数理解析研究所講究録
918 巻 1995 年 24-33

Implementation of Conditional Term Rewriting Systems equipped with Meta-computation

Masanobu NUMAZAWA, Masahito KURIHARA and Azuma OHUCHI

Faculty of Engineering, Hokkaido University,
Sapporo, 060 Japan
[numasawa|kurihara|ohuchi]@huie.hokudai.ac.jp

Abstract:

We propose a framework of conditional term rewriting systems equipped with meta-computation supported by basic mechanisms called meta-transformation and base-transformation. The arguments of a redex, the current set of meta-conditional rewrite rules and the four stacks of the reduction machine are considered as meta-level objects in our system, and by the meta-transformation they are transformed into base-level objects. The base-transformation transforms base-level objects called meta-representation into meta-level objects.

1 Introduction

Meta-computation is a computational mechanism that allows computational systems to read and modify meta-information. Implementing meta-computation in high-level languages enables us to access meta-level with high-abstract interface. In recent years this notion can be found in several fields of computer science and artificial intelligence. In particular, in the field of intelligent systems meta-computation is often called meta-inference and plays an important role in designing complex systems.

In this paper we present a framework of conditional term rewriting systems equipped with meta-computation. Conditional term rewriting systems arise naturally in the algebraic specification of abstract data types. They are also important for integrating the functional and logic programming paradigms, and provide a natural computational mechanism for this integration. A conditional term rewriting systems (abbreviated as CTRS) is a set of conditional rewrite rules. Every conditional rewrite rule has the form $e \rightarrow e' \;\; if \;\; t_1 = t'_1, \ldots, t_n = t'_n$ with terms e, e', t_1, t'_1, $\ldots$, t_n and t'_n. Depending on the interpretation of the equality sign in the conditions of the rewrite rules, different rewrite relations can be associated with a given CTRS. In this paper we restrict ourselves to the following interpretation. In a join CTRS R the equality sign in the conditions of the rewrite rules is interpreted as joinability. Formally, $s \rightarrow s'$ if and only if there exist a rewrite rule $e \rightarrow e' \;\; if \;\; t_1 = t'_1, \ldots, t_n = t'_n \in R$, a substitution θ, a context $C[\,]$ such that $s \equiv C[e\theta]$, $s' \equiv C[e'\theta]$, and $t_i\theta \downarrow_R t'_i\theta$ for all $i \in \{1, \ldots, n\}$, where the symbol $\downarrow$ denotes the joinability, i.e., $s \downarrow t$ if and only if there exists a term u such that $s \rightarrow^* u \wedge t \rightarrow^* u$ ($\rightarrow^*$ is the reflexive-transitive closure of $\rightarrow$). The context $C[\,]$ and system R in this definition are not the subject of computation in conditional term rewriting systems; they are meta-level objects existing in the interpreter level. For some substitution θ, the value $x\theta$ on a variable x is a term but, in general, is considered as a part of program, and the user program does not consider it as data. Thus $x\theta$ is also thought as meta-level objects. On the other hand, in our system terms which consist of only constructors are considered as data. We have meta-transformation which transforms meta-level objects into data, and base-transformation which transforms data into meta-level objects and changes the execution environment of the system.

In our system, there are four stacks: the subject term stack T, the candidate rewrite stack D, the context stack X and the phase stack P. These stacks define a part of computation state, and play important role in executing meta-computation in conditional term rewriting systems.

The organization of this paper is as follows. In Sect.2 we briefly describe the specification of our system. We define the syntax, the operational semantics, and two basic facilities – meta-transformation and base-transformation. In Sect.3, we define meta-representation of terms, contexts, programs and stacks. In Sect.4, we give some applications of our system. Efficient implementation of meta-transformation is presented in Sect.5. In Sect.6 we briefly discuss further research topics.

2 Specification

2.1 Syntax

Let $\mathcal{F}$ be a set of function symbols, and $\mathcal{V}$ be a countably infinite set of variables, satisfying $\mathcal{F} \cap \mathcal{V} = \emptyset$. Every $F \in \mathcal{F}$ is associated with a natural number denoting its arity. Function symbols of arity 0 are called constants. The set of terms is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$, or simply by $\mathcal{T}$.

We introduce the following syntax to invoke meta-computation. If t, z are terms, then the $t : z$ is a meta-computational expression. Meta-computational expressions are not considered as terms, because they are not objects to be rewritten. They can appear only in left-hand sides or right-hand sides of rewrite rules; namely they work only as a part of program, not as data. An expression is either a term or a meta-computational expression. An object of the form $e \rightarrow e' \;\; if \;\; t_1 = t'_1, \ldots, t_n = t'_n$ is called a meta-conditional rewrite rule if e and e' are expressions and t_1, t'_1, ..., t_n and t'_n are terms. The objects e, e', $(t_1 = t'_1, \ldots, t_n = t'_n)$ are called the left-hand side, the right-hand side and the condition, respectively. Every meta-conditional rewrite rule is subject to the following two constraints:

1) the left-hand side e is not a variable,
2) variables which occur in the right-hand side e' and the condition $(t_1 = t'_1, \ldots, t_n = t'_n)$ also occur in e.

In this paper, conditional term rewriting systems are often called programs.

Meta-conditional rewrite rules may be classified into four types depending upon whether the left-hand side or the right-hand side is a term or a meta-computational expression. As we shall see later in the next subsection, the meta-conditional rewrite rules whose left-hand sides are meta-computational expressions invoke a special operation of meta-computation called the meta-transformation. Similarly, the rules whose right-hand sides are meta-computational expressions invoke a special operation of meta-computation called the base-transformation.

2.2 Meta-transformation and Base-transformation

Extending the operational semantics of the ordinary reduction relation, our system defines two transformation primitives called meta-transformation and base-transformation. Meta-transformation transforms meta-level objects into base-level objects. The current program and the contents of the four stacks (T, D, X, P) constitute the current state of the computation. When meta-transformation is invoked by a rule whose left-hand side is a meta-computational expression, meta-level objects of the current state are transformed into data which are accessible by the user program. These data are called meta-representation of the state. $term^{\wedge}$, $context^{\wedge}$, $rules^{\wedge}$, $Tstack^{\wedge}$, $Dstack^{\wedge}$, $Xstack^{\wedge}$, $Pstack^{\wedge}$ are functions which transform terms, programs, the subject term stack, the candidate rewrite rules stack, the context stack and the phase stack, respectively, into constructor terms.

Base-transformation is the inverse of the meta-transformation, in which base-level objects are transformed back into the corresponding meta-level objects. This transformation is invoked by a rule whose right-hand side is a meta-computational expression. Note that this transformation is not defined when the meta-representation is not valid data. $term^{\vee}$, $context^{\vee}$, $rules^{\vee}$, $Tstack^{\vee}$, $Dstack^{\vee}$, $Xstack^{\vee}$, $Pstack^{\vee}$ are the inverse of the foregoing functions, i.e., they transform constructor terms into terms, contexts, programs, the subject term stack, the candidate rewrite rules stack, the context stack and the phase stack, respectively.

2.3 Reduction

A state of the reduction machine is the tuple $(T, D, X, P, \mathcal{R})$ of four stacks and a program: the subject term stack T, the candidate rewrite rules stack D, the context stack X, the phase stack P and program $\mathcal{R}$ We use notation $S[top]$ to denote the top element of stack S, and notation $S[top - i]$ to denote the i-th element counted from the top element of stack S. $T[top]$ on the subject term stack is called the current subject term. An element of the candidate rewrite rules stack D is a list of rewrite rules representing $\mathcal{R}_{top}(s)$ with $s = T[top]$ such that $\mathcal{R}_{mat}(s) \subseteq \mathcal{R}_{top}(s) \subseteq \mathcal{R}$, where $\mathcal{R}_{mat}(s) = \{e \rightarrow e' \;\; if \;\; t_1 = t'_1, \ldots, t_n = t'_n \; \in \; \mathcal{R} \mid n \geq 0, \; \exists \theta, \; s \equiv e\theta\}$. The first element of $D[top]$ is called a current candidate rewrite rule. The phase stack P is a stack of objects called phases. There are seven phases, which are *:getrules*, *:match*, *:move*, *:eq?*, *:eval*, *:replace* and *:halt*.

Computation in our system is defined by the reduction relation $\Rightarrow$ on the set of states. In our system, the reduc-

Function	Arguments	Returned value
first	list $(x_1 \ \ldots \ x_n)$	first element x_1
arity	function symbol $F \in \mathcal{F}$	arity $n(\geq 0)$
push	element t , stack S	stack S' resulted from pushing t on S
pop	stack S	stack S' resulted from removing the top element from S
remove	stack S	stack S' resulted from removing the first element from the list $S[top]$
getrules	term t, rewrite rules $\mathcal{R}$	list consisting of the elements of $\mathcal{R}_{top}$
lhs	a rewrite rule r	left-hand side of r
rhs	a rewrite rule r	right-hand side of r

Table 1 Primitive functions

tion is executed using the leftmost-outmost strategy. $\Rightarrow$ has the following twenty operation steps.

Before	Step	After
:getrules	Get-rules	*:match*
:match	No-matching	*:match*
	No-meta-matching	*:match*
	Matching-success(conditional)	*:getrules*
	Meta-matching-success(conditional)	*:getrules*
	Matching-success(unconditional)	*:replace*
	Meta-matching-success(unconditional)	*:replace*
	Matching-fail(compound term)	*:getrules*
	Matching-fail(constant or variable)	*:move*
:move	Move-context(right)	*:getrules*
	Move-context(up)	*:eq?*
	Base-transformation	P[top]
	Halt	*:halt*
:eq?	Judge-condition-success	*:eval*
	Judge-condition-fail	*:move*
:eval	Condition-evaluation-success	*:replace*
	Condition-evaluation-continuance	*:match*
	Condition-evaluation-fail	*:match*
:replace	Replace	*:getrules*
	Meta-replace	*:getrules*

We explain each operation step. For convenience, let $s = T[top]$, and the operation of pushing the empty list () on the candidate rewrite rules stack D is called initialization of D, and that of pushing □ on the context stack X is called initialization of X. Furthermore, we have a special Lisp-like syntax for representing lists.

```
() ≡ nil
(X . Y) ≡ cons[X,Y]
(X Y ... Z)
  ≡ (X .(Y .(... .(Z . nil)...)))
```

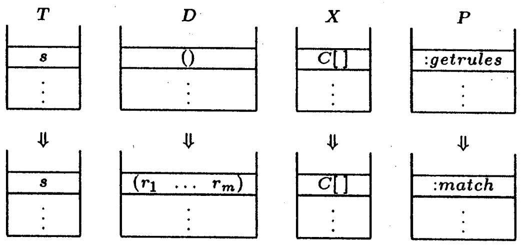
The Table 1 shows the primitive functions needed to explain the operation steps.

2.3.1 :Getrules Phase

Get-rules is the only operation step executable at the *:getrules* phase. In this step, the candidate rewrite rules stack D gets as $D[top]$ a list of the candidate rewrite rules $\mathcal{R}_{top}$ for the current subject term s, and the phase changes from *:getrules* to *:match*.

Get-rules:

$$\frac{P[top] = :getrules}{(T, D, X, P) \Rightarrow \quad (\ T,\ push(getrules(s,\ \mathcal{R}),\ pop(D)),\ X,\ push(:match,\ pop(P))\ )}$$



2.3.2 :Match Phase

There are eight operation steps on phase *:match*. In these eight steps, the current subject term s is matched to the left-hand side e of the current candidate rewrite rule $first(D[top]) = e \rightarrow e' \ \ if \ \ t_1 = t'_1, \ldots, t_n = t'_n \ \ (n \geq 0)$. If e is a meta-computational expression $t : z$, then the meta-matching procedure is invoked. It checks to see if there exists a substitution θ satisfying the following condition.

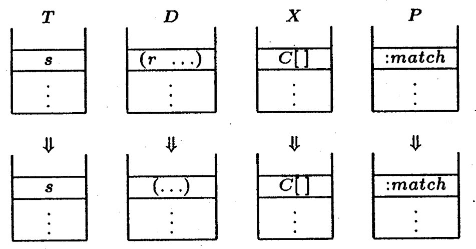
$$\left\{ \begin{array}{l} F[term^\wedge(s_1),\ \ldots,\ term^\wedge(s_i)] = t\theta, \\ \text{if R}[r] \in z, \text{ then } rules^\wedge(\mathcal{R}) = r\theta, \\ \text{if T}[a] \in z, \text{ then } Tstack^\wedge(T) = a\theta, \\ \text{if D}[d] \in z, \text{ then } Dstack^\wedge(D) = d\theta, \\ \text{if X}[x] \in z, \text{ then } Xstack^\wedge(X) = x\theta, \\ \text{if P}[p] \in z, \text{ then } Pstack^\wedge(P) = p\theta. \end{array} \right.$$

where $T[top] = F[s_1,\ \ldots,\ s_i]$, a meta-conditional rewrite rule $first(D[top]) = t : z \rightarrow e' \ \ if \ \ t_1 = t'_1, \ldots, t_n = t'_n \ \in \mathcal{R}$, and T, D, X, P are the primitive function symbols, $r, a, d, x, p \in \mathcal{T}$. We call the above condition the meta-matching condition.

No-matching :

$$\frac{D[top] = (r \ \ldots),\ \ P[top] = :match}{(T, D, X, P) \Rightarrow (\ \ T,\ \ remove(D),\ \ X,\ \ P\ \ )}$$

with $lhs(r) \in \mathcal{T}, \ \ \nexists\theta, \ s \equiv lhs(r)\theta.$

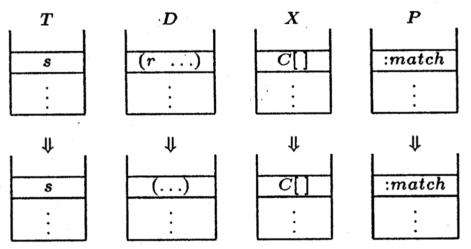


The no-matching step is executed when there exists no substitution θ satisfying $s \equiv e\theta$. In the no-matching step, the current candidate rewrite rule r is removed from $D[top] = (r\ \ldots)$.

No-meta-matching :

$$\frac{D[top] = (r\ \ldots),\ P[top] = :match}{(T, D, X, P) \Rightarrow (\ T,\ remove(D),\ X,\ P\ )}$$

with $lhs(r) \notin T,\ \nexists\theta$ *satisfying the mata-matching condition.*

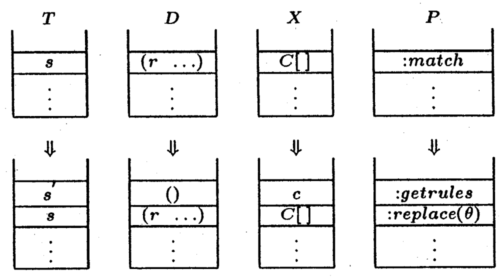


The no-meta-matching step is executed when there exists no substitution θ satisfying the meta-matching condition. In the no-meta-matching step, the current candidate rewrite rule r is removed from $D[top] = (r\ \ldots)$.

Matching-success(conditional) :

$$\frac{D[top] = (r\ \ldots),\ P[top] = :match}{(T, D, X, P) \Rightarrow (\ push(s',\ T),\ push((),\ D),\ push(c,\ X),\ push(:getrules,\ push(:replace(\theta),\ pop(P)))\ )}$$

with $n > 0,\ s' = eq[t_1\theta,\ t_1'\theta],\ s \equiv e\theta,$
$r = e \rightarrow e'\ if\ t_1 = t_1', \ldots, t_n = t_n',\ e \in T$
$c = (\Box\ eq[t_2\theta,\ t_2'\theta]\ \ldots\ eq[t_n\theta,\ t_n'\theta]).$

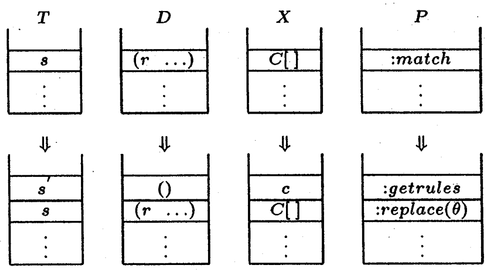


If the current candidate rewrite rule was successfully matched with a meta-conditional rewrite rule $e \rightarrow e'\ \ if\ \ t_1 = t_1', t_2 = t_2', \ldots, t_n = t_n'\ (n > 0)$, then the matching-success(conditional) step is executed. The condition $t_i\theta = t_i'\theta$ is encoded as the term $eq[t_i\theta,\ t_i'\theta]$. The first condition $eq[t_1\theta,\ t_1'\theta]$ will be pushed on the subject term stack T. The context which represents from second through n-th conditions are encoded as $(\Box\ \ eq[t_2\theta,\ t_2'\theta]\ \ \ldots\ \ eq[t_n\theta,\ t_n'\theta])$, and will be pushed on the context stack X. The stack D is initialized. After the phase *:match* is changed to the phase *:replace*(θ), the phase *:getrules* is pushed on P.

Meta-matching-success(conditional) :

$$\frac{D[top] = (r\ \ldots),\ P[top] = :match}{(T, D, X, P) \Rightarrow (\ push(s',\ T),\ push((),\ D),\ push(c,\ X),\ push(:getrules,\ push(:replace(\theta),\ pop(P)))\ )}$$

with $n > 0,\ s' = eq[t_1\theta,\ t_1'\theta],$
$r = t : z \rightarrow e'\ if\ t_1 = t_1', \ldots, t_n = t_n',$
$\exists\theta$ *satisfying the mata-matching condition,*
$c = (\Box\ eq[t_2\theta,\ t_2'\theta]\ \ldots\ eq[t_n\theta,\ t_n'\theta]).$

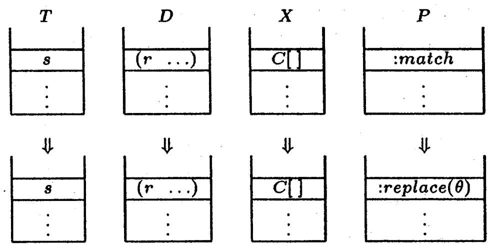


The meta-matching-success(conditional) step is executed when there exists a substitution θ satisfying the meta-matching condition and the current candidate rewrite rule is a meta-conditional rewrite rule. The operations on the stacks are the same as those of the matching-success(conditional) step.

Matching-success(unconditional) :

$$\frac{D[top] = (r\ \ldots),\ P[top] = :match}{(T, D, X, P) \Rightarrow (\ T,\ D,\ X,\ push(:replace(\theta),\ pop(P))\ )}$$

with $lhs(r) \in T,\ s \equiv lhs(r)\theta.$

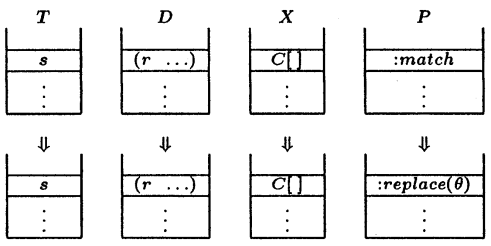


If the current candidate rewrite rule was successfully matched with an unconditional rewrite rule $e \rightarrow e'$, then the matching-success(unconditional) step is executed. The phase changes from *:match* to *:replace*(θ) with the substitution θ $(s \equiv e\theta)$.

Meta-matching-success(unconditonal) :

$$\frac{D[top] = (r\ \ldots),\ \ P[top] = :match}{(T, D, X, P) \Rightarrow (\ \ T,\ \ D,\ \ X,\ \ push(:replace(\theta),\ pop(P))\ \ )}$$

with $lhs(r) \notin T$, $\exists\theta$ *satisfying the mata-matching condition.*

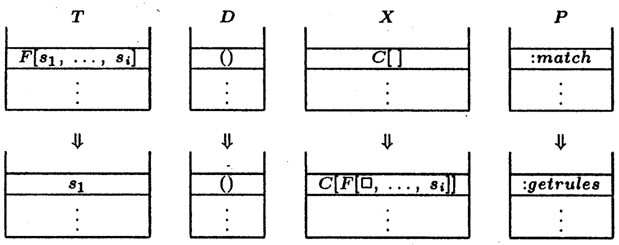


The meta-matching-success(unconditional) step is executed when there exists a substitution θ satisfing the meta-matching condition and the current candidate rewrite rule is an unconditional rewrite rule. The phase changes from *:match* to *:replace*(θ) with the substitution θ satisfying the meta-matching condition.

Matching-fail(compound term) :

$$\frac{s = F[s_1,\ \ldots,\ s_i],\ \ D[top] = (),\ \ P[top] = :match}{(T, D, X, P) \Rightarrow \quad (\ push(s_1,\ pop(T)),\ D,\ push(C[F[\Box,\ \ldots,\ s_i]],\ pop(X)),\ push(:getrules,\ pop(P))\ )}$$

with $i > 0$, $X[top] = C[\,]$.

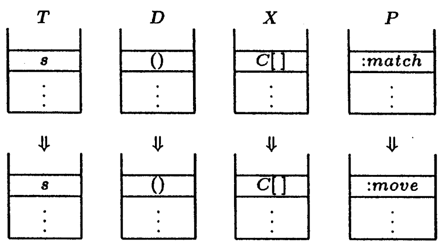


The matching-fail(compound term) step is executed when there are no candidate rewrite rules ($D[top] = ()$) and the current subject term is a compound term $F[s_1,\ \ldots,\ s_i]$ ($i > 0$). The first argument s_1 of the compound term will become the new current subject term, and the remainder parts of the compound term will be embeded in the context $C[F[\Box,\ \ldots,\ s_i]]$. The phase changes from *:match* to *:getrules.*

Matching-fail(constant or variable) :

$$\frac{D[top] = (),\ \ P[top] = :match}{(T, D, X, P) \Rightarrow (\ \ T,\ \ D,\ \ X,\ \ push(:move,\ pop(P))\ \ )}$$

with $s \in \mathcal{F}$, $arity(s) = 0$ *or* $s \in \mathcal{V}$.



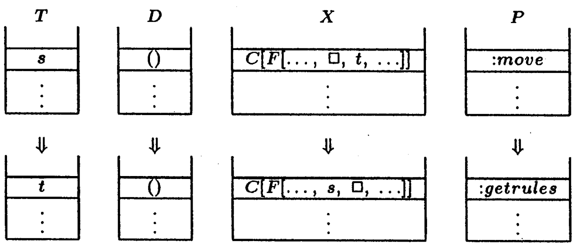
The matching-fail(constant or variable) step is executed when there are no candidate rewrite rules ($D[top] = ()$) and the current subject term is a constant or a variable. The phase *:match* is changed to *:move.*

2.3.3 :Move Phase

There are four operation steps on phase *:move.*

Move-context(right) :

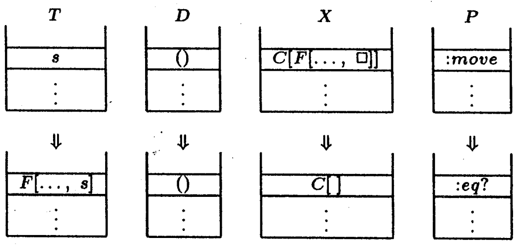
$$\frac{X[top] = C[F[\ldots,\ \Box,\ t,\ \ldots]],\ \ P[top] = :move}{(T, D, X, P) \Rightarrow \quad (\ push(t,\ pop(T)),\ D,\ push(C[F[\ldots,\ s,\ \Box, \ldots]],\ pop(X)),\ push(:getrules,\ pop(P))\ )}$$



The move-context(right) step is the operation that changes the current subject term s from the i-th to the $(i+1)$-th argument of the compound term. The phase changes from *:move* to *:getrules.*

Move-context(up) :

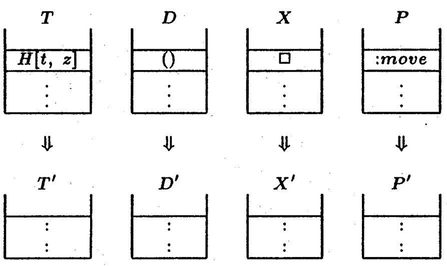
$$\frac{X[top] = C[F[\ldots,\ \Box]],\ \ P[top] = :move}{(T, D, X, P) \Rightarrow \quad (\ push(F[\ldots,\ s],\ pop(T)),\ D,\ push(C[\,],\ pop(X)),\ push(:eq?,\ pop(P))\ )}$$



If the context is $C[F[\ldots,\ \Box]]$ and the last arguments of the compound term is the current subject term, then the new current subject term will be $F[\ldots,\ s]$ and the new context will be $C[\,]$. This operation is called the move-context(up) step. The phase changes to :*eq*?.

Base-transformation :

$$\frac{T[top] = H[t, z],\ \ X[top] = \Box,\ \ P[top] = :move}{(T, D, X, P, \mathcal{R}) \Rightarrow (\ \ T',\ \ D',\ \ X',\ \ P',\ \ \mathcal{R}'\ \ )}$$



The base-transformation step is executed when the subject term is a normal form whose root symbol is the distinguished constructor H. It changes the stacks T, D, X, P and the program $\mathcal{R}$ to T', D', X', P', $\mathcal{R}'$ as follows:

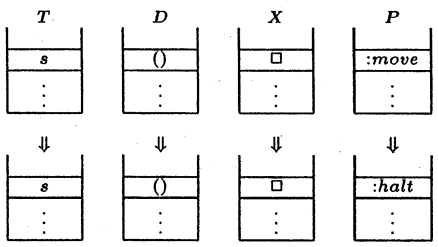
$$\left\{\begin{array}{ll} \mathcal{R}' : & \text{if (R } \ldots) \in z, \text{ then } rules^{\vee}((\text{R } \ldots)), \\ & \text{else } \mathcal{R}. \\ T' : & \text{if (T } \ldots) \in z, \text{ then } Tstack^{\vee}((\text{T } \ldots)), \\ & \text{else } T. \\ D' : & \text{if (D } \ldots) \in z, \text{ then } Dstack^{\vee}((\text{D } \ldots)), \\ & \text{else } D. \\ X' : & \text{if (X } \ldots) \in z, \text{ then } Xstack^{\vee}((\text{X } \ldots)), \\ & \text{else } X. \\ P' : & \text{if (P } \ldots) \in z, \text{ then } Pstack^{\vee}((\text{P } \ldots)), \\ & \text{else } P. \end{array}\right.$$

where R,T,D,X,P are the distinguished constructors. This is the only step that can change the program $\mathcal{R}$.

Halt :

$$\frac{X[top] = \Box,\ \ P[top] = :move}{(T, D, X, P) \Rightarrow (\ \ T,\ \ D,\ \ X,\ \ push(:halt,\ pop(P))\ \ )}$$

with $s \not\equiv H[\ldots]$.



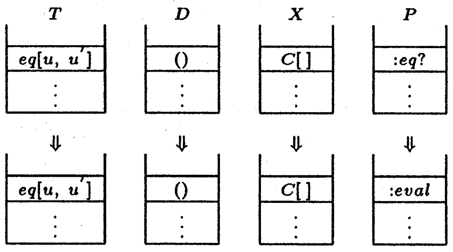
If there exists no candidate rewrite rules for the current subject term s and the context $X[top]$ is □, then the computation terminates. The current subject term s is the normal form. The phase :*move* is changed to :*halt*.

2.3.4 :Eq? Phase

There are two operation steps on phase :*eq*?, that is, the judge-condition-success step and the judge-condition-fail step. In this phase, it is checked whether or not the root function symbol of $T[top]$ is *eq*.

Judge-condition-success :

$$\frac{s = eq[u, u'],\ \ P[top] = :eq?}{(T, D, X, P) \Rightarrow (\ \ T,\ \ D,\ \ X,\ \ push(:eval,\ pop(P))\ \ )}$$

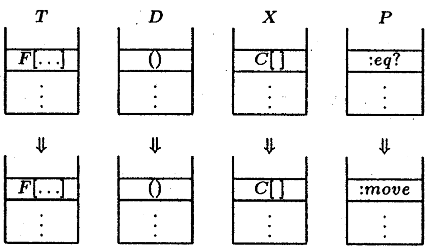


The judge-condition-success step is executed when the symbol is *eq*. The phase :*eq*? is changed to :*eval*.

Judge-condition-fail :

$$\frac{s = F[\ldots],\ \ P[top] = :eq?}{(T, D, X, P) \Rightarrow (\ \ T,\ \ D,\ \ X,\ \ push(:move,\ pop(P))\ \ )}$$

with $F \neq eq$.



In the judge-condition-fail step, the phase :*eq*? is changed to :*move*, it tries to move context again.

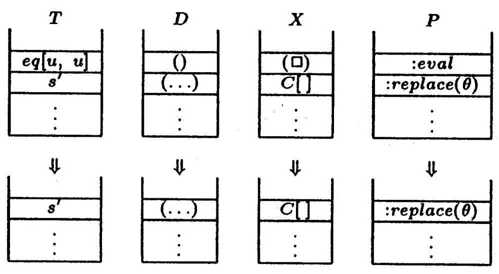
2.3.5 :Eval Phase

There are three operation steps on phase :*eval* for condition evaluation.

As we have mentioned before, in the matching-success(conditional) step, term $eq[t_i\theta,\ t_i'\theta]$ is pushed as $T[top]$. This term is rewritten $eq[u,\ u']$, such that $t_i\theta \rightarrow^*_{\mathcal{R}} u$, $t_i'\theta \rightarrow^*_{\mathcal{R}} u'$ (u, u' are the normal form). If $u \equiv u'$, then the term $eq[u,\ u']$ is removed from $T[top]$ and the term $eq[t_{i+1}\theta,\ t_{i+1}'\theta]$ taken from $X[top]$ will be the new current subject term. This operation is called the condition-evaluation-continuance step. If $X[top]$ is (□) and for this reason the next current subject term cannot be taken from the context, then the condition-evaluation-success step is executed. If $u \not\equiv u'$, then the condition-evaluation-fail step is executed.

Condition-evaluation-success :

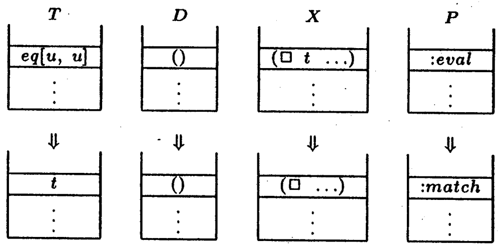
$$\frac{s = eq[u,\ u],\ \ X[top] = (\square),\ \ P[top] = :eval}{(T, D, X, P) \Rightarrow (\ \ pop(T),\ \ pop(D),\ \ pop(X),\ \ pop(P)\ \ )}$$



If all conditions of the meta-conditional rewrite rule evaluate to true, then the condition-evaluation-success step is executed. The top elements of the stacks are removed.

Condition-evaluation-continuance :

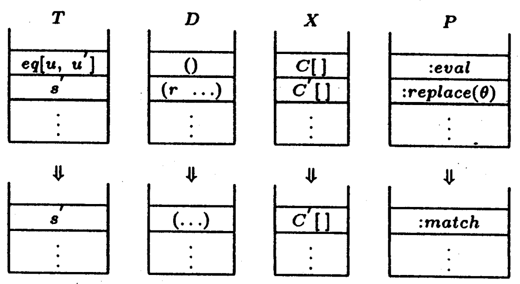
$$\frac{s = eq[u,\ u],\ \ X[top] = (\square\ \ t\ \ \ldots),\ \ P[top] = :eval}{\begin{array}{l}(T, D, X, P) \Rightarrow\ \ (\ \ push(t,\ pop(T)),\ \ D,\\ \qquad\qquad push((\square\ \ \ldots),\ pop(X)),\\ \qquad\qquad push(:match,\ pop(P))\ \ )\end{array}}$$



If the current subject term is $eq[u,\ u]$ and the unevaluated conditions exist, then the condition-evaluation-continuance step is executed. The current subject term $eq[u,\ u]$ is removed and replaced by the term t taken from the context $X[top] = C[(\square\ \ t\ \ \ldots)]$. The new $X[top]$ will be the context $C[(\square\ \ \ldots)]$. The phase *:eval* is changed to *:match*.

Condition-evaluation-fail :

$$\frac{s = eq[u,\ u'],\ \ u \not\equiv u',}{\begin{array}{l}(T, D, X, P) \Rightarrow\ \ (\ \ pop(T),\ \ remove(pop(D)),\\ \qquad\qquad pop(X),\ \ push(:match,\ pop(pop(P)))\ \ )\end{array}}$$



If u is not equal to u' in the current subject term $eq[u,\ u']$, then the condition-evaluation-fail step is executed. After the top elements of the stacks are removed, the first element r of the list $(r\ \ \ldots)$ will be removed. The phases $P[top]$ =*:eval* and $T[top-1]$ =*:replace*(θ) will be popped up, and the phase *:match* will be pushed on P.

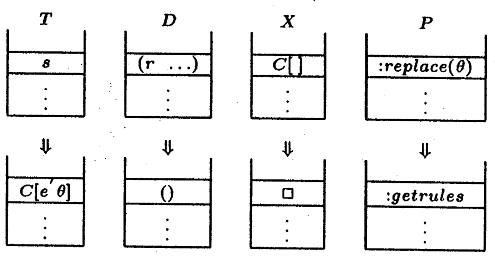
2.3.6 :Replace Phase

The replace step and the meta-replace step are executable at the *:replace* phase.

Replace :

$$\frac{D[top] = (r\ \ \ldots),\ \ P[top] = :replace(\theta)}{\begin{array}{l}(T, D, X, P) \Rightarrow\ \ (\ \ push(C[e'\theta],\ pop(T)),\ \ push((),\ pop(D)),\\ \qquad\qquad push(\square,\ pop(X)),\ \ push(:getrules,\ pop(P))\ \ )\end{array}}$$

with $rhs(r) = e' \in \mathcal{T},\ \ X[top] = C[\,]$.

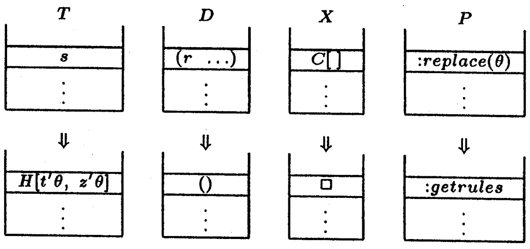


In the replace step, the current subject term will become $C[e'\theta]$, and the stacks D and X are initialized. The phase changes to *:getrules*.

Meta-replace :

$$\frac{D[top] = (r\ \ \ldots),\ \ P[top] = :replace(\theta)}{\begin{array}{l}(T, D, X, P) \Rightarrow\ \ (\ \ push(H[t'\theta,\ z'\theta],\ pop(T)),\\ \qquad\qquad push((),pop(D)),\ \ push(\square,\ pop(X)),\\ \qquad\qquad push(:getrules,\ pop(P))\ \ )\end{array}}$$

with $rhs(r) = t' : z' \notin \mathcal{T}$.



In the meta-replace step, the current subject term will become $H[t'\theta,\ z'\theta]$, and the stacks D and X are initialized. H is the primitive function symbol of arity 2.

3 Meta-representation

3.1 Syntax of the Implementation

In our implementation, we use the following syntax:

- A rewrite rule is written as $e = e'\ \ if\ \ t_1 = t'_1, \ldots, t_n = t'_n$, using the equality sign $=$ instead of the arrow $\rightarrow$.
- A variable name begins with a question mark `?`.
- Arguments of a term are surrounded by square brackets. The set $\mathcal{F}$ of function symbols are divided into operators $\mathcal{F}_D$ and constructors $\mathcal{F}_C$. When `F` is a constant operator, the corresponding term is written as `F[]`. When `F` is a constant constructor, the corresponding term is written as `F`.

3.2 Meta-representation of Terms

We assume the existence of a unique constructor constant `F'`$\in \mathcal{F}_C$ for each function symbol `F`$\in \mathcal{F}$. This makes it possible to represent function symbols as terms (in the meta-level). In particular, if `F` is a constructor constant, we define `F'`$\equiv$`F`. In our system, both `F` and `F'` are denoted by `F`.

The functions for meta-transformation and base-transformation are defined as follows:

$$\begin{array}{ll}
term^{\wedge}(\texttt{?x}) \equiv \texttt{var[x]}, & \text{when } \texttt{?x} \in \mathcal{V} \\
term^{\wedge}(\texttt{c}) \equiv \texttt{c}, & \text{when } \texttt{c} \in \mathcal{F}_C, arity(\texttt{c}) = 0 \\
term^{\wedge}(\texttt{F}[t_1, \ldots, t_n]) \equiv & (\texttt{F'}\ term^{\wedge}(t_1)\ \ldots\ term^{\wedge}(t_n)), \\
 & \text{when } arity(\texttt{F}) > 0 \text{ or} \\
 & \qquad arity(\texttt{F}) = 0 \wedge \texttt{F} \in \mathcal{F}_D
\end{array}$$

$$\begin{array}{ll}
term^{\vee}(\texttt{var[x]}) \equiv \texttt{?x}, & \text{when } \texttt{x} \in \mathcal{F}_C \\
term^{\vee}(\texttt{c}) \equiv \texttt{c}, & \text{when } \texttt{c} \in \mathcal{F}_C \\
term^{\vee}((\texttt{F'}\ t_1\ \ldots\ t_n)\ ) \equiv & \texttt{F}[term^{\vee}(t_1), \ldots, term^{\vee}(t_n)], \\
 & \text{when } \texttt{F'} \in \mathcal{F}_C
\end{array}$$

where `var` is the distinguished constructor for this purpose.

3.3 Meta-representation of Contexts

In our system, the meta-representation of contexts is defined as follows:

$$\begin{array}{l}
context^{\wedge}(\Box) \equiv \texttt{hole} \\
context^{\wedge}(C[\texttt{F}[t_1, \ldots, t_{i-1}, \Box, t_{i+1}, \ldots, t_n]]) \\
\quad \equiv \texttt{context[F'},\ (term^{\wedge}(t_{i-1})\ \ldots\ term^{\wedge}(t_1)), \\
\qquad\qquad (term^{\wedge}(t_{i+1})\ \ldots\ term^{\wedge}(t_n)), \\
\qquad\qquad context^{\wedge}(C[\,])]
\end{array}$$

where `hole` and `context` are the distinguished constructors for this purpose and `F'` is the meta-representation of `F`.

Furthermore, we give the following definition for base-transformation of contexts.

$$\begin{array}{l}
context^{\vee}(\texttt{hole}) = \texttt{hole} \\
context^{\vee}(\texttt{context}[F',\ (t_{i-1}\ \ldots\ t_1),\ (t_{i+1}\ \ldots\ t_n),\ uc]) \\
\quad = C[F[\ term^{\vee}(t_1), \ldots, term^{\vee}(t_{i-1}), \Box \\
\qquad\qquad term^{\vee}(t_{i+1}), \ldots, term^{\vee}(t_n)]]
\end{array}$$

where $C[\,] = context^{\vee}(uc)$.

3.4 Meta-representation of Programs

We define the meta-representation $rules^{\wedge}(R)$ of a program (a set of rewrite rules) $R = \{r_1, \ldots, r_n\}$ as follows:

$$\begin{array}{l}
rules^{\vee}(\texttt{nil}) = \{\,\} \\
rules^{\wedge}(\{r_1, \ldots, r_n\}) \equiv (rule^{\wedge}(r_1)\ \ldots\ rule^{\wedge}(r_n)) \\
rule^{\wedge}(e \rightarrow e'\ \ if\ \ t_1 = t'_1, \ldots, t_n = t'_n.) \\
\quad \equiv \texttt{rule}[expr^{\wedge}(e), expr^{\wedge}(e'), \\
\qquad\qquad condits^{\wedge}(\{t_1 = t'_1, \ldots, t_n = t'_n\})] \\
expr^{\wedge}(t) \equiv term^{\wedge}(t), \qquad \text{if } t \in \mathcal{T} \\
expr^{\wedge}(t : z) \equiv \texttt{meta-trans}[term^{\wedge}(t), term^{\wedge}(z)] \\
condits^{\vee}(\texttt{nil}) = \{\,\} \\
condits^{\wedge}(\{t_1 = t'_1, \ldots, t_n = t'_n\}) \\
\quad \equiv (condit^{\wedge}(t_1 = t'_1)\ \ldots\ condit^{\wedge}(t_n = t'_n)) \\
condit^{\wedge}(t = t') \equiv \texttt{condit}[term^{\wedge}(t), term^{\wedge}(t')]
\end{array}$$

where $rule^{\wedge}$, $expr^{\wedge}$, $condits^{\wedge}$, $condit^{\wedge}$ are auxiliary functions to get meta-representation of a rewrite rule, an expression, conditions and a condition, respectively, and `rule`, `meta-trans`, `condit` are the distinguished constructors.

We represent the inverse of the above functions as follows:

$$\begin{array}{l}
rules^{\vee}(\texttt{nil}) = \{\,\} \\
rules^{\vee}((r\ .\ rs)) = \{rule^{\vee}(r)\} \cup rules^{\vee}(rs) \\
rule^{\vee}(\texttt{rule}[e, e', \texttt{nil})]) = expr^{\vee}(e) \rightarrow expr^{\vee}(e') \\
rule^{\vee}(\texttt{rule}[e, e', (d\ .\ ds)]) \\
\ = expr^{\vee}(e) \rightarrow expr^{\vee}(e')\ \ if\ \ condits^{\vee}((d\ .\ ds)) \\
expr^{\vee}(t) = term^{\vee}(t) \\
\qquad \text{when } t \not\equiv \texttt{meta-trans}[\ldots, \ldots, \ldots] \\
expr^{\vee}(\texttt{meta-trans}[t, z]) = term^{\vee}(t) : term^{\vee}(z) \\
condits^{\vee}(\texttt{nil}) = \{\,\} \\
condits^{\vee}((d\ .\ ds)) = \{condit^{\vee}(d)\} \cup condits^{\vee}(ds) \\
condit^{\vee}(\texttt{condit}[t, t']) = \{term^{\vee}(t) = term^{\vee}(t')\}
\end{array}$$

3.5 Meta-representation of Stacks

In our system, the meta-representation of each stacks is defined as follows:

$$\begin{aligned}
&\mathit{Tstack}^\wedge(T) \equiv (\mathtt{T}\ \mathit{term}^\wedge(s_1)\ \ldots\ \mathit{term}^\wedge(s_n))\\
&\mathit{Dstack}^\wedge(D) \equiv (\mathtt{D}\ \mathit{rules}^\wedge(d_1)\ \ldots\ \mathit{rules}^\wedge(d_n))\\
&\mathit{Xstack}^\wedge(X) \equiv (\mathtt{X}\ \mathit{context}^\wedge(c_1)\ \ldots\ \mathit{context}^\wedge(c_n))\\
&\mathit{Pstack}^\wedge(P) \equiv (\mathtt{P}\ \mathit{term}^\wedge(p_1)\ \ldots\ \mathit{term}^\wedge(p_n))
\end{aligned}$$

where the depth of each stack is n, the elements of each stack are $T[top] = s_1, \ldots, T[top-(n-1)] = s_n$, $D[top] = d_1, \ldots, D[top-(n-1)] = d_n$, $X[top] = x_1, \ldots, X[top-(n-1)] = x_n$, $P[top] = p_1, \ldots, P[top-(n-1)] = p_n$, and T, D, X, P are the distinguished constructors.

We represent the inverse of the above functions as follows:

$$\begin{aligned}
\mathit{Tstack}^\vee((\mathtt{T}\ s'_1\ \ldots\ s'_n)) &= \mathit{push}(\mathit{term}^\vee(s'_1), \mathit{push}(\ldots,\\
&\quad \mathit{push}(\mathit{term}^\vee(s'_n), \mathit{empty\text{-}Tstack})\ldots))\\
\mathit{Dstack}^\vee((\mathtt{D}\ d'_1\ \ldots\ d'_n)) &= \mathit{push}(\mathit{rules}^\vee(d'_1), \mathit{push}(\ldots,\\
&\quad \mathit{push}(\mathit{rules}^\vee(d'_n), \mathit{empty\text{-}Dstack})\ldots))\\
\mathit{Xstack}^\vee((\mathtt{X}\ c'_1\ \ldots\ c'_n)) &= \mathit{push}(\mathit{context}^\vee(c'_1), \mathit{push}(\ldots,\\
&\quad \mathit{push}(\mathit{context}^\vee(c'_n), \mathit{empty\text{-}Xstack})\ldots))\\
\mathit{Pstack}^\vee((\mathtt{P}\ p'_1\ \ldots\ p'_n)) &= \mathit{push}(\mathit{term}^\vee(p'_1), \mathit{push}(\ldots,\\
&\quad \mathit{push}(\mathit{term}^\vee(p'_n), \mathit{empty\text{-}Pstack})\ldots))
\end{aligned}$$

where emptyTstack,etc., represent the empty stacks.

4 Applications

One can think of several applications of our system to the rewrite systems development.

For example, it can be applied to the development of the membership-conditional term rewriting systems[1][2] (abbreviated as MCTRS), in which each rewrite rule can have membership conditions which restrict the substitution values for the variables occurring in the rule. Since membership conditions need meta-level information which cannot be described in ordinary conditional rewrite rules, they have traditionally been described in ordinary programming languages such as Lisp. This has caused serious problems in portability and transparency. In our system, they can be described simply in meta-conditional rewrite rules.

Our system has a mechanism for adding and removing rewrite rules. One application of this mechanism is a simulation of augmented term rewriting systems[3] (abbreviated as ATRS). The major extension of augmented term rewriting systems was to allow a limited form of binding of a value to an atom. In ATRS, binding a value to an atom is equivalent to introducing a new rewrite rule to the system. For example, binding the value 5 to x is simulated by adding the rule `x=5` to the set of rewrite rules, where x is treated as an atom. In this way, our system can simply simulate ATRS by using the mechanism for adding and removing rewrite rules.

Another application is Martin's[4] approach to object-oriented rewriting. A typical problem that often occurs within algebraic specifications is like the following one:

- The terms $push(m,\ empty)$ and $push(n,\ push(m,\ empty))$ denote two states of a stack; How can one specify that they denote the states of one stack before and after the execution of the operation $push(n,\ \_)$?

Manipulation of an object is specified by replacing its defining equation $ob = term$ by a new one $ob = term'$, which defines the new state of the object. Given a stack of natural numbers, the stack-object $S1$ with initial state $push(0,\ empty)$ is specified by the following rewrite rule:

$$S1 = push(0,\ empty)$$

When application of the operation $push(2,\ S1)$ yields, it replaces the rewrite rule by a new rewrite rule:

$$S1 = push(2,\ push(0,\ empty))$$

This framework can simply be implemented by using the mechanism for adding and removing rewrite rules.

5 Efficient Implementation

To develop efficient implementation we have introduced the delayed evaluation mechanism into our system.

Meta-transformation transforms meta-level objects of terms, context and program into the corresponding base-level objects. The bigger the meta-level objects are, the longer time the transformation takes. Moreover, even after the meta-transformation has been performed, the transformation could be found useless, if the transformed data were not accessed at all. To avoid such useless transformation, we have implemented our system such that when meta-transformation is invoked, meta-level objects will not be transformed into base-level objects immediately. It is only when those meta-level objects are actually accessed that they are partially transformed into base-level objects according to the necessity.

The following executable examples illustrate the delayed evaluation mechanism in our system. Let > be the Lisp prompt and each command be performed sequentially. The function `term` is used to build terms. Let us build a term `f[a,b]` first.

```
> (setf x (term "f[a,b]"))
f[a,b]
> x
f[a,b]
```

The function root, args is used to get the function symbol and the arguments of the term.

```
> (root x)
f
> (args x)
(a b)
```

Note that (a b) is a Lisp object. A term is transformed into the corresponding meta-representation using the function term-wedge. If the delayed evaluation mechanism was not introduced, we would have the following result:

```
> (setf y (term-wedge x))
(f a b)
> y
(f a b)
```

Note that a term (f a b) is equivalent to a term cons[f,cons[a,cons[b,nil]]]. On the other hand, under the delayed evaluation mechanism, we have:

```
> (setf z (term-wedge x))
{f[a,b]}
> z
{f[a,b]}
```

The term enclosed by the parentheses {, } is a delayed term which encapsulates the untransformed meta-level form. Now suppose we would like to get f, the meta-representation of the head function symbol of f[a,b]. This is achieved by applying the function head to the meta-representation of f[a,b].

```
> y
(f a b)
> (head y)
f
> y
(f a b)

> z
{f[a,b]}
> (head z)
f
> z
(f . {(a b)})
```

Of course, a term (f a b) produced without the delayed evaluation never changes its form when it is accessed. However, observing the values of the variable z before and after the evaluation of (head z), we see that the term {f[a,b]} produced with the delayed evaluation has changed its form after it was accessed. These examples make it clear that when a part of the meta-level object, say f in our case, is accessed, it is partially transformed into a base-level object. But the argument (a b), which has not been accessed yet, is not transformed into meta-representation. Thus meta-level objects are partially transformed into meta-representation according to the necessity.

6 Conclusion

In this paper we have presented a framework of conditional term rewriting systems equipped with meta-computation supported by two transformation primitives called meta-transformation and base-transformation. We have discussed its syntax, the operational semantics, applications and efficient implementation.

Further research topics include the development of more powerful meta-computation facilities and their efficient implementation.

References

1) Toyama, Y.: Confluent term rewriting systems with membership conditions, *Proc. of Intern. Workshop on Conditional Term Rewriting Systems, Lecture Notes in Computer Science*, Vol.308, 1987, pp.228–241.

2) Yamada, J.: Confluence of terminating membership conditional TRS, *Proc. of Intern. Workshop on Conditional Term Rewriting Systems, Lecture Notes in Computer Science*, Vol.656, 1992, pp.378–392.

3) Leler, W.: Constraint Programming Languages: Their Specification and Generation, Addison-Wesley, 1988.

4) Martin G.: Towards object-oriented algebraic specifications, *Proc. Workshop on Specification of Abstract Data Types, Lecture Notes in Computer Science*, Vol.534, 1990, pp.98–116.

5) Kurihara, M. Sato, T. and Ohuchi, A.: Reflective Computation in Term Rewriting Systems. *Computer Software*, Vol.12, No.4, 1995, pp.3–14.