PAPER

A Higher-Order Knuth-Bendix Procedure and Its Applications

Keiichirou KUSAKARI[†a)], ***Member and*** **Yuki CHIBA**[††b)], ***Nonmember***

SUMMARY The completeness (i.e. confluent and terminating) property is an important concept when using a term rewriting system (TRS) as a computational model of functional programming languages. Knuth and Bendix have proposed a procedure known as the KB procedure for generating a complete TRS. A TRS cannot, however, directly handle higher-order functions that are widely used in functional programming languages. In this paper, we propose a higher-order KB procedure that extends the KB procedure to the framework of a simply-typed term rewriting system (STRS) as an extended TRS that can handle higher-order functions. We discuss the application of this higher-order KB procedure to a certification technique called inductionless induction used in program verification, and its application to fusion transformation, a typical kind of program transformation.

key words: *simply-typed term rewriting system, higher-order KB procedure, inductive theorem, inductionless induction, fusion transformation*

1. Introduction

A term rewriting system (TRS) can be used as a computational model of functional programming languages, in which the introduction of higher-order functions consisting of arguments and values achieves a high level of abstraction and increases the expressive power. A TRS, however, cannot directly handle higher-order functions, which makes it difficult to use accumulated results in the automatic verification of functional programs. Against this background, research on higher-order rewriting systems that can handle higher-order functions has been actively studied. This research, however, has come to place a priority on theoretical interests resulting in formalizations having excessive expressive power when viewed as a model of functional programming languages. With this in mind, we propose a simply-typed term rewriting system (STRS) as a higher-order rewriting system having sufficient expressive power for giving operational meaning to functional programming languages while still being easy to handle theoretically [16].

The completeness (i.e. confluent and terminating) property is an important concept when using a TRS as a computational model of functional programming languages. Knuth and Bendix have proposed a procedure known as the KB procedure for generating a complete TRS [14]. This KB procedure finds a complete TRS equivalent to the given set of first-order equations.

In this paper, we propose a higher-order KB procedure and demonstrate its validity. The STRS can be made to handle higher-order functions simply by slightly easing the restrictions on the data structure of first-order terms. As a consequence, many of the theoretical properties that hold for the first-order framework can be directly transported to a higher-order one. Indeed we can achieve a higher-order KB procedure. We also examine the application of this higher-order KB procedure that we have designed to inductionless induction and fusion transformation.

Most of the data structures used in functional programming are inductive structures such as list and tree structures. For this reason, most properties that a program must guarantee are formalized as inductive theorems, and as a result, a method for automatically proving inductive theorems is essential for establishing an automatic program verification method. Various methods for automatically proving inductive theorems have been proposed in the research of TRS as a computational model of functional programming languages [6], [7], [10], [15], [18], [19], [21]. In this regard, we present the application of a higher-order KB procedure to the results in [18], which extends the results of inductionless induction in [15], [21] to an STRS framework.

Programming using a functional programming language begins with the definition of basic functions that can then be combined to define more complex functions. Most basic functions are defined in the form of a data structure having an inductive structure such as a list or tree. When combining such functions, a large amount of intermediate data inherent in those data structures can be generated, and a program that generates such intermediate data is generally weak in terms of computational efficiency. It is therefore desirable that a program of this type be converted to one that does not generate such intermediate data so that program efficiency can be improved. This kind of program conversion is called a fusion transformation [8], [20], [22]. Bellegarde has proposed a fusion transformation based on the KB procedure [5]. This technique imposes the restriction that fused terms must be linear, and it does not directly use the KB procedure. Ito, Kusakari, and Toyama have shown that fusion transformation of a TRS can be performed even with direct use of the KB procedure, and have also shown by experiment that the linearity restriction on fused terms can be removed [11]. These results, however, pertain to a first-order TRS framework: they cannot handle higher-order functions. In the study presented here, we examine by ex-

Manuscript received December 17, 2004.
Manuscript revised June 19, 2006.

[†]The author is with the Graduate School of Information Science, Nagoya University, Nagoya-shi, 464–8603 Japan.
[††]The author is with Research Institute of Electrical Communication, Tohoku University, Sendai-shi, 980–8577 Japan.
a) E-mail: kusakari@is.nagoya-u.ac.jp
b) E-mail: chiba@nue.riec.tohoku.ac.jp

DOI: 10.1093/ietisy/e90–d.4.707

periment the fusion transformation of higher-order functions using a higher-order KB procedure.

2. Preliminaries

In this section, we introduce some notions for abstract reduction systems (ARSs), untyped term rewriting systems (UTRSs), simply-typed term rewriting systems (STRSs), and many-sorted term rewriting systems (MS-TRSs), needed later on. We assume that the reader is familiar with notions of term rewriting systems [3].

2.1 Abstract Reduction Systems

An *abstract reduction system* (ARS) R is a pair $\langle A, \rightarrow\rangle$ where A is a set and $\rightarrow$ is a binary relation on A. The transitive-reflexive closure of a binary relation $\rightarrow$ is denoted by $\overset{*}{\rightarrow}$, the transitive closure is denoted by $\overset{+}{\rightarrow}$, and the transitive-reflexive-symmetric closure is denoted by $\overset{*}{\leftrightarrow}$.

Let $R = \langle A, \rightarrow\rangle$ be an ARS. An element $a \in A$ is said to be a *normal form* if there exists no $b \in A$ such that $a \rightarrow b$. We denote all normal forms in R by $NF(R)$. An ARS R is said to be *weakly normalizing*, denoted by $WN(R)$, if $\forall a \in A.\ \exists b \in NF(R).\ a \overset{*}{\rightarrow} b$; to be *strongly normalizing* (*terminating*), denoted by $SN(R)$, if there exists no infinite sequence $a_0 \rightarrow a_1 \rightarrow \cdots$; to be *confluent*, denoted by $CR(R)$, if $a_1 \overset{*}{\leftarrow} a \overset{*}{\rightarrow} a_2 \Rightarrow \exists b \in A.\ a_1 \overset{*}{\rightarrow} b \overset{*}{\leftarrow} a_2$ for all $a, a_1, a_2 \in A$.

2.2 Untyped Term Rewriting Systems

Untyped term rewriting systems (UTRSs) introduced in [16][†] represent a basis for various rewriting systems: simply-typed term rewriting system, many-sorted term rewriting system, and traditional term rewriting system. In this subsection, we introduce some notions of UTRSs needed later on.

Let Σ be a *signature*, that is, a finite set of function symbols, which are denoted by $F, G, \ldots$. Let $\mathcal{V}$ be an enumerable set of variables with $\Sigma \cap \mathcal{V} = \emptyset$. Variables are denoted by $x, y, z, f, \ldots$. An *atom* is a function or variable symbol denoted by $a, a', \ldots$. The set $T(\Sigma, \mathcal{V})$ of (untyped) terms constructed from Σ and $\mathcal{V}$ is the smallest set such that $a(t_1, \ldots, t_n) \in T(\Sigma, \mathcal{V})$ whenever $a \in \Sigma \cup \mathcal{V}$ and $t_1, \ldots, t_n \in T(\Sigma, \mathcal{V})$. If $n = 0$, we write a instead of $a()$. Identity of terms is denoted by $\equiv$. For $s \equiv a(s_1, \ldots, s_n)$, we often write $s(t_1, \ldots, t_m)$ instead of $a(s_1, \ldots, s_n, t_1, \ldots, t_m)$. We define the *root symbol* by $root(a(t_1, \ldots, t_n)) = a$. $Var(t)$ is the set of variables in t. A term is said to be *closed* if no variable occurs in the term. The set of closed terms is denoted by $T(\Sigma)$. The *size* $|t|$ of t is the number of function symbols and variables in t.

A *substitution* θ is a mapping from variables to terms. Each substitution θ is naturally extended to a mapping from terms to terms, denoted by $\hat{\theta}$, as follows: $\hat{\theta}(F(t_1, \ldots, t_n)) = F(\hat{\theta}(t_1), \ldots, \hat{\theta}(t_n))$ if $F \in \Sigma$; $\hat{\theta}(z(t_1, \ldots, t_n)) = a'(t'_1, \ldots, t'_m, \theta(\hat{t}_1), \ldots, \hat{\theta}(t_n))$ if $z \in \mathcal{V}$ with $\theta(z) = a'(t'_1, \ldots, t'_m)$. For simplicity, we identify θ and $\hat{\theta}$. We write $t\theta$ instead of $\theta(t)$.

A *context* is a term which has exactly one special symbol $\square$, called hole, at a leaf position. A *suffix context* is a term which has the symbol $\square$ at the root position. For example, $F(0, \square)$ and $F(\square, xs)$ are contexts, $\square(0)$ and $\square(0, Nil)$ are suffix contexts, and $\square$ is a context and a suffix context. For a context $C[\,]$ (a suffix context $S[\,]$), $C[t]$ ($S[t]$) denotes the result of placing t in the hole of $C[\,]$ ($S[\,]$). For example, $C[t] \equiv a(t, t')$ for $C[\,] \equiv a(\square, t')$, and $S[a(t)] \equiv a(t, t')$ for $S[\,] \equiv \square(t')$. A term t' is said to be a *subterm* of a term t if there exists a context $C[\,]$ such that $t \equiv C[t']$. We denote by $Sub(t)$ all subterms of t. A term s is said to be an *instance* of a term t if there exists a substitution θ such that $s \equiv t\theta$.

A *rule* is a pair (l, r) of terms such that $Var(l) \supseteq Var(r)$. We write $l \rightarrow r$ instead of (l, r). For a set R of rules, the *reduction relation* $s \underset{R}{\rightarrow} t$ is defined as $s \equiv C[S[l\theta]]$ and $t \equiv C[S[r\theta]]$ for some $l \rightarrow r \in R$, $C[\,]$, $S[\,]$ and θ. We often omit the subscript $_R$ whenever no confusion arises. An *untyped term rewriting system* (UTRS) is an abstract reduction system $\langle T(\Sigma, \mathcal{V}), \underset{R}{\rightarrow}\rangle$, where R is a set of rules. We often denote an UTRS $\langle T(\Sigma, \mathcal{V}), \underset{R}{\rightarrow}\rangle$ by R. If t has a unique normal form in an UTRS R we denote it by $t\downarrow_R$. Note that if we don't use suffix contexts in the definition of the reduction relation, UTRSs are too restrictive to model of functional programming languages. For instance, the following UTRS is not confluent without suffix contexts.

$$\begin{cases} I(x) \rightarrow x \\ App(f, x) \rightarrow f(x) \end{cases}$$

In the system, we have

$$I(f, x) \leftarrow App(I(f), x) \rightarrow App(f, x) \rightarrow f(x).$$

However, to reduce $I(f, x)$ to $f(x)$ we need to apply the first rule in the suffix context $\square(x)$ which has the hole at a non-leaf position. Finally we give an example of UTRS which is a representation of the Map-function:

$$\begin{cases} Map(f, Nil) \rightarrow Nil \\ Map(f, x :: xs) \rightarrow f(x) :: Map(f, xs) \end{cases}$$

Note that we use the standard representation for list structures by symbols Nil and $Cons$, and abbreviate $Cons(x, xs)$ to $x :: xs$ throughout the paper. Then we have the following reduction sequence.

$$\begin{aligned} & Map(F,\ F(0) :: 0 :: Nil) \\ & \quad \underset{R}{\rightarrow} F(F(0)) :: Map(F,\ 0 :: Nil) \\ & \quad \underset{R}{\rightarrow} F(F(0)) :: F(0) :: Map(F,\ Nil) \\ & \quad \underset{R}{\rightarrow} F(F(0)) :: F(0) :: Nil \end{aligned}$$

[†]In [16], UTRSs were called term rewriting systems with higher-order variables (TRS-HVs). Since there exists no "higher-order variable" in untyped systems, we use UTRS in the paper.

2.3 Simply-Typed Term Rewriting Systems

We introduced simply-typed term rewriting systems (STRSs), which are defined as UTRSs with simply-type constraints [16].

A set of *basic types* (*sort*) is denoted by $\mathcal{B}$. The set $\mathcal{T}$ of *simple-types* is generated from $\mathcal{B}$ by the constructor $\to$ as $\mathcal{T} ::= \mathcal{B} \mid (\mathcal{T} \to \mathcal{T})$. To minimize the number of parentheses, we assume that $\to$ is right-associative, and omit redundant parentheses. A *type attachment* τ is a function from $\Sigma \cup \mathcal{V}$ to $\mathcal{T}$. A term $a(t_1, \ldots, t_n)$ has a type β if $\tau(a) = (\alpha_1 \to (\cdots \to (\alpha_n \to \beta)\cdots))$ and each t_i has the type α_i. A term t is said to be a *simply-typed term* if it has a simple-type. We denote all simply-typed terms by $T_\tau(\Sigma, \mathcal{V})$, and denote all simply-typed terms with a type α by $T_\alpha(\Sigma, \mathcal{V})$. A simply-typed term t is said to be *ground* if t is closed and of basic type. We denote all ground terms by $T_{\mathcal{B}}(\Sigma)$. We use V_h to stand for the set of higher-order variables (i.e. $V_h = \{x \in \mathcal{V} \mid \tau(x) \in \mathcal{T} \setminus \mathcal{B}\}$).

To keep the type consistency, we assume that $\tau(x) = \tau(\theta(x))$ for all $x \in \mathcal{V}$ and substitutions θ. We also prepare the hole $\square_\alpha$ with a simple type α, and for each context $C[\,]$ (suffix context $S[\,]$) with a hole $\square_\alpha$ we assume that $\tau(t) = \alpha$ whenever we denote $C[t]$ ($S[t]$). We define *order* of types by $ord(\alpha) = 1$ if $\alpha \in \mathcal{B}$; $ord(\alpha) = \max(1 + ord(\alpha_1), ord(\alpha_2))$ if $\alpha = \alpha_1 \to \alpha_2$. We notice that each t_i is of basic type in a simply-typed term $a(t_1, \ldots, t_n)$ whenever $ord(\tau(a)) \leq 2$.

A *simply-typed rule* is a rule $l \to r$ such that $\tau(l) = \tau(r)$. A *simply-typed term rewriting system* (STRS) is an abstract reduction system $\langle T_\tau(\Sigma, \mathcal{V}), \underset{R}{\to} \rangle$, where R is a set of simply-typed rules. We often denote an STRS $\langle T_\tau(\Sigma, \mathcal{V}), \underset{R}{\to} \rangle$ by R. For example, the *Map*-function is also represented by the STRS, which is the UTRS presented in the previous subsection with $\tau(Nil) = L$, $\tau(::) = N \to L \to L$ and $\tau(Map) = (N \to N) \to L \to L$. For an STRS $\langle T_\tau(\Sigma, \mathcal{V}), \underset{R}{\to} \rangle$, we define $GNF(R)$, $GWN(R)$, $GSN(R)$ and $GCR(R)$ as $NF(R')$, $WN(R')$, $SN(R')$ and $CR(R')$ in the ARS $R' = \langle T_{\mathcal{B}}(\Sigma), \underset{R}{\to} \rangle$, respectively. A *simply-typed equation* is a pair (e_1, e_2) of simply-typed terms $e_1, e_2 \in T_\tau(\Sigma, \mathcal{V})$ such that $\tau(e_1) = \tau(e_2)$. We write $e_1 = e_2$ instead of (e_1, e_2). For any set E of equations, $\underset{E}{\to}$ is defined as similar to reduction relation.

According to the traditional way, we partition off Σ into $\mathcal{D}$ and C, called by *defined symbols* and *constructors*, respectively. A simply-typed term $t \in T_\tau(C, \mathcal{V}_h)$ is said to be a *pseudo-value* if any variable occurrence is at a leaf position. We denote all pseudo-values by $PVal(C, \mathcal{V}_h)$. A STRS R is said to be *strongly quasi-reducible*, denoted by $SQR(R)$, if any basic-typed term $F(t_1, \ldots, t_n)$ is reducible by R whenever $t_1, \ldots, t_n \in PVal(C, \mathcal{V}_h)$ and $F \in \mathcal{D}$.

Simply-typed terms s and t are *unifiable* if there exists a substitution θ such that $s\theta \equiv t\theta$. Then θ is said to be a *unifier* of s and t. A unifier θ of s and t is said to be a *most general unifier* if for any unifier θ' of s and t there exists θ'' such that $\theta' = \theta'' \circ \theta$. Let $l_1 \to r_1$ and $l_2 \to r_2$ be simply-typed rules, and suppose these rules have no variables in common. If $l_2 \equiv C[l'_2]$, $l'_2 \notin \mathcal{V}$, and l'_2 and l_1 are unifiable with the most general unifier θ, then the pair $\langle C[r_1]\theta, r_2\theta \rangle$ is called a *critical pair*.

Finally we introduce a result for proving termination of STRSs.

Proposition 2.1 [16] Let R be an STRS. R is terminating iff there exists a reduction order $>$ such that $\forall l \to r \in R.\ l > r$. Here a *reduction order* is a well-founded order closed under contexts and substitutions.

Definition 2.2 [17] A *precedence* $\rhd$ is a strict partial order on Σ. For any simply-typed terms $s \equiv a(s_1, \ldots, s_n)$ and $t \equiv a'(t_1, \ldots, t_m)$, we define $s >_{lpo} t$ if $\tau(s)$ and $\tau(t)$ have the same type whenever all basic types are identified, and one of the following properties holds:

- $\tau(s) \in \mathcal{B}$, $a \rhd a'$ and for all j either $s >_{lpo} t_j$ or $\exists$i. $s_i \geq_{lpo} t_j$,
- $a = a'$, $[s_1, \ldots, s_n] >^{lex}_{lpo} [t_1, \ldots, t_m]$ and for all j either $s >_{lpo} t_j$ or $\exists$i. $s_i \geq_{lpo} t_j$,
- there exists k such that $\exists i.\ s_i \geq_{lpo} a'(t_1, \ldots, t_k)$ and $\forall j > k.\ \exists i_j.\ s_{i_j} \geq_{lpo} t_j$.

Here $\geq_{lpo}$ is defined as $>_{lpo} \cup \equiv$.

Proposition 2.3 [17] The lexicographic path order $>_{lpo}$ is a reduction order.

2.4 Many-Sorted Term Rewriting Systems

We introduced many-sorted term rewriting systems (MS-TRSs), which are defined as UTRSs with sort constraints.

A set of *sort* is denoted by $Sort$, and the set of non-empty sequences of sorts is denoted by $Sort^+$. We often denote $\alpha_1, \ldots, \alpha_n, \beta \in Sort^+$ by $\alpha_1 \times \cdots \times \alpha_n \to \beta$. A *sort attachment st* is a function from $\Sigma \cup \mathcal{V}$ to $Sort^+$ such that $st(x) \in Sort$ for any $x \in \mathcal{V}$. A term $a(t_1, \ldots, t_n)$ is of $\beta \in Sort^+$ if $st(a) = \alpha_1 \times \cdots \times \alpha_n \to \beta$ and each t_i is of α_i. A term $a(t_1, \ldots, t_n)$ is said to be a *sorted term* if it is of α for some $\alpha \in Sort^+$. We denote all sorted terms by $T_{st}(\Sigma, \mathcal{V})$. We restrict substitutions and contexts to sort preserving ones. A *many-sorted rule* is a rule $l \to r$ such that $st(l) = st(r)$. A *many-sorted term rewriting system* (MS-TRS) is an abstract reduction system $\langle T_{st}(\Sigma, \mathcal{V}), \underset{R}{\to} \rangle$, where R is a set of many-sorted rules. The notion of critical pair is defined as in STRSs. Note that usual first-order term rewriting systems correspond to MS-TRSs with $|Sort| = 1$.

3. Higher-Order Knuth-Bendix Procedure

In order to solve word problems in universal algebras, Knuth and Bendix proposed a completion procedure known as the KB procedure [14]. The KB procedure attempts to transform a finite set of equations into a complete TRS, which serves as a decision procedure for word problems. In this section, we propose a higher-order KB procedure based on the formulation in [4], and prove its soundness.

Procedure 3.1 (Higher-Order KB Procedure)

```
Input: A set $E$ of equations and a reduction order $>$.
Output: A complete STRS $R$ that is logically equivalent to $E$.
(1) $R := \emptyset$
(2) If $E = \emptyset$ then return $R$; otherwise repeat the following (3)–(9).
(3) Pick an equation $s = t$ (or $t = s$) from $E$ such that $s > t$; if none exists, terminate with failure.
(4) $R := \{l \to r{\downarrow_{R'}} \mid l \to r \in R\}$ where $R' = \{s \to t\} \cup R$
(5) Add to $E$ all critical pairs between $s \to t$ and each rule in $\{s \to t\} \cup R$.
(6) Remove all rules from $R$ whose left-hand side contains an instance of $s$.
(7) $R := R \cup \{s \to t\}$
(8) $E := \{e_1{\downarrow_R} = e_2{\downarrow_R} \mid e_1 = e_2 \in E\}$
(9) Remove any equation in $E$ whose reduced sides are identical.
```

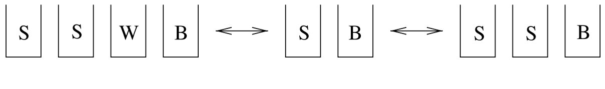
We note that for an STRS R, the completeness property is defined as $CR(R)$ and $SN(R)$, and logically equivalent to E means that $\overset{*}{\underset{R}{\leftrightarrow}} = \overset{*}{\underset{E}{\leftrightarrow}}$.

In the following, we use a glass-replacement puzzle to explain the solution to a word problem using a higher-order Knuth-Bendix procedure (HKB procedure).

Assume a certain sequence of sake, whisky, and beer glasses. This row of glasses may be replaced according to the following glass-replacement rules.

- A sake glass may be inserted to the left of a beer glass. Conversely, a sake glass having a beer glass to its right may be removed.
- A sake glass and whisky glass may be added to the left and right, respectively, of an appropriately selected contiguous glasses. The reverse operation may also be performed.

For example, the following changes are possible. In the following figure, sake, whisky, and beer glasses are indicated by S, W, and B, respectively.



We consider the following problems:

(1) Can the sake-sake-whisky-beer sequence be replaced by the sake-sake-beer-whisky sequence?
(2) Can the sake-whisky-whisky-whisky sequence be replaced by the sake-whisky-whisky-beer sequence?

The answer to problem (1) is "yes" and the answer to problem (2) is "no". The answer to (1) can be obtained by actually searching for a valid replacement procedure. In this case, the procedure can be found by a simple trial-and-error process. The difficulty arises when the answer is "no" as in problem (2). Because there is an infinite number of procedures for replacing these glasses, inspecting all of them would take forever.

This problem can be formalized as a typical word problem. We denote sake, whisky, and beer glasses as S, W, and B, respectively, and the type of these symbols as $S, W, B : * \to *$, where $*$ is a basic type. Now, if we introduce the function symbol $\perp : *$ to represent the end of the glass sequence, then the state of glasses arranged in the order of sake-sake-beer-whisky-beer can be expressed as $S(S(B(W(B(\perp)))))$. In this formalization, the above two replacement rules can be given by the following set of equalities denoted by E.

$$E = \left\{ \begin{array}{r} S(B(y)) = B(y) \\ S(x(W(y))) = x(y) \end{array} \right.$$

Here, x, y are variables. We note in particular that variable x used in the second rule has the function type $* \to *$, which means that it is a higher-order variable. The two problems described above can therefore be formalized by the following term-based word problem.

(1) $S(S(W(B(\perp)))) \overset{*}{\underset{E}{\leftrightarrow}} S(S(B(W(\perp))))$?
(2) $S(W(W(W(\perp)))) \overset{*}{\underset{E}{\leftrightarrow}} S(W(W(B(\perp))))$?

We attempt to solve these two word problems using an HKB procedure. If we use the lexicographic path order (Def. 2.2) as a reduction order (in this example, there is no need to give precedence), we obtain the following complete STRS denoted by R.

$$R = \left\{ \begin{array}{r} S(B(y))) \to B(y) \\ S(x(W(y))) \to x(y) \\ B(W(y))) \to B(y) \end{array} \right.$$

The top two rewrite rules here are obtained by giving direction to the set E of rules, and the last rule is obtained by giving direction to the added critical pair. If we now try to solve the above word problems using this STRS, we obtain the following results.

(1) $S(S(W(B(\perp)))){\downarrow} = B = S(S(B(W(\perp)))){\downarrow}$
(2) $S(W(W(W(\perp)))){\downarrow} = W(W(\perp)) \neq W(B(\perp)) = S(W(W(B(\perp)))){\downarrow}$

Accordingly, as the normal forms of the two glass sequences in problem (1) agree, the answer is "yes"; and as they do not agree in problem (2), the answer is "no". These results also indicate that the formalization denoted by $S(x(W(y))) = x(y)$ is not possible in a first-order TRS, which points to the difficulty of directly handling the appearance of arbitrary contiguous glasses in the glass sequence. Although some readers may think that the equality $S(x(W(y))) = x(y)$ can be represented in a first-order setting by substituting the closed terms of the type $* \to *$ for the higher-order variable x, this approach generates an infinite system because there exist infinite terms of the type $* \to *$. Using an STRS that can handle terms that include higher-order variables can suppress such problems.

The rest of this section demonstrates the validity of our HKB procedure, or more specifically, that the STRS R obtained as output is logically equivalent to the equality set E given as input, and complete as well. The proof is given by dropping into a first-order framework using currying [13].

Definition 3.2 For any simple types α and β, we prepare the special constant $@_{\alpha,\beta}$. We define $\Sigma^@ = \Sigma \cup \{@_{\alpha,\beta} \mid \alpha, \beta \in \mathcal{T}\}$. For any simple type α, we prepare the sort σ_α, and define $Sort_{\mathcal{T}} = \{\sigma_\alpha \mid \alpha \in \mathcal{T}\}$. For a type attachment τ, we define the sort attachment st_τ by $st_\tau(@_{\alpha,\beta}) = \sigma_{\alpha\to\beta} \times \sigma_\alpha \to \sigma_\beta$ and $st_\tau(a) = \sigma_{\tau(a)}$ for any $a \in \Sigma \cup \mathcal{V}$.

For any simply-typed term $t \in T_\tau(\Sigma, \mathcal{V})$, we inductively define the sorted term $t^@ \in T_{st_\tau}(\Sigma^@, \mathcal{V})$ as follows:

- $a^@ = a$ for any $a \in \Sigma \cup \mathcal{V}$
- $a(t_1, \dots, t_n)^@ = @_{\alpha,\beta}(a(t_1, \dots, t_{n-1})^@, t_n^@)$ if $n \geq 1$ and $\tau(a(t_1, \dots, t_{n-1})) = \alpha \to \beta$

We notice that $T_{st_\tau}(\Sigma^@, \mathcal{V}) = \{t^@ \mid t \in T_\tau(\Sigma, \mathcal{V})\}$. We naturally extend the notion over substitutions as $\theta^@(x) = (\theta(x))^@$, and over sets of pairs (like equations or rules) as $E^@ = \{(s^@, t^@) \mid (s, t) \in E\}$.

Lemma 3.3 The equality $C[S[t\theta]]^@ \equiv C^@[S^@[t^@\theta^@]]$ holds for any term t, context $C[\,]$, suffix context $S[\,]$ and substitution θ such that $C[S[t\theta]]$ has a simple type.

Proof. Firstly we prove $(t\theta)^@ \equiv t^@\theta^@$ by induction on $|t|$. Let $t \equiv a(t_1, \dots, t_n)$. The case $n = 0$ is trivial. Suppose that $n > 0$. Let $u \equiv a(t_1, \dots, t_{n-1})$ and $\tau(u) = \alpha \to \beta$. Then $(t\theta)^@ \equiv (u(t_n)\theta)^@ \equiv ((u\theta)(t_n\theta))^@ \equiv @_{\alpha,\beta}((u\theta)^@, (t_n\theta)^@) \equiv @_{\alpha,\beta}(u^@\theta^@, t_n^@\theta^@) \equiv @_{\alpha,\beta}(u^@, t_n^@)\theta^@ \equiv t^@\theta^@$.

We can also prove $S[t\theta]^@ \equiv S^@[(t\theta)^@]$ by induction on $S[\,]$, and $C[S[t\theta]]^@ \equiv C^@[S[t\theta]^@]$ by induction on $C[\,]$. Hence we obtain $C[S[t\theta]]^@ \equiv C^@[S^@[t^@\theta^@]]$. □

The following properties directly follows from this lemma.

Lemma 3.4

(i) For any STRS R, $s \underset{R}{\to} t \iff s^@ \underset{R^@}{\to} t^@$.

(ii) Let P be all critical pairs between $l_1 \to r_1$ and $l_2 \to r_2$, Q be all critical pairs between $l_1^@ \to r_1^@$ and $l_2^@ \to r_2^@$. Then $Q = P^@$.

(iii) A subterm of a simply-typed term s is an instance of a simply-typed term t if and only if a subterm of the sorted term $s^@$ is an instance of the sorted term $t^@$.

Theorem 3.5 Let E be a set of simply-typed equations and $>$ be a reduction order. If the HKB procedure applied to E and $>$ terminates successfully with output R, then R is a finite complete STRS that is logically equivalent to E.

Proof. The HKB procedure (Procedure 3.1) corresponds to the first-order KB procedure presented in [4]. We define $>'$ by $s^@ >' t^@ \iff s > t$. Then $>'$ is a reduction order on $T_{st}(\Sigma^@, \mathcal{V})$. Hence, thanks to Lemma 3.4 (ii, iii), the first-order KB procedure with sort constraints applied to $E^@$ and $>'$ terminates successfully with output $R^@$[†]. Thanks to Lemma 3.4 (i), STRS R is finite, complete and logically equivalent to E. □

This proof shows that the STRS can be viewed within the framework of a curried TRS, which is a first-order TRS with sort constraints. Nevertheless, there are three reasons for using STRS:

- First, for all terms in a curried TRS, all appearances at internal nodes take on $@_{\alpha,\beta}$, which only classify type information. This is an exceptionally strong restriction in the proof for termination. In particular, recursive path order and lexicographic path order, as well as the dependency pair method (strictly speaking, the argument filtering method) cannot be used in most cases (cf. [16], [17]).
- The second reason concerns the problem of execution efficiency. It can be seen from a simple calculation that $|t^@| = 2|t| - 1$. In other words, the curried term $t^@$ has almost twice the redundancy of term t. That is a fatal problem in implementation.
- The third reason relates to the problem of readability. Let's examine actual definitions for addition in both STRS and curried TRS (omitting the subscript of @).

$$\begin{cases} Add(x, 0) \to 0 \\ Add(x, S(y)) \to S(Add(x, y)) \end{cases}$$

$$\begin{cases} @(@(Add, x)), 0) \to 0 \\ @(@(Add, x), @(S, y)) \\ \qquad \to @(S, @(@(Add, x), y)) \end{cases}$$

Clearly STRS on the upper is more readable. While TRS is widely used in research of algebraic specifications, the specifications themselves are to be prepared by people regardless of how far research advances. This calls for a specific description language with high readability to avoid human errors. For this reason, STRS is superior to curried TRS as an algebraic specification language.

4. Inductionless Induction

The concept of inductive theorems is extremely important in practical applications. In actuality, most data structures used in functional programming are inductive structures such as list and tree structures. As a result, most properties that a program must guarantee are formalized as inductive theorems. For example, consider the following STRS R:

[†]The correctness of the KB procedure in [4] is proved on one-sorted TRSs (first-order TRSs). These proof still holds on many-sorted TRSs, because many-sorted TRSs has the subject property ($s \underset{R}{\to} t \Rightarrow st(s) = st(t)$).

$$\begin{cases} App(Nil, ys) \to ys \\ App(x :: xs, ys) \to x :: App(xs, ys) \\ Rev(Nil) \to Nil \\ Rev(x :: xs) \to App(Rev(xs), x :: Nil) \\ F(Nil, ys) \to ys \\ F(x :: xs, ys) \to F(xs, x :: ys) \\ Frev(xs) \to F(xs, Nil) \end{cases}$$

Both Rev and $Frev$ give a definition of a list-reverse function, but from the viewpoint of execution efficiency, $Frev$ results in a more efficient implementation. The transformation from Rev to $Frev$ is a typical example of improving program efficiency. The problem here, however, is that $Rev(xs) \overset{*}{\underset{R}{\leftrightarrow}} Frev(xs)$ does not hold in R. Why should this be the case despite the fact that $Rev([a_1, \ldots, a_n]) \overset{*}{\underset{R}{\leftrightarrow}} [a_n, \ldots, a_1] \overset{*}{\underset{R}{\leftrightarrow}} Frev([a_1, \ldots, a_n])$ for any list $[a_1, \ldots, a_n]$? The answer is that xs, which is given as input to $Rev, Frev$, is a variable, which prevents a specific list from being evaluated. In reality, the property that $Rev(t) \overset{*}{\underset{R}{\leftrightarrow}} Frev(t)$ is important for all input t that can be considered. This is the concept of inductive theorems. Furthermore, to enable an equality like $Rev = Frev$ to be handled in a higher-order framework, it is also necessary to incorporate the concept of extensionality in inductive theorems. We here introduce the definition of inductive theorems in STRSs given in [18].

Definition 4.1 [18] Let R be a set of equations. An equation $s = t$ is said to be a *primitive inductive theorem* in R, denoted by $R \vdash_{pind} s = t$, if $S_g[s\theta_c] \overset{*}{\underset{E}{\leftrightarrow}} S_g[t\theta_c]$ for all ground suffix context $S_g[\,]$ and closed substitution θ_c (i.e.$\forall x \in Var(s) \cup Var(t).\ x\theta_c \in T_\tau(\Sigma)$). We define $R \vdash^1_{pind} s = t$ by $R \vdash_{pind} s = t$; $R \vdash^{n+1}_{pind} s = t$ by $R' \vdash_{pind} s = t$ where $R' = \{u = v \mid R \vdash^n_{pind} u = v\}$. An equation $s = t$ is said to be an *inductive theorem* in R, denoted by $R \vdash_{ind} s = t$, if $R \vdash^n_{pind} s = t$ for some n. We also denote $R \vdash_{ind} E$ ($R \vdash_{pind} E$) if $R \vdash_{ind} s = t$ ($R \vdash_{pind} s = t$) for all $s = t \in E$.

In general, automated reasoning for inductive theorems is not so easy. To overcome the difficulty, the inductionless induction method, which provides a mechanical support for inductive theorems, was proposed by Musser [19], and extended by Huet and Hullot [10]. Toyama made clear an essence of the method [21]. Recently, Kusakari, Sakai and Sakabe extended the method to higher-order settings [18].

Proposition 4.2 [18] Let R and R' be STRSs, and E be a set of equations. If all of the following properties hold then $R \vdash_{pind} E$ and $R \vdash_{ind} E$.

(i) $\overset{*}{\underset{R\cup E}{\leftrightarrow}} \subseteq \overset{*}{\underset{R'}{\leftrightarrow}}$ in $T_{\mathcal{B}}(\Sigma)$
(ii) $GWN(R)$
(iii) $GCR(R')$
(iv) $GNF(R) \subseteq GNF(R')$

Example 4.3 Consider the following STRS R.

$$R = \begin{cases} Map(f, Nil) \to Nil \\ Map(f, x :: xs) \to f(x) :: Map(f, xs) \\ App(Nil, ys) \to ys \\ App(x :: xs, ys) \to x :: App(xs, ys) \end{cases}$$

We suppose that $\Sigma = \{0, S, Nil, ::, App, Map\}$, $\tau(0) = N$, $\tau(S) = N \to N$, $\tau(Nil) = L$, $\tau(::) = N \to L \to L$, $\tau(App) = L \to L \to L$ and $\tau(Map) = (N \to N) \to L \to L$.

Based on Proposition 4.2, we prove that the following equation is an inductive theorem.

$$Map(f, App(xs, ys)) = App(Map(f, xs), Map(f, ys))$$

Let R' be the union of R and the above equation. Then all conditions in Proposition 4.2 hold, and hence we succeed to prove that the equation is an inductive theorem in R.

The above example is relatively easy because the generated STRS R' is ground confluent. However, this is in general not always the case. The difficulty can be overcome by using the HKB procedure.

Example 4.4 Consider the following STRS R.

$$R = \begin{cases} Add(0, y) \to y \\ Add(S(x), y) \to S(Add(x, y)) \\ One(0) \to S(0) \\ One(S(x)) \to S(0) \\ Len(Nil) \to 0 \\ Len(x :: xs) \to S(Len(xs)) \\ Sum(Nil) \to 0 \\ Sum(x :: xs) \to Add(x, Sum(xs)) \\ Map(f, Nil) \to Nil \\ Map(f, x :: xs) \to f(x) :: Map(f, xs) \end{cases}$$

We suppose that $\Sigma = \{0, S, Nil, ::, Add, One, Len, Sum, Map\}$, $\tau(0) = N$, $\tau(S) = N \to N$, $\tau(Nil) = L$, $\tau(::) = N \to L \to L$, $\tau(Add) = N \to N \to N$, $\tau(One) = N \to N$, $\tau(Len) = L \to N$, $\tau(Sum) = L \to N$ and $\tau(Map) = (N \to N) \to L \to L$.

Based on Proposition 4.2, we prove that the following equation is an inductive theorem.

$$Sum(Map(One, xs)) = Len(xs)$$

Firstly, as similar to Example 4.3, let R' be the union of R and the above equation. However, R' is not confluent. In fact, we have:

$$\begin{aligned} Sum(Map(One, x :: xs)) &\to Len(x :: xs) \\ &\to S(Len(xs)) \end{aligned}$$

$$\begin{aligned} &Sum(Map(One, x :: xs)) \\ &\quad \to Sum(One(x) :: Map(One, xs)) \\ &\quad \to Add(One(x), Sum(Map(One, xs))) \\ &\quad \to Add(One(x), Len(xs)) \end{aligned}$$

Since $S(Len(xs))$ and $Add(One(x), Len(xs))$ are distinct normal forms, STRS R' is not confluent.

By the HKB procedure, we can overcome the problem. Indeed, we succeed to transform $R' \cup \{Add(x,0) \to x\}$ to a complete STRS. We note that we prove the lemma $Add(x,0) = x$ together with $Sum(Map(One,xs)) = Len(xs)$ because our HKB implementation could not transform R' to a complete STRS. As a reduction order, we use the lexicographic path order (Definition 2.2) with the precedence $Map \rhd Len \rhd Sum \rhd Add \rhd One \rhd :: \rhd Nil \rhd S \rhd 0$. Then the HKB procedure returns the following STRS R'':

$$\left\{\begin{array}{rcl} Add(0,y) & \to & y \\ Add(x,0) & \to & x \\ Add(S(x),y) & \to & S(Add(x,y)) \\ One(x) & \to & S(0) \\ Len(Nil) & \to & 0 \\ Len(x::xs) & \to & S(Len(xs)) \\ Sum(Nil) & \to & 0 \\ Sum(x::xs) & \to & Add(x,Sum(xs)) \\ Map(f,Nil) & \to & Nil \\ Map(f,x::xs) & \to & f(x)::Map(f,xs) \\ Sum(Map(One,xs)) & \to & Len(xs) \end{array}\right.$$

From Theorem 3.5, R'' is confluent, hence $GCR(R'')$ holds. The properties $\overset{*}{\underset{R\cup E}{\leftrightarrow}} \subseteq \overset{*}{\underset{R''}{\leftrightarrow}}$ in $T_{\mathcal{B}}(\Sigma)$, $GWN(R)$ and $GNF(R) \subseteq GNF(R'')$ can be showed as similar to Example 4.3. Therefore we have:

$$R \vdash_{ind} Sum(Map(One,xs)) = Len(xs)$$

Proposition 4.5 [18] Let R and R' be STRSs, and E be a set of equations. Suppose that all of the following four conditions hold:

(i) $\underset{R}{\to} \subseteq \overset{+}{\underset{R'}{\to}} \wedge \overset{*}{\underset{R\cup E}{\leftrightarrow}} = \overset{*}{\underset{R'}{\leftrightarrow}}$ in $T_{\mathcal{B}}(\Sigma)$
(ii) $GSN(R')$
(iii) $GCR(R)$
(iv) $GNF(R) \not\subseteq GNF(R')$

Then we have $R \nvdash_{pind} E$. Moreover if all of the following three conditions additionally hold then we also have $R \nvdash_{ind} E$.

(v) $SQR(R)$
(vi) $ord(\tau(C)) \leq 2$ for any $C \in \mathcal{C}$
(vii) $l \notin T_{\tau}(\mathcal{C}, V)$ for any $l \to r \in R$

The HKB procedure is also useful for disproving inductive theorems.

Example 4.6 Consider the following STRS R.

$$\left\{\begin{array}{rcl} Add(0,y) & \to & y \\ Add(S(x),y) & \to & S(Add(x,y)) \\ One(0) & \to & 0 \\ One(S(x)) & \to & S(0) \\ Len(Nil) & \to & 0 \\ Len(x::xs) & \to & S(Len(xs)) \\ Sum(Nil) & \to & 0 \\ Sum(x::xs) & \to & Add(x,Sum(xs)) \\ Map(f,Nil) & \to & Nil \\ Map(f,x::xs) & \to & f(x)::Map(f,xs) \\ Sum(Map(One,xs)) & \to & Len(xs) \end{array}\right.$$

This R is obtained by the STRS in Example 4.4: we change $One(0) \to S(0)$ into $One(0) \to 0$.

Based on Proposition 4.5, we prove that

$$Sum(Map(One,xs)) = Len(xs)$$

is not an inductive theorem. As similar to Example 4.3, let R' be the union of R and the above equation. Unfortunately, $GNF(R) \subseteq GNF(R')$ holds, that is, the condition (iv) does not hold. Hence we transform R' to a complete STRS as similar to Example 4.4. Then the HKB procedure returns the following STRS R'':

$$\left\{\begin{array}{rcl} Add(x,0) & \to & x \\ Add(0,y) & \to & y \\ One(x) & \to & 0 \\ Len(Nil) & \to & 0 \\ Len(x::xs) & \to & Len(xs) \\ Sum(Nil) & \to & 0 \\ Sum(x::xs) & \to & Add(x,Sum(xs)) \\ Map(f,Nil) & \to & Nil \\ Map(f,x::xs) & \to & f(x)::Map(f,xs) \\ Sum(Map(One,xs)) & \to & Len(xs) \\ S(y) & \to & y \end{array}\right.$$

Since $S(0) \in GNF(R)$ and $S(0) \notin GNF(R')$, the condition (iv) holds. All remaining conditions can be proved as similar to Example 4.4. Therefore we have:

$$R \nvdash_{ind} Sum(Map(One,xs)) = Len(xs)$$

5. Fusion Transformation

Programming using a functional programming language begins with the definition of basic functions that can then be combined to define more complex functions. Most basic functions are defined in the form of a data structure having an inductive structure such as a list or tree. When combining such functions, a large amount of intermediate data inherent in those data structures can be generated, and a program that generates such intermediate data is generally weak in terms of computational efficiency. It is therefore desirable that a program of this type be converted to one that does not generate such intermediate data so that program efficiency can be improved. This kind of program conversion is called a fusion transformation [8], [20], [22].

Bellegarde has proposed a fusion transformation based on the KB procedure [5]. This technique imposes the restriction that fused terms must be linear, and it does not directly use the KB procedure. Ito, Kusakari and Toyama have shown that fusion transformation of a TRS can be performed even with direct use of the KB procedure, and have also shown by experiment that the linearity restriction on fused terms can be removed [11]. These results, however, pertain to a first-order TRS framework: they cannot handle higher-order functions. In this section, we examine by experiment the fusion transformation of higher-order functions using the HKB procedure.

Consider a function $SqSum$ satisfying the following specification:

$$SqSum\ [a_1,\ a_2,\ \ldots,\ a_n] = {a_1}^2 + {a_2}^2 + \cdots + {a_n}^2$$

Let R be the following STRS:

$$R = \left\{ \begin{array}{r} Add(0, y) \to y \\ Add(S(x), y) \to S(Add(x, y)) \\ Mul(0, y) \to 0 \\ Mul(S(x), y) \to Add(Mul(x, y), y) \\ Sq(x) \to Mul(x, x) \\ Sum(Nil) \to 0 \\ Sum(x :: xs) \to Add(x, Sum(xs)) \\ Map(f, Nil) \to Nil \\ Map(f, x :: xs) \to f(x) :: Map(f, xs) \end{array} \right.$$

By using this STRS, we also give the specification for $SqSum$ as follows:

$$SqSum(xs) = Sum(Map(Sq, xs))$$

Here we use the HKB procedure. Its inputs are $R \cup \{SqSum(xs) = Sum(Map(Sq, xs))\}$ and the lexicographic path order (Definition 2.2) with $Map \rhd SqSum \rhd Sum \rhd Sq \rhd Mul \rhd Add \rhd\ ::\ \rhd Nil \rhd S \rhd 0$. Then the HKB procedure returns the union of R and the following three rules:

$$\begin{array}{r} Sum(Map(Sq, xs)) \to SqSum(xs) \\ SqSum(Nil) \to 0 \\ SqSum(x :: xs) \to Add(Mul(x, x), SqSum(xs)) \end{array}$$

Then two back rules give the definition of $SqSum$:

$$\left\{ \begin{array}{l} SqSum(Nil) \to 0 \\ SqSum(x, xs) \to Add(Mul(x, x), SqSum(xs)) \end{array} \right.$$

This definition is more efficient than the input $SqSum(xs) = Sum(Map(Sq, xs))$. Actually, this definition can be obtained only by carrying out scan once, although scan of list xs is carried out twice in the definition at the time of an input. Thus, fusion transformation can be performed using the HKB procedure.

6. Concluding Remarks

The higher-order KB procedure proposed in this study has already been implemented and subjected to various experiments. Since the achievement of a higher-order KB procedure is presently not known, we feel that this study will make a significant contribution to the field. This higher-order KB procedure is currently being applied directly to inductionless induction. When applying the KB procedure to inductionless induction, obtaining complete output for only ground terms is sufficient, and as a result, various improvements are being made in advanced research within the first-order TRS framework. In particular, the technique called linear strategy proposed by Fribourg is especially effective in suppressing the generation of critical pairs [9]. The introduction of results such as these will be taken up as a future research theme. The application of our higher-order KB procedure to fusion transformation is currently being performed by experiment only, but good results are being achieved. Its theoretical analysis must also be pursued as a future research theme. Of considerable interest here is Example 4.4, which presents an application of this higher-order KB procedure to inductionless induction. In this example, the rule $\{One(0) \to S(0), One(S(x)) \to S(0)\}$ is optimized as $\{One(x) \to S(0)\}$ by the higher-order KB procedure. The use of the procedure should be analyzed within a much larger framework than simply fusion transformation.

A higher-order KB procedure and its application to inductionless induction were implemented by the first author Kusakari, as a post-doctorate sub-theme at the Japan Advanced Institute of Science and Technology (JAIST) in 1999. Example 4.3, which presents an application to inductionless induction (Proposition 4.2), was first presented at that time. Results similar to these with respect to inductionless induction were also presented in 2003 by Aoto, Yamada, and Toyama [1]. Unfortunately, all of the above results confused the difference between inductive theorems and primitive inductive theorems, and erroneously used the term "inductive theorems". In [18], Kusakari, Sakai, and Sakabe made a clear distinction between the concepts of inductive theorems and primitive inductive ones. They showed that inductive theorems correspond to initial extensional algebra semantics and presented a sufficient condition for inductive theorems to agree with primitive inductive theory. At the request of Toyama, Kusakari presented a lecture on the results in [18] to Aoto and Yamada in August 2003. In the following year, Aoto, Yamada, and Toyama presented results on the automated proving of inductive theorems [2] using a formalization different from that of [18]. Furthermore, in 2002, the second author Chiba re-implemented the higher-order KB procedure in graduate work applying it to the glass-replacement puzzle and fusion transformation.

Acknowledgement

We would like to thank Associate Professor WATANABE Takuo and Professor SAKAI Masahiko for their help on the theory and implementation of the KB procedure and to Professor TOYAMA Yoshihito for various kinds of advice on inductionless induction. In regards to applications to fusion transformation, we would like to again thank Professor Toyama as well as ITOH Yoshihiro for their helpful discussions on this topic. Finally we would like to thank the anonymous referees for their useful comments.

This work is partly supported by MEXT. KAKENHI #15500007 and #16300005, by Artificial Intelligence Research Promotion Foundation, and by the 21st Century COE Program (Intelligent Media Integration for Social Infrastructure), Nagoya University.

References

[1] T. Aoto, T. Yamada, and Y. Toyama, "Proving inductive theorems of higher-order functional programs," Proc. Forum on Information Technology 2003 (FIT2003), Information Technology Letters, vol.2, pp.21–22, 2003.
[2] T. Aoto, T. Yamada, and Y. Toyama, "Inductive theorems for higher-order rewriting," Proc. 15th Int. Conf. on Rewriting Techniques and Applications (RTA 2004), LNCS 3091, pp.269–284, 2004.
[3] F. Baader and T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.
[4] L. Bachmair and N. Dershowitz, "Equational inference, canonical proofs, and proof orderings," J. ACM, vol.41, no.2, pp.236–276, 1994.
[5] F. Bellegarde, "Automating synthesis by completion," Journees Francophones des Langages Applicatifs, pp.177–203, 1995.
[6] A. Bouhoula, "Automated theorem proving by test set induction," J. Symbolic Computation, vol.23, pp.47–77, 1997.
[7] R.S. Boyer and J.S. Moore, A Computational Logic, Academic Press, NewYork, 1979.
[8] W.-N. Chin, "Safe fusion of functional expressions II: Further improvements," J. Functional Programming, vol.4, no.4, pp.515–555, 1994.
[9] L. Fribourg, "A strong restriction of the inductive completion procedure," J. Symbolic Computation, vol.8, pp.253–276, 1989.
[10] G. Huet and J.M. Hullot, "Proof by induction in equational theories with constructors," J. Comput. Syst. Sci., vol.25, pp.239–266, 1982.
[11] Y. Itoh, K. Kusakari, and Y. Toyama, "On the termination of program fusion based on completion procedure," IEICE Technical Report, COMP2002-84, March 2003.
[12] D. Kapur, P. Narendran, and H. Zhang, "Proof by induction using test sets," Proc. 8th Int. Conf. on Automated Deduction (CADE-8), LNCS 230, pp.99–117, 1986.
[13] R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries, "Comparing curried and uncurried rewriting," J. Symbolic Computation, vol.21, no.1, pp.15–39, 1996.
[14] D.E. Knuth and P.B. Bendix, "Simple word problems in universal algebra," in ed. J. Leech, Computational Problems in Abstract Algebra, pp.263–297, Pergamon Press, Oxford, U.K., 1970.
[15] H. Koike and Y. Toyama, "Comparison between inductionless induction and rewriting induction," JSSST Computer Software, vol.17, no.6, pp.1–12, 2000.
[16] K. Kusakari, "On proving termination of term rewriting systems with higher-order variables," IPSJ Trans. Programming, vol.42, no.SIG 7 (PRO 11), pp.35–45, 2001.
[17] K. Kusakari, "Higher-order path orders based on computability," IEICE Trans. Inf. & Syst., vol.E87-D, no.2, pp.352–359, Feb. 2004.
[18] K. Kusakari, M. Sakai, and T. Sakabe, "Primitive inductive theorems bridge implicit induction methods and inductive theorems in higher-order rewriting," IEICE Trans. Inf. & Syst., vol.E88-D, no.12, pp.2715–2726, Dec. 2005.
[19] D.R. Musser, "On proving induction properties of abstract data types," Proc. 7th ACM Symp. Principles of Programming Languages, pp.154–162, 1980.
[20] H. Seidl and M.H. Sørensen, Constraints to Stop Deforestation, Science of Computer Programming 32(1-3), pp.73–107, 1998.
[21] Y. Toyama, "How to prove equivalence of term rewriting systems without induction," Theor. Comput. Sci., vol.90, no.2, pp.369–390, 1991.
[22] P. Wadler, "Deforestation: Transforming programs to eliminate trees," Theor. Comput. Sci., vol.73, pp.231–248, 1990.

Keiichirou Kusakari received B.E. from Tokyo Institute of Technology in 1994, received M.E. and the Ph.D. degrees from Japan Advanced Institute of Science and Technology in 1996 and 2000. From 2000, he was a research associate at Tohoku University. He transferred to Nagoya University's Graduate School of Information Science in 2003 as an assistant professor and became an associate professor in 2006. His research interests include term rewriting systems, program theory, and automated theorem proving. He is a member of IPSJ and JSSST.

Yuki Chiba received his B.E. and M.E. degrees from Tohoku University in 2003 and 2005. He is currently a doctoral student of Research Institute of Electrical Communication, Tohoku University. His research interests include term rewriting systems.