.2).
2) Select variables X and Y, where X ∈ Var(A), Y ∈ Var(A), and X ≠ Y.
3) Create a clause set cls as an initial state. cls is composed of A and noteq(X, Y).

\[ \text{noteq}(\text{Arg}1, \text{Arg}2) \] means that Arg1 is not equal to Arg2.

\[ \text{cls} = \{\text{yes} \Leftarrow A. \text{noteq}(X, Y)\} \]

4) Replace cls using a rule r in a rule set R_r.

\[ \text{R}_r \] consists of some ET rules and one Inductively-used rule (Refer to Section IV-D). Typical ET rules used by the proposed algorithm are general unfolding rules and specialization rules. Only ET rules can be applied to cls while Inductively-used rules can be applied to atoms obtained through replacement by unfolding rules.

\[ \text{trans}(\text{cls}, r \in \text{R}_r) \] used in Fig. 4 is defined as follows:

\[ \text{trans}(\text{cls}, r \in \text{R}_r) = \text{A} \cup \text{B} \]

\[ \text{A} = \{cl_a | (cl \in \text{cls}) \text{ & } (cl \text{ is applicable to } r) \text{ & } (cl_a \text{ is obtained from } cl \text{ applied to } r)\} \]

\[ \text{B} = \{cl_b | (cl_b \in \text{cls}) \text{ & } (cl_b \text{ is not applicable to } r)\} \]

5) Add a Spec rule to a Spec rule set S. If cls is an empty set, then add rule to S and go to 7, else go to 6.

6) Select a rule r ∈ R_r.

If there exist rules applicable to a body atom set of a clause cl ∈ cls, then select one of them and go to 4. Otherwise, generate a new ET rule r, and then add it to R_r, select r and go to 4.

7) Select other variables X and Y. If X and Y are selected, then go to 3, else go to 8.

8) Output S.

D. Rule set R_r used in generation process

\[ \text{R}_r \] consists of some ET rules and one Inductively-used rule. While ET rules can be applied to any atoms, Inductively-used rules can be applied to only atoms that satisfy a certain condition. The atoms to which Inductively-used rules can be applied are a set of atoms obtained following the application of general unfolding rules. The control of rule application in the proposed algorithm is shown below.

1) Inductively-used rules should primarily be applied to a set of atoms to which general unfolding rules were applied previously.

2) For the other atoms, ET rules are applied.

1) ET rules: Typical ET rules used by the proposed algorithm are general unfolding rules and specialization rules. A general unfolding rule can be created by simple transformation of a clause which is the definition of a predicate. For example, we will show a general unfolding rule with respect to \text{app} predicate. \text{app} predicate is defined by the following two clauses.

\[ \text{app}([\ ], A, A) \Leftarrow \]

\[ \text{app}([A | B], C, [A | D]) \Leftarrow \text{app}(B, C, D) \]

The following general unfolding rule is created based on the definition of \text{app} predicate.

\[ \text{app}(Vx, Vy, Vz) \Rightarrow \{Vx = [\ ], Vy = Vz\}; \]

\[ = \{Vx = [Vp | Vq], Vz = [Vp | Vw], \text{app}(Vq, Vy, Vw)\}. \]

This rule replaces a clause to which this rule can be applied with two clauses, one of which executes the computation of Vx = [ ] and Vy = Vz, and the other replaces \text{app}(Vx, Vy, Vz) with \text{app}(Vq, Vy, Vw) after executing Vx = [Vp | Vq] and Vz = [Vp | Vw].
A specialization rule describes a unification of arguments or the definition of a false condition. For example, basic ET rules on `eq` which describes a unification of arguments are as follows:

\[
eq(X, Y) \Rightarrow \{(Vx, [\_]), \}\.
\]

\[
eq([Vx | Vy], [Vz | Vw]) \Rightarrow \{eq(Vx, Vz), eq(Vy, Vw)\}.
\]

\[
\text{noteq}(Vx, Vz) \Rightarrow \{false\}.
\]

\[
\text{noteq}([Vx | Vy], [Vx | Vz]) \Rightarrow \{\text{noteq}(Vx, Vz)\}.
\]

`false` is a special built-in atom with the meaning that “a head atom of a rule does not satisfy a predicate constraint”.

2) **Inductively-used rules**: An inductively-used rule is created based on `Speq` rules generated by the proposed algorithm. By using the proposed algorithm, the following `Speq` rule is created.

\[
A \Rightarrow eq(X, Y), A.
\]

Since only the logical equivalence of this rule is guaranteed, using it as it is at this stage may cause an infinite loop. To avoid such a risk, a transformation of the rule based on the following terms is employed.

1) When a variable `X` is not equal to a variable `Y`, this ET rule is applicable to an atom set `(not(X == Y))`.

2) Move `eq` atom to the execution part, and define `eq` atom using a built-in description (`X = Y`).

A rule derived by such a transformation is Inductively-used rule, which is defined as follows.

\[
A \Rightarrow \{\text{not}(X == Y)\} \Rightarrow \{X == Y\}, A.
\]

Let `app(Vx, [\_], Vy)` be an input atom set. Let `X(e \in Var(app(Vx, [\_],Vy)))` be a variable `X` and `Y(e \in Var(app(Vx, [\_],Vy)))` a variable `Y`. Then, the following Inductively-used rule is created.

\[
\begin{align*}
app(Vx, [\_], Vy) \Rightarrow & \{\text{not}(Vx == Vy)\} \Rightarrow \{Vx == Vy\}, app(Vx, [\_], Vy).
\end{align*}
\]

### V. Examples for generating `Speq` rules by the proposed algorithm

A. **Generation of `Speq` rule on \{app(Vx, [\_], Vy)\**

Based on the algorithm shown in Fig. 4, let us generate `Speq` rules for an atom set `app(Vx, [\_], Vy)`. An input is \{app(Vx, [\_], Vy)\}. `app(Arg1, Arg2, Arg3)` means that the concatenation of a list `Arg1` and a list `Arg2` is a list `Arg3`. Since the second argument of `app(Vx, [\_], Vy)` is an empty list, it is not difficult to deduce that `Vx` is equivalent to `Vy`. In this example, we will present the process in which a set `S`, taking the following `Speq` rule `r` as its element, is output.

\[
r_{app} : \{app(Vx, [\_], Vy) \Rightarrow \{eq(Vx, Vy), app(Vx, [\_], Vy)\}.
\]

**[Selection of `X` and `Y` and creation of initial clause set `cls`]**

First, select `X(\in Var(app(Vx, [\_], Vy)))` and `Y(\in Var(app(Vx, [\_], Vy)))`, and then the initial clause set `cls` using `A` and `noteq(X, Y)`. Let `X` and `Y` be `X = Vx` and `Y = Vy`, respectively. `cls` is created based on `app(Vx, [\_], Vy)` and `noteq(Vx, Vy)`.

\[
\text{cls} = \{\text{yes} \Leftarrow \text{app}(Vx, [\_], Vy), \text{noteq}(Vx, Vy)\}
\]

If an empty set is obtained by replacing `cls` using a rule set `R`, then `\{\text{app}(Vx, [\_], Vy) \Rightarrow eq(Vx, Vy)\}` can be proven.

**[Initial rule set `R`]**

`R` consists of four rules, which are `r1`, `r2` and `r3` representing ET rules and `r4` representing an Inductively-used rule, shown in Table II.

The meaning of each rule is as follows. `r1` replaces a clause with two clauses. These are a clause that executes `Vx = [\_]` and `Vy = Vz` and one that replaces `app(Vx, Vy, Vz)` with `app(Vq, Vy,Vw)` by executing `Vx = [Vp | Vq]` and `Vz = [Vp | Vw]`. `r2` means that `noteq(Vx, Vy)` does not satisfy the constraints of `noteq` predicate, `r3` means that `noteq([Vx | Vy], [Vx | Vz])` is replaced with `not(Vy, Vz)`, and `r4` means that if
Definite clauses representation

\[ r_1 : \text{app}(Vx, Vy, Vz) \Rightarrow \{ Vx = [], Vy = Vz \}; \]
\[ \Rightarrow \{ Vx = [Vp | Vq], Vz = [Vp | Vw] \}, \text{app}(Vy, Vx, Vw). \]
\[ r_2 : \text{noteq}(Vy, Vx) \Rightarrow \{ \text{false} \}. \]
\[ r_3 : \text{noteq}([Vx | Vy], [Vx | Vz]) \Rightarrow \text{noteq}(Vy, Vz). \]
\[ r_4 : \text{app}(Vx, [v], Vy) \| \text{noteq}(Vx = = Vy) \Rightarrow \{ Vx = Vy \}, \text{app}(Vx, [v], Vy). \]

\[ Vx \text{ is not equal to } Vy, \text{ then } \text{app}(Vx, [v], Vy) \text{ is replaced with } \text{app}(Vx, [v],Vy) \text{ by executing } Vx = Vy. \]

**[Presentation of Spec rule r app generation process]**

Table I shows the transformation process in which a set of definite clauses is successively transformed through an application of qualified rules in \( R_0 \). The underlined atoms in each step are the ones to which a rule has been applied. An initial clause set \( s_0 \) is finally transformed to an empty set \( s_2 \). An empty set signifies that no answer which satisfies the initial clause set exists. Let us see the transformation process of a definite clause set. By applying an ET rule, an initial clause set \( s_2 \) is replaced with a singleton \( s_2 \) with one definite clause.

\[ s_2 = \{ \text{yes} \leftarrow \text{noteq}([Va | Vb], [Va | Vc]), \text{app}(Vb, [v], Vc) \} \]

An Inductively-used rule \( r_4 \) is applied to \( \text{app}(Vb, [v], Vc) \) in \( s_2 \). Since a definite clause set \( s_3 \) is obtained as a result, an ET rule \( r_2 \) is applied to \( \text{noteq}([Va | Vb], [Va | Vb]) \) and consequently an empty set is obtained. An empty set is obtained from \( s_3 \), therefore, \( \forall (\text{app}(Vx, [], Vy) \Rightarrow \text{eq}(Vx, Vy)) \) can be proven. Therefore, the following Spec rule is generated.

\[ \text{app}(Vx, [], Vy) \Rightarrow \text{eq}(Vx, Vy), \text{app}(Vx, [], Vy). \]

Because differing variables substituted for \( X \) and \( Y \) other than \( X = Vx \) and \( Y = Vy \), respectively, do not exist, the following set \( S \) is obtained as an output, where \( X \in \text{Var}(\text{app}(Vx, [v], Vy)) \) and \( Y \in \text{Var}(\text{app}(Vx, [v], Vy)). \)

\[ S = \{ \text{app}(Vx, [v], Vy) \Rightarrow \text{eq}(Vx, Vy), \text{app}(Vx, [v], Vy) \}. \]

**B. Generation of Spec rule on \( \text{rev}(Vx, Vy), \text{rev}(Vx, Vz) \)**

In Section V-A, \( S \) was obtained without requiring generation of new ET rules. In this section, we will present an example where generation of new ET rules using the proposed method or meta-computation method (“select ET rule generation-method” in Fig. 4) is needed. An input is an atom set \( \text{rev}(Vx, Vy), \text{rev}(Vy, Vz) \). \( \text{rev}(Arg1, Arg2) \) means that a list \( Arg2 \) is elements of a list \( Arg1 \) in reverse order. It is not difficult, therefore, to deduce that a variable \( Vx \) is equivalent to a variable \( Vz \). In this example, we will present the process in which a set \( S \), taking the following Spec rule \( r_{rev} \) as its element, is output.

\[ r_{rev} : \text{rev}(Vx, Vy), \text{rev}(Vy, Vz) \]
\[ \Rightarrow \text{eq}(Vx, Vy), \text{rev}(Vx, Vy), \text{rev}(Vy, Vz). \]

**[Selection of X and Y, Creation of initial clause set cls]**

Let \( X = Vx \) and \( Y = Vz \), where \( X \in \text{(Var}(\text{rev}(Vx, Vy)) \cup \text{Var}(\text{rev}(Vy, Vz))) \) and \( Y \in \text{(Var}(\text{rev}(Vx, Vy)) \cup \text{Var}(\text{rev}(Vy, Vz))) \). Let \( cls \) be the following formula. \( cls \) is composed of \( \text{rev}(Vx, Vy), \text{rev}(Vx, Vz) \) and \( \text{noteq}(Vx, Vz) \).

\[ cls = \{ \text{yes} \leftarrow \text{rev}(Vx, Vy), \text{rev}(Vy, Vz), \text{noteq}(Vx, Vz) \} \]

If an empty set is obtained by replacing \( cls \) using \( R_0 \), then \( \forall (\text{rev}(Vx, Vy), \text{rev}(Vy, Vz) \Rightarrow \text{eq}(Vx, Vy)) \) is proven.

**[Initial rule set \( R_0 \)]**

\( R_0 \) consists of five rules, which are \( r_1, r_2, r_3 \) and \( r_4 \) representing ET rules and \( r_5 \) representing an Inductively-used rule, shown in Table IV.

**TABLE II**

<table>
<thead>
<tr>
<th>Name</th>
<th>Definite clauses representation</th>
<th>Rule applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0 )</td>
<td>{ \text{yes} \leftarrow \text{app}(Vx, [v], Vb), \text{noteq}(Vx, Vb) }</td>
<td>( r_1 )</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>{ \text{yes} \leftarrow \text{noteq}(Vx, Vb) }</td>
<td>( r_2 )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>{ \text{yes} \leftarrow \text{noteq}(Vx, Vb, Vx), \text{app}(Vb, [v], Vc) }</td>
<td>( r_4 )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>{ \text{yes} \leftarrow \text{noteq}(Vx, Vb, Vx), \text{app}(Vb, [v], Vc) }</td>
<td>( r_2 )</td>
</tr>
</tbody>
</table>

**TABLE I**

Transformation sequence of definite clauses for the rule \( \{ \text{app}(Vx, [v], Vy) \} \)

**TABLE III**

<table>
<thead>
<tr>
<th>Name</th>
<th>Definite clauses representation</th>
<th>Rule applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_0 ) for generating Spec rule for ( { \text{app}(Vx, [v], Vy) } )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE IV**

<table>
<thead>
<tr>
<th>Name</th>
<th>Definite clauses representation</th>
<th>Rule applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_0 ) for generating Spec rule for ( { \text{rev}(Vx, Vy), \text{rev}(Vy, Vz) } )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The meaning of each rule is as follows. \( r_1 \) replaces a clause with two clauses. These are a clause that executes \( Vx = [v] \) and \( Vy = [v] \) and a clause that replaces \( \text{rev}(Vx, Vy) \) with \( \text{rev}(Vx, Vw) \) and \( \text{app}(Vw, [v], Vy) \) by executing \( Vx = [v] \). The meaning of \( r_2, r_3 \) and \( r_4 \) is the same as that of respective ET rules described in Section V-A. \( r_5 \) means that if \( Vx \) is not equal to \( Vz \), then a set of \( \text{rev}(Vx, Vy) \) and \( \text{rev}(Vy, Vz) \) is replaced with a set of \( \text{rev}(Vx, Vy) \) and \( \text{rev}(Vy, Vz) \) by executing \( Vx = Vz \).
### TABLE III

<table>
<thead>
<tr>
<th>Name</th>
<th>Definite clauses representation</th>
<th>Rule applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_0)</td>
<td>({\text{yes} \rightarrow \text{rev}(V_a, V_b), \text{rev}(V_b, V_c), \text{noteq}(V_a, V_c)})</td>
<td>(r_1)</td>
</tr>
<tr>
<td>(s_1)</td>
<td>({\text{yes} \rightarrow \text{rev}(V_a, V_b), \text{noteq}(V_a, V_c)})</td>
<td>(r_1)</td>
</tr>
<tr>
<td>(s_2)</td>
<td>({\text{yes} \rightarrow \text{rev}(V_a, V_b), \text{noteq}(V_a, V_c)})</td>
<td>(r_4)</td>
</tr>
<tr>
<td>(s_3)</td>
<td>({\text{yes} \rightarrow \text{rev}(V_a, V_b), \text{rev}(V_b, V_c), \text{app}(V_a, V_c)})</td>
<td>(r_1)</td>
</tr>
<tr>
<td>(s_4)</td>
<td>({\text{yes} \rightarrow \text{rev}(V_a, V_b), \text{app}(V_a, V_c)})</td>
<td>(r_1)</td>
</tr>
<tr>
<td>(s_5)</td>
<td>({\text{yes} \rightarrow \text{rev}(V_a, V_b), \text{rev}(V_b, V_c), \text{app}(V_a, V_c)})</td>
<td>(r_4)</td>
</tr>
<tr>
<td>(s_6)</td>
<td>({\text{yes} \rightarrow \text{rev}(V_a, V_b), \text{rev}(V_b, V_c), \text{app}(V_a, V_c)})</td>
<td>(r_5)</td>
</tr>
<tr>
<td>(s_7)</td>
<td>({\text{yes} \rightarrow \text{rev}(V_a, V_b), \text{rev}(V_b, V_c), \text{app}(V_a, V_c)})</td>
<td>(r_6)</td>
</tr>
<tr>
<td>(s_8)</td>
<td>({\text{yes} \rightarrow \text{rev}(V_a, V_b), \text{rev}(V_b, V_c), \text{app}(V_a, V_c)})</td>
<td>(r_7)</td>
</tr>
<tr>
<td>(s_9)</td>
<td>({\text{yes} \rightarrow \text{rev}(V_a, V_b), \text{rev}(V_b, V_c), \text{app}(V_a, V_c)})</td>
<td>(r_8)</td>
</tr>
<tr>
<td>(s_{10})</td>
<td>({\text{yes} \rightarrow \text{rev}(V_a, V_b), \text{rev}(V_b, V_c), \text{app}(V_a, V_c)})</td>
<td>(r_9)</td>
</tr>
<tr>
<td>(s_{11})</td>
<td>({\text{yes} \rightarrow \text{rev}(V_a, V_b), \text{rev}(V_b, V_c), \text{app}(V_a, V_c)})</td>
<td>(-)</td>
</tr>
</tbody>
</table>

#### Presentation of Specq rule \(r_0\) generation process

Table III shows the transformation process in which a set of definite clauses is successively transformed through an application of qualified rules in \(\mathbb{R}\). Let us see the transformation process. By repeatedly applying ET rules one by one, \(\text{cls}\) is replaced by a singleton \(s_0\) with two definite clauses.

\[s_0 = \{\text{yes} \rightarrow \text{noteq}(V_a, V_b), \text{rev}(V_b, V_c), \text{app}(V_a, V_c)\}\]

Next, the following ET rule which can be applied to \(\text{rev}(V_b, [\ ]\}\) is generated by meta-computation method.

\[r_0 : \text{rev}(V_x, \ [\ ] \Rightarrow (V_x = \ [\ ]))\]

Since \(r_0\) is added to a rule set \(\mathbb{R}\), \(r_6\) can be applied to \(\text{rev}(V_b, [\ ]\}\) in \(s_0\). Consequently, a definite clause set \(s_7\) is obtained. By repeatedly applying ET rules one by one, \(s_7\) is replaced by a singleton \(s_{13}\) with one definite clause.

\[s_{13} = \{\text{yes} \rightarrow \text{noteq}(V_a, V_b), \text{rev}(V_b, V_c), \text{app}(V_a, V_c)\}\]

The following ET rule which can be applied to the set of \(\text{rev}(V_a, [\ ]\}\) and \(\text{app}(V_c, V_a)\) is generated using the third Logical Equivalence (LE)-based method (Refer to Fig. 4).

\[r_7 : \text{app}(V_a, V_c, V_a) \Rightarrow \text{rev}(V_x, V_p), \text{app}(V_y, V_p, V_w)\]

Since \(r_7\) is added to \(\mathbb{R}\), \(r_7\) can be applied to the set of \(\text{rev}(V_a, [\ ]\}\) and \(\text{app}(V_c, V_a)\) in a definite clause set \(s_{13}\). Consequently, a definite clause set \(s_{14}\) is obtained. By repeatedly applying ET rules one by one, \(s_{14}\) is replaced by a singleton \(s_{17}\) with one clause.

\[s_{17} = \{\text{yes} \rightarrow \text{noteq}(V_a, V_b), \text{rev}(V_b, [\ ]\}\}

The following rule for \(\text{rev}\) predication which can be applied to the set of \(\text{rev}(V_a, V_c)\) and \(\text{app}(V_a, V_c)\) is generated by meta-computation method.

\[r_8 : \text{rev}(V_x, V_y), \text{app}(V_y, V_x) \Rightarrow \text{rev}(V_x, V_y)\]

Since \(r_8\) is added to \(\mathbb{R}\), \(r_8\) can be applied to the conjuction of \(\text{rev}(V_a, V_c)\) and \(\text{app}(V_a, V_c)\) in a definite clause.
set $s_{17}$. Consequently, a definite clause set $s_{18}$ is obtained.

An Inductively-used rule $r_5$ is applied to the conjunction of $\text{rev}(Vb, \{Vc \in Vf\})$ and $\text{rev}(\{Ve \in Vf\}, Ve)$ in a definite clause set $s_{19}$. Consequently, a definite clause set $s_{20}$ is obtained. Finally, $r_4$ is applied to noteq($Vb$, $Vb$) in a definite clause set $s_{20}$, and consequently an empty set is obtained.

Since an empty set is obtained from cls, $\forall r \text{rev}(Vx, Vy) \rightarrow \text{eq}(Vx, Vz)$ can be proven. Consequently, the following Speq rule is generated.

$$\text{rev}(Vx, Vy), \text{rev}(Vz, Vz) \Rightarrow eq(Vx, Vz), \text{rev}(Vx, Vy), \text{rev}(Vy, Vz).$$

Since a Speq rule cannot be obtained from a pair of $X$ and $Y$ other than a pair of $X = Vx$ and $Y = Vz$, the following set $S$ is obtained as an output.

$$S = \{\text{rev}(Vx, Vy), \text{rev}(Vz, Vz) \Rightarrow eq(Vx, Vz), \text{rev}(Vx, Vy), \text{rev}(Vy, Vz)\}$$

VI. DISCUSSIONS

A. Comparison of the proposed algorithm and current meta-computation method

This section discusses the following two topics based on the comparison of the proposed algorithm and a current meta-computation method. 1. The proposed algorithm can generate more Speq rules than a meta-computation method can do. 2. When a Speq rule with respect to a certain input is generated by the proposed algorithm, it is important that the form of the Speq rule is understood before rule generation.

The proposed algorithm is a special method for generating Speq rules. By specializing Speq rules, the proposed algorithm can generate more Speq rules than a meta-computation method can do. For example, we show the following four Speq rules ($r_1$, $r_2$, $r_3$ and $r_4$). The proposed algorithm can generate all the Speq rules, but a meta-computation method can generate only $r_2$ and $r_4$.

$$r_1 : \text{rev}(Vx, Vz), \text{rev}(Vz, Vz) \Rightarrow eq(Vx, Vz), \text{rev}(Vx, Vy), \text{rev}(Vy, Vz).$$

$$r_2 : \text{rev}(Vx, Vz), \text{rev}(Vx, Vy), \text{rev}(Vx, Vz) \Rightarrow eq(Vz, Vw), \text{rev}(Vx, Vz), \text{rev}(Vx, Vz), \text{rev}(Vx, Vz) \Rightarrow eq(Vx, Vz), \text{rev}(Vx, Vz).$$

$$r_3 : \text{app}(Vx, []), Vz \Rightarrow eq(Vx, Vz), \text{app}(Vx, []), Vz \Rightarrow eq(Vx, Vz), \text{app}(Vx, []), Vz.$$

$$r_4 : \text{app}(Vx, Vz), Vy \Rightarrow eq(Vx, Vz), \text{app}(Vx, Vz), Vy \Rightarrow eq(Vx, Vz), \text{app}(Vx, Vz), Vy \Rightarrow eq(Vx, Vz), \text{app}(Vx, Vz), Vy.$$

The head of $r_1$ and $r_2$ is the conjunction of two $rev$ atoms. $r_2$ can be generated by a meta-computation method, but $r_1$ cannot be generated by it. We explain the reason why the proposed method can generate more Speq rules than a meta-computation method can do. When Speq rules are generated by the proposed algorithm, a form of the Speq rules obtained ultimately is understood before Speq rule generation; a meta-computation, however, does not provide such projection. This gives the proposed algorithm a big advantage in generating Speq rules, i.e., a Speq rule obtained ultimately can be inductively used during the process of Speq rule generation in the proposed algorithm.

We will show the effectiveness where rules to be inductively derived can be used. If $r_1$ is generated by a meta-computation method, then an atom set $\{\text{rev}(Vx, Vz), \text{rev}(Vy, Vz)\}$ is replaced to another one by a basic equivalent transformation and $r_1$ is generated by a finite synthesis of the transformations. The length of lists applicable to $Vx$ and $Vz$, respectively, is arbitrary. If the number of lists is fixed, the number of basic equivalent transformations is finite. However, since lists lengthen infinitely when generating $r_1$, $r_1$ cannot be generated by a finite basic equivalent transformation. Since rules to be finally derived are inductively used in the proposed algorithm, computation that needs an infinite transformation can be replaced with one that needs a finite transformation. Therefore, the proposed algorithm can generate Speq rules which need an infinite transformation.

B. Importance of Speq rules for generating other ET rules

In this section, we discuss the proposed algorithm in detail that it not only generates Speq rules efficiently but also expands its capability of generating other ET rules.

Although our interest is in the generation of Speq rules, we are also intrigued by the generation of other ET rules and, therefore, studying those as well. Also in the generation of other ET rules, a logical formula is proven using a rule set in order to guarantee the correctness of a logical equivalence. When an ET rule is generated, a Speq rule plays an important role. Therefore, the generation of Speq rules is essential.

For example, a Speq rule is necessary when generating the following ET rule.

$$\text{app}(Vx, Vz, Vw), \text{rev}(Vy, Vq) \Rightarrow \text{rev}(Vx, Vp), \text{rev}(Vy, Vq), \text{app}(Vq, Vp, Vw).$$

This ET rule is used when generating a Speq rule with respect to an atom set $\{\text{rev}(Vx, Vz), \text{rev}(Vy, Vz)\}$ (Refer to Section V-B). The above ET rule can be generated from the following logical equivalence.

$$\exists Vz [(\text{app}(Vx, Vz, Vw) \rightarrow \exists Vp, Vq[(\text{rev}(Vx, Vp), \text{rev}(Vy, Vq), \text{app}(Vq, Vp, Vw))].$$

A Speq rule required in the above rule generation is one with respect to an atom set $\{\text{app}(Vx, [ ], Vz)\}$ (Refer to Section V-A). This Speq rule cannot be generated by a meta-computation method as shown in the previous section.

Therefore, it can be suggested that we have expanded the capability of generating other ET rules by proposing an algorithm by which Speq rules are generated.

VII. CONCLUSIONS

We focused on goal-oriented rule generation where the rule to be eventually obtained is designated, and proposed an algorithm for generating Speq rules. The proposed algorithm made it possible to efficiently generate Speq rules difficult to generate with the conventional rule-generation method. When an atom set is given as an input, the proposed algorithm outputs all Speq rules relating to the input atom set based on proof of a logical formula. By changing the logical formula,
variety of ET rules can be generated. A distinctive feature of this algorithm is that it includes a rule obtained ultimately in a rule set used for proving a logical formula. When generating a Speq rule, all ET rules necessary for generating the Speq rule are found and successively generated by recursively using the proposed algorithm. Future works include development of new methods for efficiently generating other ET rules that are important for solving many problems.

References


