Electronic Notes in Theoretical Computer Science 59 No. 4 (2001)
URL: `http://www.elsevier.nl/locate/entcs/volume59.html` 16 pages

A Class of Rewriting Rules and Reverse Transformation for Rule-based Equivalent Transformation

Kiyoshi Akama [1,2]

Center for Information and Multimedia Studies
Hokkaido University
Sapporo, Hokkaido, 060-0811, Japan

Ekawit Nantajeewarawat [3,4]

IT Program, Sirindhorn International Institute of Technology
Thammasat University, Rangsit Campus
P.O. Box 22, Thammasat-Rangsit Post Office, Pathumthani 12121, Thailand

Hidekatsu Koike [5]

Division of System and Information Engineering
Hokkaido University
Sapporo, Hokkaido, 060-0811, Japan

Abstract

In the rule-based equivalent transformation (RBET) paradigm, where computation is based on meaning-preserving transformation of declarative descriptions, a set of rewriting rules is regarded as a program. The syntax for a large class of rewriting rules is determined. The incorporation of meta-variables of two different kinds enables precise control of rewriting-rule instantiations. As a result, the applicability of rewriting rules and the results of rule applications can be rigorously specified. A theoretical basis for justifying the correctness of rewriting rules is established. Reverse transformation operation in the RBET framework is discussed, and it is shown that a correct rewriting rule is reversible, i.e., a correct rewriting rule can in general be constructed by syntactically reversing another correct rewriting rule.

[1] Akama was partly supported by Grant-in-Aid for Scientific Research (B)(2) #12480076.
[2] Email: `akama@cims.hokudai.ac.jp`
[3] Nantajeewarawat was supported partially by the Thailand Research Fund.
[4] Email: `ekawit@siit.tu.ac.th`
[5] Email: `koke@cims.hokudai.ac.jp`

©2001 *Published by Elsevier Science B. V.* Open access under CC BY-NC-ND license.

1 Introduction

Rule-based equivalent transformation of declarative descriptions (RBET) [1] is a new promising method of problem solving. In the RBET framework, a problem is formulated as a *declarative description*, represented by the union of two sets of definite clauses, one of which is called the *definition part*, and the other the *query part*. The definition part provides general knowledge about the problem domain and descriptions of some specific problem instances. The query part specifies a question regarding the content of the definition part. From the definition part, a set of *rewriting rules*—rules for transforming declarative descriptions—is prepared. The problem is then solved by transforming the query part successively, using the prepared rewriting rules, into another set of definite clauses from which the answers to the specified question can be obtained easily and directly.

Example 1.1 Consider a simple problem formulated as the union of a definition part D_{init} consisting of the four definite clauses

$$\begin{array}{l} initial(X, Z) \leftarrow append(X, Y, Z) \\ append([\,], Y, Y) \leftarrow \\ append([A|X], Y, [A|Z]) \leftarrow append(X, Y, Z) \\ equal(X, X) \leftarrow \end{array}$$

and a query part Q containing only the definite clause

$$C_1: \quad ans(X) \leftarrow initial(X, [1, 2, 3]), initial(X, [1, 3, 5]).$$

To solve this problem, i.e., to find the answers to the query part Q, by means of RBET, Q will be transformed successively, using some rewriting rules prepared from D_{init}, until the simpler query part Q' consisting of the two unit clauses

$$\begin{array}{l} ans([\,]) \leftarrow \\ ans([1]) \leftarrow \end{array}$$

is obtained, from which the answers, i.e., $X = [\,]$ and $X = [1]$, can be directly drawn. One possible successive transformation of Q into Q' is demonstrated in the appendix. $\square$

A rewriting rule specifies, in its left-hand side, a pattern of atomic formulas (atoms) to which it can be applied, and defines the result of its application by specifying, in its right-hand side, one or more patterns of replacement atoms. The rule is applicable to a definite clause when the pattern in the left-hand side matches atoms contained in the body of the clause—in other words, when atoms contained in the body of the clause are instances of the specified pattern. When applied, the rule rewrites the clause into a number of clauses, resulting from replacing the matched body atoms with instances of the patterns in the right-hand side of the rule. Determination of rule applicability by pattern matching, rather than unification, allows one to tailor a rewriting rule for some specific pattern of atoms for the sake of computation efficiency.

Illustrations of rewriting rules are deferred until Section 2.

The crucial roles of atom patterns in determining rule applicability and specifying the results of rule applications necessitate an appropriate syntactic structure for representing the patterns in such a way that their instantiations can be precisely and suitably controlled. For this purpose, the notion of *meta-atom* is introduced. Meta-atoms have the same structure as usual atoms except that two kinds of *meta-variables*—&-variables and #-variables—are used instead of ordinary variables. The two kinds of meta-variables have different instantiation characteristics. Not only do the differences allow precise specifications of rewriting rules; they enable rigorous investigation of several important properties of several kinds of transformation steps, e.g., correctness of expanding transformation [7], and, moreover, as shown in [3], systematic generation of correct rewriting rules from a problem specification.

In the RBET framework, the correctness of computation relies solely on the correctness of each transformation step. Given a declarative description $D \cup Q$, where D and Q represent the definition part and the query part, respectively, of a problem, the query part Q is said to be transformed correctly in one step into a new query part Q' by an application of a rewriting rule, if and only if the declarative descriptions $D \cup Q$ and $D \cup Q'$ are equivalent, i.e., they have the same declarative meaning. A rewriting rule is considered to be correct, if and only if its application always results in a correct transformation step. A correct rewriting rule will be referred to as an *Equivalent Transformation rule (ET rule).* If ET rules are employed in all transformation steps, the answers obtained by means of RBET are guaranteed to be correct.

1.1 Comparison Between RBET and the Logic Programming Paradigm

Computation

Although declarative descriptions considered in this paper have the same form as definite logic programs [5], computation in RBET differs significantly from that in logic programming. Computation in logic programming is based on logical deduction—computation is viewed as the process of constructing, based on the resolution principle [10], a proof of an existentially quantified query by finding variable substitutions, called *computed substitutions*, that make the query follow logically from a given logic program. In RBET, by contrast, computation is regarded as transformation of declarative descriptions rather than logical deduction.

Separation of Programs from Declarative Descriptions

In logic programming, a set of definite clauses has a dual function: it serves as a declarative description of a problem—it declaratively represents the knowledge about the problem domain and defines what the problem is—while at the same time functions as a program—it specifies how to solve the problem. The programming character of a set of definite clauses arises from viewing it as a description of a search whose structure is determined by interpreting the logical connectives and quantifiers as fixed search instructions [6]. The procedural expressive power of a logic programming language, such as Prolog, is limited by such fixed procedural interpretation and the fixed search strategy embedded in the proof procedure associated with the language.

In the RBET framework, instead of a set of definite clauses, a set of rewriting rules is regarded as a program. The procedural interpretation of definite clauses can be realized using rewriting rules of a basic kind, called *unfolding-based rewriting rules* [1]. However, several other rewriting rules can additionally be employed in RBET, thereby a wider variety of computation paths are allowed and a more efficient program can consequently be achieved [1]. The use of a set of rewriting rules as a program also enables flexible computation—an effective control strategy can be materialized by means of, for example, rule-firing control and user-defined priority-based selection of rules [1].

Theoretical Foundation for Correctness
While the correctness of computation in RBET is based solely on meaning preservation of declarative descriptions, the correctness of computation in logic programming is grounded upon the logical consequence relation ($\models$), i.e., given a logic program P and an atom q, a computed substitution θ is correct if and only if $P \models \forall(q\theta)$. The notion of logical consequence in turn relies on the elementary concepts, e.g., the concepts of interpretation, satisfaction, and model, of the model theory associated with first-order logic. These concepts are not necessary in the RBET framework.

The correctness of computation in logic programming cannot be guaranteed by the correctness of inference rules solely; it also depends on the computation procedure employed. When the computation procedure is improved or extended, the correctness of the procedure as a whole has to be proven. In comparison, to verify the correctness of computation in RBET, it suffices to prove the correctness of each individual rewriting rule. A program in the RBET framework can therefore be decomposed; consequently, RBET-based systems are amenable to modification and extension.

1.2 Comparison Between RBET and Program Transformation in Logic Programming

Objectives and Transformed Parts
The objective of RBET is different from that of program transformation in logic programming (PT) [8,9]. While RBET is a method for computing the answers to a question with respect to a given definition part, PT is a methodology for deriving an efficient logic program from the definition part. Let a definition part D_0 be given. In RBET, to compute the answers to a query part Q_0 with respect to D_0, one constructs from Q_0, by successive application of rewriting rules prepared from D_0, a sequence $Q_0, \ldots, Q_n$ such that for each

i $(0 \leq i < n)$, $D_0 \cup Q_i$ and $D_0 \cup Q_{i+1}$ have the same declarative meaning and the answers can be directly obtained from Q_n. The definition part D_0 is unchanged throughout the transformation process. In comparison, in PT only the definition part is transformed. That is, from D_0, which is regarded as the initial logic program, one constructs by using transformation rules, such as the unfolding and folding rules, a sequence of logic programs $D_0, \ldots, D_m$ such that D_0 and D_m yield the same answers to some class of queries, but D_m is more efficient than D_0; then, when a query in that class is given, the program D_m will be used for computing the answers to the query by means of some proof procedure.

Example 1.2 Consider the definition part D_{init} of Example 1.1. Following PT, D_{init} may be transformed successively, using the unfolding and folding rules, into the logic program D'_{init}:

$$\begin{array}{l} initial([\,], Y) \leftarrow \\ initial([A|X], [A|Z]) \leftarrow initial(X, Z) \\ append([\,], Y, Y) \leftarrow \\ append([A|X], Y, [A|Z]) \leftarrow append(X, Y, Z) \end{array}$$

D_{init} and D'_{init} have the same declarative meaning with respect to the predicates *initial* and *append*; however, computing the answers to a query containing the predicate *initial* using D'_{init} requires fewer number of resolution steps than using D_{init}. □

In PT the efficiency of the program resulting from a transformation process, rather than the transformation process itself, is the primary concern. In RBET, on the other hand, as transformation is the main computation mechanism, transformation processes are required to be efficient. The efficiency of a transformation process in RBET is achieved by the employment of efficient rewriting rules and appropriate rule-application control strategies [1].

Correctness and Independence of Rules

In PT, a transformation step which derives D_{k+1} from a transformation sequence $D_0, \ldots, D_k$ is correct, if and only if for each query q containing only predicate symbols which occur in D_k, D_k and D_{k+1} provide the same answers to q. The correctness of a transformation step in PT can in general not be determined independently; e.g., the correctness of a folding step deriving D_{k+1} from a transformation sequence $D_0, \ldots, D_k$ requires some conditions to ensure that enough unfolding steps have been performed in the sequence $D_0, \ldots, D_k$ [9]. The next example shows that an application of the folding rule may yield an incorrect transformation step.

Example 1.3 Refer to the definition part D_{init} of Example 1.1. Folding the first clause, i.e.,

$$initial(X, Z) \leftarrow append(X, Y, Z),$$

using itself results in the logic program D''_{init}:

$$\begin{aligned}&initial(X, Z) \leftarrow initial(X, Z)\\&append([\,], Y, Y) \leftarrow\\&append([A|X], Y, [A|Z]) \leftarrow append(X, Y, Z)\end{aligned}$$

Since the meaning of the predicate *initial* defined in D_{init} is lost in D''_{init}, this transformation step does not preserve the answers to queries concerning the predicate *initial* and is therefore not correct. □

In RBET, by contrast, since only a query part, which depends exclusively on a fixed definition part, is transformed, the correctness of a transformation step can be justified independently, i.e., given a definition part D, the correctness of a transformation step deriving a query part Q_{j+1} from a query part Q_j is determined by the meanings of $D \cup Q_j$ and $D \cup Q_{j+1}$ solely, regardless of its preceding transformation steps. Consequently, the correctness of a rewriting rule can also be determined independently in the RBET framework. Such independence of rewriting rules is apparently desirable for the construction of large-scale rule-based systems.

1.3 Objectives of the Paper

Syntax for Rewriting Rules. The first objective of this paper is to determine appropriate syntax for a large class of rewriting rules. The syntactic structure of rewriting-rule components as well as their instantiations should be suitably defined in order that they can be used to precisely specify rule applicability and the results of rule applications.

Theoretical Framework for Correctness of Rewriting Rules. The next objective is to establish, based on meaning-preserving transformation of declarative descriptions rather than logical inference, a theoretical framework for discussing the correctness of rewriting rules.

Reverse Transformation. The third objective is to introduce the reverse transformation operation, and to show that in the RBET framework an ET rule is reversible, i.e., one can obtain a rewriting rule the operation of which reverses that of another rewriting rule by syntactically reversing the latter rewriting rule, and the correctness of the former depends solely on the correctness of the latter.

Section 2 explains the necessity of meta-variables, and provides introductory examples of rewriting rules, reverse transformation, and reverse rewriting rules. Section 3 defines preliminary syntactic components, which are used for defining declarative descriptions and their meanings in Section 4, and rewriting rules, their applications, and their correctness in Section 5. Section 6 investigates the correctness of reverse rewriting rules.

2 Meta-Variables and Reverse Rewriting Rules

The need for the use of meta-variables of two distinct kinds for specifying patterns of atoms, and the necessity of conditions for regulating meta-variable instantiations will be described first. Reverse transformation operation and reverse rewriting rules will then be introduced.

2.1 Need for Meta-Variables of Two Kinds

Consider the definition part D_{init} and the query parts Q and Q' of Example 1.1. As the first step of a possible transformation sequence leading to Q', the clause

C_1: $ans(X) \leftarrow initial(X, [1, 2, 3]), initial(X, [1, 3, 5])$

in Q may be transformed by replacing its first body atom with $append(X, Y, [1, 2, 3])$, resulting in the clause

C_2: $ans(X) \leftarrow append(X, Y, [1, 2, 3]), initial(X, [1, 3, 5])$.

This transformation step is correct since $D_{init} \cup \{C_1\}$ and $D_{init} \cup \{C_2\}$ have the same meaning.

The above transformation step can be described by the rewriting rule

r_1: $initial(\&X, \&Z) \;\rightarrow\; append(\&X, \&Y, \&Z)$,

where the arrow "$\rightarrow$" intuitively means "can be replaced with" and the left-hand side and the right-hand side of r_1 specify the pattern of atoms to which the rule is applicable and the pattern of replacement atoms, respectively. The symbols $\&X, \&Y$ and $\&Z$ are used in r_1 as instantiation wild cards, i.e., each of them can be instantiated into an arbitrary term, and also as equality constraints, i.e., each occurrence of the same wild card must be instantiated into the same term. By instantiating $\&X, \&Y$ and $\&Z$ into the terms X, Y and $[1, 2, 3]$, respectively, the pattern in the left-hand side matches the first body atom of C_1 and that in the right-hand side is instantiated into the first body atom of C_2—that is, by applying r_1 to the first body atom of C_1 using this instantiation, C_1 is transformed into C_2.

The dual role of the symbols $\&X, \&Y$ and $\&Z$ as wild cards and equality constraints is reminiscent of the concept of variable. Notwithstanding, these symbols should be distinguished from ordinary variables that are used in definite clauses since they are used differently; for example, they can be instantiated into ordinary variables but they are not substituted for ordinary variables in any substitution application. To emphasize the differences, the symbols $\&X, \&Y$ and $\&Z$ will be regarded as meta-variables, and will be referred to as &-variables.

However, the rewriting rule r_1 does not always specify a correct transformation step. For example, the application of r_1 to the first body atom of C_1 by instantiating $\&Y$ into the variable X transforms C_1 into the clause

C_3: $ans(X) \leftarrow append(X, X, [1, 2, 3]), initial(X, [1, 3, 5])$, but $D_{init} \cup \{C_1\}$ and $D_{init} \cup \{C_3\}$ have different meanings.

To ensure a correct transformation step, some restrictions on rule instantiations are required. Another kind of meta-variable, called #-variables, is introduced for this purpose. As an example, a #-variable, $\#Y$, will be used instead of the &-variable $\&Y$ in the right-hand side of r_1, i.e., the rule

$$r_2\text{:}\quad \mathit{initial}(\&X, \&Z) \ \rightarrow\ \mathit{append}(\&X, \#Y, \&Z)$$

will be used instead of r_1. Then, any instantiation of this rule is regulated in such a way that the #-variable $\#Y$ can only be instantiated into an ordinary variable that does not appear in the other part of the clause resulting from an application of the rule. This instantiation constraint precludes the instantiation of $\#Y$ into the ordinary variable X when the rule r_2 is applied to the first body atom of C_1; as a result, the transformation of C_1 into C_3 is prevented.

2.2 Reverse Transformation

In the RBET framework, the reverse of a correct transformation step is always a correct transformation step. For instance, from the step transforming C_1 into C_2 illustrated in the preceding subsection, one can have the reverse step transforming C_2 into C_1, which may be described by the rewriting rule

$$r_3\text{:}\quad \mathit{append}(\&X, \&Y, \&Z) \ \rightarrow\ \mathit{initial}(\&X, \&Z),$$

and the correctness of the latter step follows from the correctness of the former step. In general, however, the application of the rule r_3 may result in an incorrect transformation step. For example, by instantiating the &-variable $\&Y$ into X, the application of r_3 to the first body atom of the clause C_3 of the previous subsection yields an incorrect transformation step deriving C_1 from C_3.

Again the employment of meta-variables of the two kinds, with different instantiation characteristics, remedies this problem. Instead of using r_3, the transformation of C_2 into C_1 can be described using the rewriting rule

$$r_4\text{:}\quad \mathit{append}(\&X, \#Y, \&Z) \ \rightarrow\ \mathit{initial}(\&X, \&Z),$$

while the application of r_4 to the first body atom of C_3 can be ruled out by appropriately restricting the instantiation of the #-variable $\#Y$, i.e., $\#Y$ is only allowed to be instantiated into a variable that does not occur in the other part of C_3.

Rigorous description of rewriting rules and their applications demands precise conditions for instantiations of meta-variables in rule applications. For the sake of generality and regularity, the conditions should not be specialized for any particular case, but common to all rewriting rules. Such common conditions will be defined in Section 5 (Conditions (MVI-1), (MVI-2), (MVI-3) and (RRA-2)).

Notice that the rule r_4 can be obtained by simply reversing the rule r_2 of the preceding subsection. It will be shown in Section 6 that in the RBET framework a correct rewriting rule can in general be constructed by reversing another correct rewriting rule.

3 Basic Syntactic Components

The alphabet used in the paper will now be given; then, the notions of term and atom, which are basic components of definite clauses and declarative descriptions, and those of meta-term and meta-atom, which are used for specifying patterns of terms and atoms, respectively, will be defined.

Alphabet

An *&-variable* is a variable that begins with the symbol &; for example, $\&N$ and $\&X$ are &-variables. A *#-variable* is a variable that begins with the symbol #; for example, $\#X$ and $\#Y$ are #-variables. An &-variable as well as a #-variable is called a *meta-variable*. An ordinary variable is assumed to begin with neither & nor #.

Throughout the paper, an alphabet $\Delta = \langle K, F, V, R\rangle$ is assumed, where K is a set of constants, including integers and *nil*; F a set of functions, including the binary function *cons*; V is the disjoint union of two sets

- V_1 of ordinary variables,
- V_2 of meta-variables;

and R is the union of two mutually disjoint sets of predicates

- $R_1 = \{\mathit{initial}, \mathit{append}, \mathit{equal}, \ \dots\ \}$,
- $R_2 = \{\mathit{ans}, \mathit{yes}, \ \dots\ \}$.

When no confusion is possible, an ordinary variable in V_1 and a meta-variable in V_2 will be simply called a variable and a meta-variable respectively.

Terms, Meta-Terms, Atoms, and Meta-Atoms

Usual first-order terms on $\langle K, F, V_1\rangle$ and on $\langle K, F, V_2\rangle$ will be referred to as *terms* and *meta-terms*, respectively, on Δ. Given $R' \subseteq R$, usual first-order atoms on $\langle K, F, V_1, R'\rangle$ and on $\langle K, F, V_2, R'\rangle$ will be referred to as *atoms on* R' and *meta-atoms on* R', respectively. For example, assume that $\{X, Y\} \subseteq V_1$ and $\{\&X, \#Y\} \subseteq V_2$. Then, *nil*, X and $\mathit{cons}(X, \mathit{cons}(Y, \mathit{nil}))$ are terms on Δ; *nil*, $\&X$ and $\mathit{cons}(\&X, \mathit{cons}(\#Y, \mathit{nil}))$ are meta-terms on Δ; $\mathit{initial}(X, \mathit{cons}(X, \mathit{cons}(Y, \mathit{nil})))$ is an atom on R_1; and $\mathit{initial}(\&X, \mathit{cons}(\&X, \mathit{cons}(\#Y, \mathit{nil})))$ is a meta-atom on R_1. The standard Prolog notation for lists is adopted; e.g., $[X, Y]$ and $[7, \#X|\&Y]$ are abbreviations for the term $\mathit{cons}(X, \mathit{cons}(Y, \mathit{nil}))$ and the meta-term $\mathit{cons}(7, \mathit{cons}(\#X, \&Y))$, respectively.

First-order atoms on $\langle K, F, \emptyset, R\rangle$ are called *ground atoms* on Δ. In the sequel, let $\mathcal{T}$ be the set of all terms on Δ, and $\mathcal{G}$ the set of all ground atoms on Δ; also let $\mathcal{A}_i$ and $\hat{\mathcal{A}}_i$ be the set of all atoms and the set of all meta-atoms, respectively, on R_i, where $i \in \{1, 2\}$.

4 Declarative Descriptions and Their Meanings

In general, the RBET framework can deal with several data structures other than usual first-order terms, e.g., multisets, strings; and a declarative description can be represented by a set of definite clauses extended with these data structures [2,4]. For simplicity, however, only usual terms are used in this paper; that is, a declarative description is a set of usual definite clauses. Definite clauses and declarative descriptions considered herein as well as the meanings of declarative descriptions will now be defined.

Definite Clauses and Declarative Descriptions

A *definite clause* C *on* Δ is an expression of the form $A \leftarrow Bs$, where A is an atom on R and Bs is a (possibly empty) set of atoms on R. The atom A is called the *head* of C, denoted by $head(C)$; the set Bs is called the *body* of C, denoted by $Body(C)$; each element of $Body(C)$ is called a *body atom* of C. When $Body(C) = \emptyset$, C will be called a *unit clause*. The set notation is used in the right-hand side of C so as to stress that the order of the atoms in $Body(C)$ is immaterial. However, for the sake of simplicity, the braces enclosing the body atoms in the right-hand side of a definite clause will often be omitted; e.g., the definite clause $ans(X) \leftarrow \{append(Y, X, Z), initial(Y, Z)\}$ will often be written as $ans(X) \leftarrow append(Y, X, Z), initial(Y, Z)$.

Let $i \in \{1, 2\}$. A definite clause C is said to be *from* R_1 *to* R_i, if and only if $Body(C) \subseteq \mathcal{A}_1$ and $head(C) \in \mathcal{A}_i$. A *declarative description from* R_1 *to* R_i is a set of definite clauses from R_1 to R_i. The set of all declarative descriptions from R_1 to R_i will be denoted by $Dscr(R_1, R_i)$.

Meanings of Declarative Descriptions

Let $\mathcal{S}$ be the set of all substitutions on $\langle K, F, V_1\rangle$. The application of a substitution θ to an expression E (which can be, for example, a term, an atom, a set of atoms, or a definite clause) will be denoted by $E\theta$. Given a declarative description $P \in Dscr(R_1, R_i)$, the mapping T_P on $2^{\mathcal{G}}$ is given by

$$\begin{aligned} T_P(X) \;=\; \{head(C\theta) \mid {} & (C \in P) \;\&\; (\theta \in \mathcal{S}) \\ & \&\; (head(C\theta) \in \mathcal{G}) \;\&\; (Body(C\theta) \subseteq X)\}, \end{aligned}$$

and then, the meaning of P, denoted by $\mathcal{M}(P)$, is defined by

$$\mathcal{M}(P) \;=\; T_P^1(\emptyset) \cup T_P^2(\emptyset) \cup T_P^3(\emptyset) \cup \cdots \;=\; \textstyle\bigcup_{n=1}^{\infty} T_P^n(\emptyset),$$

where $T_P^1(\emptyset) = T_P(\emptyset)$ and $T_P^n(\emptyset) = T_P(T_P^{n-1}(\emptyset))$ for each $n \geq 2$.

5 Rewriting Rules, Their Applications, and Their Correctness

The syntax for a large class of rewriting rules is next presented. Coupled with some restrictions on meta-variable instantiations, this syntax enables one to control the applicability of rewriting rules and to specify the results of rule applications in a precise way.

Syntax of Rewriting Rules

A *rewriting rule on* R_1 takes the form

$$\begin{array}{lll} \hat{H}s & \rightarrow & \hat{B}s_1; \\ & & \cdots \\ & \rightarrow & \hat{B}s_n, \end{array}$$

where $n \geq 0$, and $\hat{H}s$ and the $\hat{B}s_i$ are subsets of $\hat{\mathcal{A}}_1$. For the sake of simplicity, the braces enclosing the meta-atoms in each side of a rewriting rule may be omitted; e.g., the rewriting rule $\{\mathit{initial}(\&X, \&Z)\} \rightarrow \{\mathit{append}(\&X, \#Y, \&Z)\}$ will also be written as $\mathit{initial}(\&X, \&Z) \rightarrow \mathit{append}(\&X, \#Y, \&Z)$.

Meta-Variable Instantiations

A *meta-variable instantiation* is a mapping θ from V_2 to $\mathcal{T}$ that satisfies the following three conditions:

(MVI-1) For each #-variable v, $\theta(v)$ is a variable.

(MVI-2) For any distinct #-variables v and v', $\theta(v) \neq \theta(v')$.

(MVI-3) For any &-variable u and #-variable v, $\theta(v)$ does not occur in $\theta(u)$.

Let $\hat{E}$ be an expression containing meta-variables ($\hat{E}$ can be, for example, a meta-term, a meta-atom, or a set of meta-atoms). Then, given a meta-variable instantiation θ, let $\hat{E}\theta$ denote the expression obtained from $\hat{E}$ by simultaneously replacing each occurrence of each meta-variable u in $\hat{E}$ with $\theta(u)$.

Applicability of Rewriting Rules

Let r be a rewriting rule on R_1

$$\begin{array}{lll} \hat{H}s & \rightarrow & \hat{B}s_1; \\ & & \cdots \\ & \rightarrow & \hat{B}s_n, \end{array}$$

where $n \geq 0$, and $\hat{H}s$ and the $\hat{B}s_i$ are subsets of $\hat{\mathcal{A}}_1$. Let C be a definite clause

$$A \leftarrow Bs \cup Bs'$$

from R_1 to R_2. The rewriting rule r is said to be *applicable* to C at Bs by using a meta-variable instantiation θ, if and only if the following conditions are both satisfied:

(RRA-1) $\hat{H}s\theta = Bs$.

(RRA-2) For any #-variable v, $\theta(v)$ occurs in neither A nor Bs'.

When r is applied to C at Bs by using the meta-variable instantiation θ, it rewrites C into n definite clauses $C_1, \dots, C_n$, where for each i $(1 \leq i \leq n)$,

$$C_i \;=\; (A \;\leftarrow\; \hat{Bs}_i\theta \cup Bs').$$

When Bs is a singleton set $\{B\}$, the application of r to C at Bs will also be referred to as the application of r to the body atom B of C.

When there are more than one applicable rewriting rule, one of them will be nondeterministically selected; hence, computation in RBET is nondeterministic.

Examples illustrating the application of rewriting rules are given below.

Example 5.1 Refer to the rewriting rules r_2 and r_4 and the definite clauses C_1, C_2 and C_3 of Section 2. Let $\theta : V_2 \to \mathcal{T}$ such that $\theta(\&X) = X$, $\theta(\#Y) = Y$, $\theta(\&Z) = [1,2,3]$ and θ satisfies Conditions (MVI-1), (MVI-2) and (MVI-3). Then, since Y occurs in neither the head nor the second body atom of C_1, r_2 can be applied to C_1 at $\{\mathit{initial}(X,[1,2,3])\}$ by using θ, and this application rewrites C_1 into C_2. Likewise, the application of r_4 to C_2 at $\{\mathit{append}(X,Y,[1,2,3])\}$ by using θ rewrites C_2 into C_1. Now consider the clause C_3. The rule r_4 is not applicable to C_3, since every $\sigma : V_2 \to \mathcal{T}$ such that

$$\mathit{append}(\&X, \#Y, \&Z)\sigma = \mathit{append}(X, X, [1,2,3])$$

requires that $\sigma(\&X) = X = \sigma(\#Y)$, violating Condition (MVI-3). □

Example 5.2 Consider the rewriting rule

$$\begin{array}{ll} r_5\text{:} & \mathit{append}(\&X, \&Y, \&Z) \\ & \quad \to \; \mathit{equal}(\&X, [\,]), \mathit{equal}(\&Y, \&Z); \\ & \quad \to \; \mathit{equal}(\&X, [\#A|\#X]), \mathit{equal}(\&Z, [\#A|\#Z]), \\ & \qquad \mathit{append}(\#X, \&Y, \#Z), \end{array}$$

and the clause

$$C_4\text{:} \quad \mathit{ans}(X) \leftarrow \mathit{append}(X, [E], [1,2]).$$

The application of the rule r_5 to C_4 transforms C_4 into the two definite clauses

$$C_5\text{:} \quad \mathit{ans}(X) \leftarrow \mathit{equal}(X, [\,]), \mathit{equal}([E], [1,2])$$

$$\begin{array}{ll} C_6\text{:} & \mathit{ans}(X) \leftarrow \mathit{equal}(X, [A1|X1]), \mathit{equal}([1,2], [A1|Z1]), \\ & \qquad\qquad \mathit{append}(X1, [E], Z1) \end{array}$$

by using a meta-variable instantiation θ such that $\theta(\&X) = X$, $\theta(\&Y) = [E]$, $\theta(\&Z) = [1,2]$, $\theta(\#A) = A1$, $\theta(\#X) = X1$ and $\theta(\#Z) = Z1$. The clause C_6 can be further transformed by the application of r_5. Notice that r_5 is also applicable to the clause C_2 of Section 2 at $\{\mathit{append}(X,Y,[1,2,3])\}$. □

Since the rule r_2 of Section 2 and the rule r_5 of Example 5.2 are applicable to an *initial*-atom of any pattern and an *append*-atom of any pattern, respectively, and their applications correspond to the unfolding operation, they will be referred to as *unfolding-based general* rewriting rules. The next example illustrates rewriting rules that are devised for atoms of specific patterns.

Example 5.3 Referring to the definition part D_{init} of Example 1.1, consider the query part consisting only of the clause C_4 of Example 5.2. Suppose that the rewriting rules prepared from the definition part D_{init} include the rules:

$$\begin{aligned} r_6\text{:}\quad & append(\&X, [\&E], [\&A, \&B|\&Z]) \\ & \quad \rightarrow\ equal(\&X, [\&A|\#W]), append(\#W, [\&E], [\&B|\&Z]) \\ r_7\text{:}\quad & append(\&X, [\&E], [\&A]) \rightarrow\ equal(\&X, [\,]), equal(\&E, \&A) \end{aligned}$$

The rewriting rule r_6 can be applied to C_4 at $\{append(X, [E], [1, 2])\}$, transforming C_4 into the clause

$$C_7\text{:}\quad ans(X) \leftarrow equal(X, [1|W]), append(W, [E], [2]).$$

Then, by applying the rule r_7 to C_7 at $\{append(W, [E], [2])\}$, C_7 can be transformed into the clause

$$C_8\text{:}\quad ans(X) \leftarrow equal(X, [1|W]), equal(W, [\,]), equal(E, 2),$$

from which the answer, $X = [1]$, can be derived. In comparison to the application of the rule r_5 in Example 5.2, notice that neither the application of r_6 nor that of r_7 increases the number of clauses in the query part. In general, the efficiency of computation can be improved by avoiding transformation steps that increase the number of clauses. □

Next, what it means for a rewriting rule to be correct is formally defined.

Correctness of Rewriting Rules
Let $D \in Dscr(R_1, R_1)$. A rewriting rule r on R_1 is *correct with respect to D and R_2*, if and only if for any declarative description $Q \in Dscr(R_1, R_2)$ and any definite clauses $C, C_1, \ldots, C_n$ from R_1 to R_2, if r rewrites C into $C_1, \ldots, C_n$, then

$$\mathcal{M}(D \cup Q \cup \{C\}) \ =\ \mathcal{M}(D \cup Q \cup \{C_1, \ldots, C_n\}).$$

6 Correctness of Reverse Rewriting Rules

Based on the established foundation for correctness of rewriting rules, it will now be shown that one can in general construct a correct rewriting rule by simply reversing another correct rewriting rule.

Theorem 6.1 (Correctness of Reverse Rewriting Rules)
Let $D \in Dscr(R_1, R_1)$. Let r be a rewriting rule

$$\hat{A}s \ \rightarrow\ \hat{B}s$$

on R_1. Let $reverse(r)$ be the rewriting rule

$$\hat{B}s \ \rightarrow\ \hat{A}s$$

on R_1. If r is correct with respect to D and R_2, then $reverse(r)$ is also correct with respect to D and R_2.

Proof.

Let Q be a declarative description in $Dscr(R_1, R_2)$, C a definite clause

$$C: \quad H \leftarrow Bs \cup Bs'$$

from R_1 to R_2, and let r be correct with respect to D and R_2. Suppose that $reverse(r)$ is applied to C at Bs by using a meta-variable instantiation θ. Then, $Bs = \hat{B}s\theta$ and $reverse(r)$ rewrites C into the clause

$$C': \quad H \leftarrow \hat{A}s\theta \cup Bs'.$$

It has to be shown that $\mathcal{M}(D \cup Q \cup \{C\}) = \mathcal{M}(D \cup Q \cup \{C'\})$. Clearly, by using the meta-variable instantiation θ, r is applicable to C' at the set $\hat{A}s\theta$. This application of r rewrites the set $\hat{A}s\theta$ in the body of C' into $\hat{B}s\theta$, which is equal to Bs. That is, C' is rewritten into C by this application. Since r is correct with respect to D and R_2, $\mathcal{M}(D \cup Q \cup \{C\})$ and $\mathcal{M}(D \cup Q \cup \{C'\})$ are equal. So $reverse(r)$ is correct with respect to D and R_2. □

7 Conclusions

Each resolution step in the proof procedures associated with logic programming corresponds to an unfolding transformation step in RBET, which can be realized by the employment of unfolding-based general rewriting rules. However, while resolution is the only means of inference in logic programming, a variety of other rewriting rules can be used in RBET. The RBET framework therefore allows a wider variety of computation paths and, as a result, more efficient programs. Despite its simplicity, the RBET framework enables the development of a solid theoretical basis for determining the correctness of rewriting rules of various kinds. As long as correct rewriting rules are used throughout a transformation process, correct computation is always obtained. Experimental RBET-based knowledge processing systems in various application domains have been implemented at Hokkaido University, and satisfactory results revealing the usefulness of the framework have been obtained.

In this paper, the syntax for a large class of rewriting rules is proposed. This class of rewriting rules can represent unfolding-based general rewriting rules (e.g., the rules r_2 and r_5 of Subsection 2.1 and Example 5.2, respectively), folding-like rules (e.g., the rule r_4 of Subsection 2.2), and rules that are applicable to atoms of specific patterns (e.g., the rules r_6 and r_7 of Example 5.3). By incorporation of meta-variables of two kinds (&-variables and #-variables), the proposed syntax facilitates precise control of rewriting-rule instantiations and applications, which is necessary for ensuring the correctness of computation. A theoretical basis for verifying the correctness of rewriting rules is formulated. The reverse transformation operation is introduced, and it is shown that in general a correct rewriting rule can be obtained by simply reversing another correct rewriting rule.

In addition to the necessity identified in this paper of the use of meta-variables of the two kinds for specifying atom patterns in rewriting rules, it is demonstrated in [3] that the distinction between these two kinds of meta-variables also enables meaningful manipulation of atom patterns in the process of systematically generating rewriting rules from a definition part by means of meta-rules and is essential for controlling the generation process. Although reverse transformation may lead to an infinite loop in ordinary computation, it provides a foundation of folding-like meta-level transformation in the generation of rewriting rules and the correctness of reverse rewriting rules is essential for verifying the correctness of folding-like meta-rules.

Appendix

Referring to Example 1.1, Q can be transformed into Q' as follows. (The selected atom in each step is underlined.)

$$\begin{array}{rl}
1: & \mathit{ans}(X) \leftarrow \underline{\mathit{append}(X, Y1, [1,2,3])}, \mathit{initial}(X, [1,3,5]) \\
2: & \mathit{ans}([\,]) \leftarrow \underline{\mathit{initial}([\,], [1,3,5])} \\
 & \mathit{ans}([1|X1]) \leftarrow \mathit{append}(X1, Y1, [2,3]), \mathit{initial}([1|X1], [1,3,5]) \\
3: & \mathit{ans}([\,]) \leftarrow \underline{\mathit{append}([\,], Y2, [1,3,5])} \\
 & \mathit{ans}([1|X1]) \leftarrow \mathit{append}(X1, Y1, [2,3]), \mathit{initial}([1|X1], [1,3,5]) \\
4: & \mathit{ans}([\,]) \leftarrow \\
 & \mathit{ans}([1|X1]) \leftarrow \underline{\mathit{append}(X1, Y1, [2,3])}, \mathit{initial}([1|X1], [1,3,5]) \\
5: & \mathit{ans}([\,]) \leftarrow \\
 & \mathit{ans}([1]) \leftarrow \underline{\mathit{initial}([1], [1,3,5])} \\
 & \mathit{ans}([1|[2|X2]]) \leftarrow \mathit{append}(X2, Y1, [3]), \mathit{initial}([1|[2|X2]], [1,3,5]) \\
6: & \mathit{ans}([\,]) \leftarrow \\
 & \mathit{ans}([1]) \leftarrow \underline{\mathit{append}([1], Y3, [1,3,5])} \\
 & \mathit{ans}([1|[2|X2]]) \leftarrow \mathit{append}(X2, Y1, [3]), \mathit{initial}([1|[2|X2]], [1,3,5]) \\
7: & \mathit{ans}([\,]) \leftarrow \\
 & \mathit{ans}([1]) \leftarrow \underline{\mathit{append}([\,], Y3, [3,5])} \\
 & \mathit{ans}([1|[2|X2]]) \leftarrow \mathit{append}(X2, Y1, [3]), \mathit{initial}([1|[2|X2]], [1,3,5]) \\
8: & \mathit{ans}([\,]) \leftarrow \\
 & \mathit{ans}([1]) \leftarrow \\
 & \mathit{ans}([1|[2|X2]]) \leftarrow \mathit{append}(X2, Y1, [3]), \underline{\mathit{initial}([1|[2|X2]], [1,3,5])} \\
9: & \mathit{ans}([\,]) \leftarrow \\
 & \mathit{ans}([1]) \leftarrow \\
 & \mathit{ans}([1|[2|X2]]) \leftarrow \mathit{append}(X2, Y1, [3]), \underline{\mathit{append}([1|[2|X2]], Y4, [1,3,5])} \\
10: & \mathit{ans}([\,]) \leftarrow \\
 & \mathit{ans}([1]) \leftarrow \\
 & \mathit{ans}([1|[2|X2]]) \leftarrow \mathit{append}(X2, Y1, [3]), \underline{\mathit{append}([2|X2], Y4, [3,5])} \\
11: & \mathit{ans}([\,]) \leftarrow \\
 & \mathit{ans}([1]) \leftarrow
\end{array}$$

There are several other possible ways of transforming Q into Q', some of which may result in a sequence that is shorter than the one shown above.

References

[1] Akama, K., Shigeta, Y., and Miyamoto, E., *Solving Problems by Equivalent Transformation of Logic Programs*, in Proceedings of the Fifth International Conference on Information Systems Analysis and Synthesis (ISAS'99), Orlando, Florida, 1999.

[2] Akama, K., Kawaguchi, Y., and Miyamoto, E., *Equivalent Transformation for Equality Constraints on Multiset Domains* (in Japanese), Journal of the Japanese Society for Artificial Intelligence **13** (1998), pp. 395–403.

[3] Akama, K., Koike, H., and Miyamoto, E., *Program Synthesis from a Set of Definite Clauses and a Query*, in Proceedings of the Fifth International Conference on Information Systems Analysis and Synthesis (ISAS'99), Orlando, Florida, 1999.

[4] Akama, K., Okada, K., and Miyamoto, E., *A Foundation of Equivalent Transformation of Negative Constraints on String Domains* (in Japanese), IEICE Technical Report, SS97-91, pp. 33–40, 1998.

[5] Lloyd, J. W., "Foundations of Logic Programming", second, extended edition, Springer-Verlag, 1987.

[6] Loveland, D. W. and Nadathur, G., *Proof Procedures for Logic Programming*, in: Gabbay, D. M., Hogger, C. J., and Robinson, J. A. (eds.), "Handbook of Logic in Artificial Intelligence and Logic Programming", Vol. 5, Oxford University Press, 1998, pp. 163–234.

[7] Nantajeewarawat, E., Akama, K., and Koike, H., *Expanding Transformation as a Basis for Correctness of Rewriting Rules*, in Proceedings of the Second International Conference on Intelligent Technologies (InTech'01), Bangkok, Thailand, 2001.

[8] Pettorossi, K. and Proietti, M., *Transformation of Logic Programs: Foundations and Techniques*, Journal of Logic Programming **19/20** (1994), pp. 261–320.

[9] Pettorossi, K. and Proietti, M., *Transformation of Logic Programs*, in: Gabbay, D. M., Hogger, C. J., and Robinson, J. A. (eds.), "Handbook of Logic in Artificial Intelligence and Logic Programming", Vol. 5, Oxford University Press, 1998, pp. 697–787.

[10] Robinson, J. A., *Machine-Oriented Logic Based on the Resolution Principle*, Journal of the ACM **12** (1965), pp. 23–41.