*Electronic Notes in Theoretical Computer Science 59 No. 4 (2001)*

*URL:* [*http://www.elsevier.nl/locate/entcs/volume59.html*](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 OÆce, 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 di erent 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 speci ed. 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 Scienti c Research (B)(2) #12480076.*

*2 Email:* [*akama@cims.hokudai.ac.jp*](mailto:akama@cims.hokudai.ac.jp)

*3 Nantajeewarawat was supported partially by the Thailand Research Fund.*

*4 Email:* [*ekawit@siit.tu.ac.th*](mailto:ekawit@siit.tu.ac.th)

*5 Email:* [*koke@cims.hokudai.ac.jp*](mailto:koke@cims.hokudai.ac.jp)

*c 2001 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license.*](http://creativecommons.org/licenses/by-nc-nd/3.0/)

# *1 Introduction*

*Rule-based equivalent transformation of declarative descriptions (RBET) [1] is* a new promising method of problem solving. In the RBET framework, a prob- lem is formulated as a declarative description, represented by the union of two sets of de nite clauses, one of which is called the de nition part, and the other the query part. The de nition part provides general knowledge about the prob- lem domain and descriptions of some speci c problem instances. The query part speci es a question regarding the content of the de nition part. From the de nition 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 de nite clauses from which the answers to the speci ed question can be obtained easily and directly.

*Example 1.1 Consider a simple problem formulated as the union of a de -* nition part Dinit consisting of the four de nite clauses

*initial(X; Z) append(X; Y; Z) append([ ]; Y;Y )*

*append([AjX]; Y; [AjZ]) append(X; Y; Z) equal(X; X)*

*and a query part Q containing only the de nite clause*

*C1: ans(X) initial(X; [1; 2; 3]); initial(X; [1; 3; 5]).*

*To solve this problem, i.e., to nd the answers to the query part Q, by means of* RBET, Q will be transformed successively, using some rewriting rules prepared from Dinit, until the simpler query part Q0 consisting of the two unit clauses

*ans([ ])*

*ans([1])*

*is obtained, from which the answers, i.e., X = [ ] and X = [1], can be directly* drawn. One possible successive transformation of Q into Q0 is demonstrated in the appendix. *2*

*A rewriting rule speci es, in its left-hand side, a pattern of atomic formulas* (atoms) to which it can be applied, and de nes 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 de nite 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 speci ed 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 uni cation, allows one to tailor a rewriting rule for some speci c pattern of atoms for the sake of computation eÆciency.

*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 di erent instantiation characteristics. Not only do the di erences allow precise speci cations 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 speci cation.

*In the RBET framework, the correctness of computation relies solely on the* correctness of each transformation step. Given a declarative description D[Q, where D and Q represent the de nition 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 Q0 by an application of a rewriting rule, if and only if the declarative descriptions D [ Q and D [ Q0 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 de nite logic programs [5], computation in RBET di ers signi cantly 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 quanti ed query by

*nding 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 de nite clauses has a dual function: it serves as a* declarative description of a problem|it declaratively represents the knowledge about the problem domain and de nes what the problem is|while at the same time functions as a program|it speci es how to solve the problem. The programming character of a set of de nite clauses arises from viewing

*it as a description of a search whose structure is determined by interpreting* the logical connectives and quanti ers as xed search instructions [6]. The procedural expressive power of a logic programming language, such as Prolog, is limited by such xed procedural interpretation and the xed search strategy embedded in the proof procedure associated with the language.

*In the RBET framework, instead of a set of de nite clauses, a set of rewrit-* ing rules is regarded as a program. The procedural interpretation of de nite clauses can be realized using rewriting rules of a basic kind, called unfolding- based rewriting rules [1]. However, several other rewriting rules can addition- ally be employed in RBET, thereby a wider variety of computation paths are allowed and a more eÆcient program can consequently be achieved [1]. The use of a set of rewriting rules as a program also enables exible computation| an e ective control strategy can be materialized by means of, for example, rule- ring control and user-de ned 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 (j=), i.e., given a logic program P and an atom q, a computed substitution is correct if and only if P j= 8(q ). 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 rst-order logic. These concepts are not necessary in the RBET framework.

*The correctness of computation in logic programming cannot be guaran-* teed by the correctness of inference rules solely; it also depends on the com- putation 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 suÆces 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 modi cation and extension.

*1.2 Comparison Between RBET and Program Transformation in Logic Pro-* gramming

*Objectives and Transformed Parts*

*The objective of RBET is di erent 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 de nition part, PT is a method- ology for deriving an eÆcient logic program from the de nition part. Let a de nition part D0 be given. In RBET, to compute the answers to a query part Q0 with respect to D0, one constructs from Q0, by successive application of rewriting rules prepared from D0, a sequence Q0;::: ; Qn such that for each

*i (0 i < n), D0 [ Qi and D0 [ Qi+1 have the same declarative meaning* and the answers can be directly obtained from Qn. The de nition part D0 is unchanged throughout the transformation process. In comparison, in PT only the de nition part is transformed. That is, from D0, 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 D0;::: ; Dm such that D0 and Dm yield the same answers to some class of queries, but Dm is more eÆcient than D0; then, when a query in that class is given, the program Dm will be used for computing the answers to the query by means of some proof procedure.

*Example 1.2 Consider the de nition part Dinit of Example 1.1. Following* PT, Dinit may be transformed successively, using the unfolding and folding

*rules, into the logic program D0 :*

*init*

*initial([ ];Y )*

*initial([AjX]; [AjZ]) initial(X; Z) append([ ]; Y;Y )*

*append([AjX]; Y; [AjZ]) append(X; Y; Z)*

*Dinit and D0*

*init*

*have the same declarative meaning with respect to the predi-*

*cates initial and append; however, computing the answers to a query contain-*

*ing the predicate initial using D0 requires fewer number of resolution steps*

*init*

*than using Dinit.* *2*

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

*Correctness and Independence of Rules*

*In PT, a transformation step which derives Dk+1 from a transformation se-* quence D0;::: ; Dk is correct, if and only if for each query q containing only predicate symbols which occur in Dk, Dk and Dk+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 Dk+1 from a transformation sequence D0;::: ; Dk requires some conditions to ensure that enough unfolding steps have been performed in the sequence D0;::: ; Dk [9]. The next example shows that an application of the folding rule may yield an incorrect transformation step.

*Example 1.3 Refer to the de nition part Dinit of Example 1.1. Folding the*

*rst clause, i.e.,*

*initial(X; Z) append(X; Y; Z),*

*using itself results in the logic program D00 :*

*init*

*initial(X; Z) initial(X; Z) append([ ]; Y;Y )*

*append([AjX]; Y; [AjZ]) append(X; Y; Z)*

*Since the meaning of the predicate initial de ned in Dinit is lost in D00*

*init*

*, this*

*transformation step does not preserve the answers to queries concerning the* predicate initial and is therefore not correct. *2*

*In RBET, by contrast, since only a query part, which depends exclusively* on a xed de nition part, is transformed, the correctness of a transformation step can be justi ed independently, i.e., given a de nition part D, the correct- ness of a transformation step deriving a query part Qj+1 from a query part Qj is determined by the meanings of D [ Qj and D [ Qj+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 rst objective of this paper is to determine* appropriate syntax for a large class of rewriting rules. The syntactic struc- ture of rewriting-rule components as well as their instantiations should be suitably de ned in order that they can be used to precisely specify rule ap- plicability 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 dis- cussing the correctness of rewriting rules.

*Reverse Transformation. The third objective is to introduce the reverse trans-* formation 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 lat- ter 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 introduc-* tory examples of rewriting rules, reverse transformation, and reverse rewriting rules. Section 3 de nes preliminary syntactic components, which are used for de ning declarative descriptions and their meanings in Section 4, and rewrit- ing 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 rst. Reverse transformation operation and reverse rewriting rules will then be introduced.

*2.1 Need for Meta-Variables of Two Kinds*

*Consider the de nition part Dinit and the query parts Q and Q0 of Example 1.1.* As the rst step of a possible transformation sequence leading to Q0, the clause

*C1: ans(X) initial(X; [1; 2; 3]); initial(X; [1; 3; 5])*

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

*C2: ans(X) append(X; Y; [1; 2; 3]); initial(X; [1; 3; 5]).*

*This transformation step is correct since Dinit [ fC1g and Dinit [ fC2g have* the same meaning.

*The above transformation step can be described by the rewriting rule r1: initial(&X; &Z) ! append(&X; &Y; &Z),*

*where the arrow \!" intuitively means \can be replaced with" and the left-*

*hand side and the right-hand side of r1 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 r1 as instantiation wild cards, i.e., each of them can be instantiated into an arbitrary term, and also as equality con- straints, 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 rst body atom of C1 and that in the right-hand side is instantiated into the rst body atom of C2|that is, by applying r1 to the rst body atom of C1 using this instantiation, C1 is transformed into C2.

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

*However, the rewriting rule r1 does not always specify a correct transfor-* mation step. For example, the application of r1 to the rst body atom of C1 by instantiating &Y into the variable X transforms C1 into the clause

*C3: ans(X) append(X; X; [1; 2; 3]); initial(X; [1; 3; 5]),*

*but Dinit [ fC1g and Dinit [ fC3g have di erent meanings.*

*To ensure a correct transformation step, some restrictions on rule instan-* tiations 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 r1, i.e., the rule

*r2: initial(&X; &Z) ! append(&X; #Y; &Z)*

*will be used instead of r1. 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 instantia- tion of #Y into the ordinary variable X when the rule r2 is applied to the rst body atom of C1; as a result, the transformation of C1 into C3 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 C1 into C2 illustrated in the preceding subsection, one can have the reverse step transforming C2 into C1, which may be described by the rewriting rule

*r3: append(&X; &Y; &Z) ! 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 r3 may result in an incorrect transformation step. For example, by instantiating the &-variable &Y into X, the application of r3 to the rst body atom of the clause C3 of the previous subsection yields an incorrect transformation step deriving C1 from C3.

*Again the employment of meta-variables of the two kinds, with di erent* instantiation characteristics, remedies this problem. Instead of using r3, the transformation of C2 into C1 can be described using the rewriting rule

*r4: append(&X; #Y; &Z) ! initial(&X; &Z),*

*while the application of r4 to the rst body atom of C3 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 C3.

*Rigorous description of rewriting rules and their applications demands pre-* cise 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 con- ditions will be de ned in Section 5 (Conditions (MVI-1), (MVI-2), (MVI-3) and (RRA-2)).

*Notice that the rule r4 can be obtained by simply reversing the rule r2 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 de nite clauses and declarative descriptions, and those of meta-term and meta-atom, which are used for spec- ifying patterns of terms and atoms, respectively, will be de ned.

*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 = hK; F; V; Ri 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

*V1 of ordinary variables,*

*V2 of meta-variables;*

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

*R1 = finitial; append; equal; ::: g,*

*R2 = fans; yes; ::: g.*

*When no confusion is possible, an ordinary variable in V1 and a meta-variable* in V2 will be simply called a variable and a meta-variable respectively.

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

*Usual rst-order terms on hK; F; V1i and on hK; F; V2i will be referred to as terms and meta-terms, respectively, on . Given R0 R, usual rst-order atoms on hK; F; V1; R0i and on hK; F; V2; R0i will be referred to as atoms on R0 and meta-atoms on R0, respectively. For example, assume that fX; Y g V1 and f&X; #Y g V2. Then, nil, X and cons(X; cons(Y; nil)) are terms on ; nil, &X and cons(&X; cons(#Y; nil)) are meta-terms on ; initial(X; cons(X; cons(Y; nil))) is an atom on R1; and initial(&X; cons(&X; cons(#Y; nil))) is a meta-atom on R1. The standard Prolog notation for lists is adopted; e.g., [X; Y ] and [7; #Xj&Y ] are abbreviations for the term cons(X; cons(Y; nil)) and the meta-term cons(7; cons(#X; &Y )), respectively.*

*First-order atoms on hK; F; ;; Ri are called ground atoms on . In the* sequel, let T be the set of all terms on , and G the set of all ground atoms on ; also let Ai and A^i be the set of all atoms and the set of all meta-atoms,

*respectively, on Ri, where i 2 f1; 2g.*

# *4 Declarative Descriptions and Their Meanings*

*In general, the RBET framework can deal with several data structures other* than usual rst-order terms, e.g., multisets, strings; and a declarative descrip- tion can be represented by a set of de nite clauses extended with these data structures [2,4]. For simplicity, however, only usual terms are used in this pa- per; that is, a declarative description is a set of usual de nite clauses. De nite clauses and declarative descriptions considered herein as well as the meanings of declarative descriptions will now be de ned.

*De nite Clauses and Declarative Descriptions*

*A de nite clause C on is an expression of the form A 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) = ;, 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 de nite clause will often be omitted; e.g., the de nite clause ans(X) fappend(Y; X; Z); initial(Y; Z)g will often be written as ans(X) append(Y; X; Z); initial(Y; Z).

*Let i 2 f1; 2g. A de nite clause C is said to be from R1 to Ri, if and only* if Body(C) A1 and head(C) 2 Ai. A declarative description from R1 to Ri is a set of de nite clauses from R1 to Ri. The set of all declarative descriptions from R1 to Ri will be denoted by Dscr(R1; Ri).

*Meanings of Declarative Descriptions*

*Let S be the set of all substitutions on hK; F; V1i. The application of a sub-* stitution to an expression E (which can be, for example, a term, an atom, a set of atoms, or a de nite clause) will be denoted by E . Given a declarative description P 2 Dscr(R1; Ri), the mapping TP on 2G is given by

*TP (X) = fhead(C ) j (C 2 P ) & ( 2 S)*

*& (head(C ) 2 G) & (Body(C ) X)g,* and then, the meaning of P , denoted by M(P ), is de ned by

*1*

*M(P ) = T 1 (;) [ T 2 (;) [ T 3 (;) [ = S T n(;),*

*P P P n=1 P*

*P*

*P*

*where T 1 (;) = TP*

*P*

*(;) and T n(;) = TP*

*(T n 1(;)) for each n 2.*

# *5 Rewriting Rules, Their Applications, and Their Cor-* rectness

*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 R1 takes the form*

*H^s !*

*!*

*B^s1;*

*B^sn,*

*where n 0, and H^s and the B^si are subsets of A^1. For the sake of simplic-* ity, the braces enclosing the meta-atoms in each side of a rewriting rule may be omitted; e.g., the rewriting rule finitial(&X; &Z)g ! fappend(&X; #Y; &Z)g will also be written as initial(&X; &Z) ! append(&X; #Y; &Z).

*Meta-Variable Instantiations*

*A meta-variable instantiation is a mapping from V2 to T that satis es the* following three conditions:

*(MVI-1) For each #-variable v, (v) is a variable.*

*(MVI-2) For any distinct #-variables v and v0, (v) 6= (v0).*

*(MVI-3) For any &-variable u and #-variable v, (v) does not occur in*

*(u).*

*Let E^*

*be an expression containing meta-variables (E^*

*can be, for example,*

*a meta-term, a meta-atom, or a set of meta-atoms). Then, given a meta-*

*variable instantiation , let*

*E^ denote the expression obtained from E^ by*

*simultaneously replacing each occurrence of each meta-variable u in E^*

*(u).*

*Applicability of Rewriting Rules* Let r be a rewriting rule on R1

*with*

*H^s !*

*!*

*B^s1;*

*B^sn,*

*where n 0, and H^s and the B^si are subsets of A^1. Let C be a de nite clause* A Bs [ Bs0

*from R1 to R2. The rewriting rule r is said to be applicable to C at Bs by* using a meta-variable instantiation , if and only if the following conditions are both satis ed:

*(RRA-1)*

*H^s = Bs.*

*(RRA-2) For any #-variable v, (v) occurs in neither A nor Bs0.*

*When r is applied to C at Bs by using the meta-variable instantiation , it* rewrites C into n de nite clauses C1;::: ; Cn, where for each i (1 i n),

*Ci = (A*

*B^si [ Bs0 ).*

*When Bs is a singleton set fBg, 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 nondeter- ministic.

*Examples illustrating the application of rewriting rules are given below.*

*Example 5.1 Refer to the rewriting rules r2 and r4 and the de nite clauses C1; C2 and C3 of Section 2. Let : V2 ! T such that (&X) = X, (#Y ) = Y ,*

*(&Z) = [1; 2; 3] and satis es Conditions (MVI-1), (MVI-2) and (MVI-3).* Then, since Y occurs in neither the head nor the second body atom of C1, r2 can be applied to C1 at finitial(X; [1; 2; 3])g by using , and this application rewrites C1 into C2. Likewise, the application of r4 to C2 at fappend(X; Y; [1; 2; 3])g by using rewrites C2 into C1. Now consider the clause C3. The rule r4 is not applicable to C3, since every : V2 ! T such that

*append(&X; #Y; &Z) = append(X; X; [1; 2; 3])*

*requires that (&X) = X = (#Y ), violating Condition (MVI-3).* *2*

*Example 5.2 Consider the rewriting rule r5: append(&X; &Y; &Z)*

*! equal(&X; [ ]); equal(&Y; &Z);*

*! equal(&X; [#Aj#X]); equal(&Z; [#Aj#Z]); append(#X; &Y; #Z),*

*and the clause*

*C4: ans(X) append(X; [E]; [1; 2]).*

*The application of the rule r5 to C4 transforms C4 into the two de nite clauses C5: ans(X) equal(X; [ ]); equal([E]; [1; 2])*

*C6: ans(X) equal(X; [A1jX1]); equal([1; 2]; [A1jZ1]);*

*append(X1; [E]; Z1)*

*by using a meta-variable instantiation such that (&X) = X, (&Y ) = [E],*

*(&Z) = [1; 2], (#A) = A1, (#X) = X1 and (#Z) = Z1. The clause C6 can be further transformed by the application of r5. Notice that r5 is also applicable to the clause C2 of Section 2 at fappend(X; Y; [1; 2; 3])g.* *2*

*Since the rule r2 of Section 2 and the rule r5 of Example 5.2 are applicable* to an initial-atom of any pattern and an append-atom of any pattern, respec- tively, 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 speci c patterns.

*Example 5.3 Referring to the de nition part Dinit of Example 1.1, consider* the query part consisting only of the clause C4 of Example 5.2. Suppose that the rewriting rules prepared from the de nition part Dinit include the rules:

*r6: append(&X; [&E]; [&A; &Bj&Z])*

*! equal(&X; [&Aj#W ]); append(#W; [&E]; [&Bj&Z])*

*r7: append(&X; [&E]; [&A]) ! equal(&X; [ ]); equal(&E; &A)*

*The rewriting rule r6 can be applied to C4 at fappend(X; [E]; [1; 2])g, trans- forming C4 into the clause*

*C7: ans(X) equal(X; [1jW ]); append(W; [E]; [2]).*

*Then, by applying the rule r7 to C7 at fappend(W; [E]; [2])g, C7 can be trans- formed into the clause*

*C8: ans(X) equal(X; [1jW ]); equal(W; [ ]); equal(E; 2),*

*from which the answer, X = [1], can be derived. In comparison to the applica-* tion of the rule r5 in Example 5.2, notice that neither the application of r6 nor that of r7 increases the number of clauses in the query part. In general, the eÆciency of computation can be improved by avoiding transformation steps that increase the number of clauses. *2*

*Next, what it means for a rewriting rule to be correct is formally de ned.*

*Correctness of Rewriting Rules*

*Let D 2 Dscr(R1; R1). A rewriting rule r on R1 is correct with respect to D and R2, if and only if for any declarative description Q 2 Dscr(R1; R2) and any de nite clauses C; C1;::: ; Cn from R1 to R2, if r rewrites C into C1;::: ; Cn, then*

*M(D [ Q [ fCg) = M(D [ Q [ fC1;::: ; Cng).*

# *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 2 Dscr(R1; R1). Let r be a rewriting rule

*A^s ! B^s*

*on R1. Let reverse(r) be the rewriting rule*

*B^s ! A^s*

*on R1. If r is correct with respect to D and R2, then reverse(r) is also correct* with respect to D and R2.

*Proof.*

*Let Q be a declarative description in Dscr(R1; R2), C a de nite clause C: H Bs [ Bs0*

*from R1 to R2, and let r be correct with respect to D and R2. Suppose that* reverse(r) is applied to C at Bs by using a meta-variable instantiation . Then, Bs = B^s and reverse(r) rewrites C into the clause

*C0: H*

*A^s [ Bs0.*

*It has to be shown that M(D [ Q [ fCg) = M(D [ Q [ fC0g). Clearly, by* using the meta-variable instantiation , r is applicable to C0 at the set A^s . This application of r rewrites the set A^s in the body of C0 into B^s , which

*is equal to Bs. That is, C0 is rewritten into C by this application. Since r is* correct with respect to D and R2, M(D [ Q [ fCg) and M(D [ Q [ fC0g) are equal. So reverse(r) is correct with respect to D and R2. *2*

# *7 Conclusions*

*Each resolution step in the proof procedures associated with logic program-* ming corresponds to an unfolding transformation step in RBET, which can be realized by the employment of unfolding-based general rewriting rules. How- ever, while resolution is the only means of inference in logic programming, a variety of other rewriting rules can be used in RBET. The RBET frame- work therefore allows a wider variety of computation paths and, as a result, more eÆcient 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 applica- tion 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 r2 and r5 of Subsection 2.1 and Example 5.2, respec- tively), folding-like rules (e.g., the rule r4 of Subsection 2.2), and rules that are applicable to atoms of speci c patterns (e.g., the rules r6 and r7 of Ex- ample 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 identi ed 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 de nition part by means of meta-rules and is essential for controlling the generation process. Although reverse transformation may lead to an in nite loop in ordinary computation, it provides a foundation of folding-like meta-level transformation in the genera- tion 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 Q0 as follows. (The* selected atom in each step is underlined.)

*1: ans(X) append(X; Y 1; [1; 2; 3]); initial(X; [1; 3; 5])*

*2: ans([ ]) initial([ ]; [1; 3; 5])*

*ans([1jX1]) append(X1;Y 1; [2; 3]); initial([1jX1]; [1; 3; 5])*

*3: ans([ ]) append([ ];Y 2; [1; 3; 5])*

*ans([1jX1]) append(X1;Y 1; [2; 3]); initial([1jX1]; [1; 3; 5]) 4: ans([ ])*

*ans([1jX1]) append(X1;Y 1; [2; 3]); initial([1jX1]; [1; 3; 5]) 5: ans([ ])*

*ans([1]) initial([1]; [1; 3; 5])*

*ans([1j[2jX2]]) append(X2;Y 1; [3]); initial([1j[2jX2]]; [1; 3; 5]) 6: ans([ ])*

*ans([1]) append([1];Y 3; [1; 3; 5])*

*ans([1j[2jX2]]) append(X2;Y 1; [3]); initial([1j[2jX2]]; [1; 3; 5]) 7: ans([ ])*

*ans([1]) append([ ];Y 3; [3; 5])*

*ans([1j[2jX2]]) append(X2;Y 1; [3]); initial([1j[2jX2]]; [1; 3; 5]) 8: ans([ ])*

*ans([1])*

*ans([1j[2jX2]]) append(X2;Y 1; [3]); initial([1j[2jX2]]; [1; 3; 5]) 9: ans([ ])*

*ans([1])*

*ans([1j[2jX2]]) append(X2;Y 1; [3]); append([1j[2jX2]];Y 4; [1; 3; 5]) 10: ans([ ])*

*ans([1])*

*ans([1j[2jX2]]) append(X2;Y 1; [3]); append([2jX2];Y 4; [3; 5]) 11: ans([ ])*

*ans([1])*

*There are several other possible ways of transforming Q into Q0, 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 Arti cial Intelligence 13 (1998), pp. 395{403.*

*[3] Akama, K., Koike, H., and Miyamoto, E., Program Synthesis from a Set of De nite 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 Arti cial 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 Arti cial 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.*