We propose a framework in this paper for transforming programs with templates based on term rewriting. The programs are given by term rewriting systems. We discuss how to validate the correctness of program transformation within our framework. We introduce a notion of developed templates and a simple method of constructing such templates without explicit use of induction. We then show that in any transformation of programs using the developed templates, their correctness can be verified automatically. The correctness of program transformation within our framework is discussed based on operational semantics. We also present some examples of program transformations in our framework.

1. Introduction

Automatically transforming given programs to optimize efficiency is one of the most fascinating techniques for programming languages\[^{12,13}\]. Several techniques for transforming functional programming languages have been developed\[^{2,11,19}\]. Huet and Lang\[^{11}\] presented a framework of automated program transformation in which programs are transformed according to a given program transformation template, where the template consists of program schemas for input and output programs, and a set of equations the input (and output) programs must validate to guarantee the correctness of transformation. The programs and program schemas in their framework are given by second-order simply-typed lambda terms. They gave a second-order matching algorithm to verify whether a template could be applied to an input program. They also showed how to validate the correctness of program transformation using the denotational semantics.

After Huet and Lang’s pioneering work, Curien et al.\[^{7}\] provided an improved matching algorithm using top-down matching method. Yokoyama et al.\[^{20}\] presented sufficient conditions to have at most one solution and a deterministic algorithm to find such a solution. Recently, de Moor and Sittampalam\[^{9}\] presented a matching algorithm that could also be applied to third-order matching problems. The programs in all of these algorithms are represented by lambda terms and higher-order substitutions are achieved by the $\beta$-reduction of lambda calculus. However, in contrast to this successive work on matching algorithms, the formal verification component of the correctness of transformation has been neglected within the framework of program transformation using templates. Thus, the verification of the correctness of transformation in this framework still depends on Huet and Lang’s original technique based on denotational semantics.

We propose a framework of program transformation using templates in this paper based on term rewriting. In contrast to the existing work previously mentioned, programs and program schemas are given by term rewriting systems (TRSs for short) and TRS patterns. Thus, there is little difficulty in discussing the correctness of transformation based on operational semantics.

We introduce the notion of term homomorphisms to describe how a TRS pattern matches a concrete TRS. A key part of our procedure of TRS transformation using templates—the TRS pattern matching problem—is solved using the term pattern matching algorithm\[^{4}\].

To guarantee the correctness of transformation, we also introduce the notion of developed templates and a simple method of constructing such templates without explicit use of induction. We then show that in any transformations of programs using the developed templates, the correctness of transformation can be verified automatically.

The rest of the paper is organized as follows. In Section 2, we recall basic notions in term rewriting that will be used throughout this paper. In Section 3, we present some motivating examples of TRS transformations using templates, and introduce our framework. We dis-
cuss how the correctness of TRS transformation is validated using templates in Section 4. We first describe a simple technique to prove the equality of two TRSs using a manual transformation from one to the other. We then introduce notions of CS homomorphisms and developed templates to conduct such a manual transformation at the template level. In Section 5, we briefly explain RAPT, which implements our framework. Various other examples executed by RAPT are given in Section 6. We conclude with the results in Section 7.

2. Preliminaries

This section introduces notions of term rewriting systems\(^1\),\(^6\).

Let \( \mathcal{F} \) and \( \mathcal{V} \) be a set of function symbols and variables, respectively. We assume that these sets are mutually disjoint. Any function symbol \( f \in \mathcal{F} \) has its arity (denoted by \( \text{arity}(f) \)). We define the set \( T(\mathcal{F}, \mathcal{V}) \) of terms like this: (1) \( \mathcal{V} \subseteq T(\mathcal{F}, \mathcal{V}) \); and (2) \( f(t_1, \ldots, t_n) \subseteq T(\mathcal{F}, \mathcal{V}) \) for any \( f \in \mathcal{F} \) such that \( \text{arity}(f) = n \) and \( t_1, \ldots, t_n \in T(\mathcal{F}, \mathcal{V}) \). A term without variables is a \textit{ground} term. The set of ground terms is denoted by \( T(\mathcal{F}) \). A \textit{linear} term is a term in which any variable appears at most once. For any term \( s \), the set of function symbols and variables in \( s \) are denoted by \( \mathcal{F}(s) \) and \( \mathcal{V}(s) \), respectively. For a term \( s = f(s_1, \ldots, s_n) \), the root symbol of \( s \) is \( f \) (denoted by root\((s) = f\)). A substitution \( \theta \) is a mapping from \( \mathcal{V} \) to \( T(\mathcal{F}, \mathcal{V}) \). A substitution \( \theta \) is extended to a mapping \( \theta \) over term \( T(\mathcal{F}, \mathcal{V}) \) like this: (1) \( \theta(x) = \theta(x) \) if \( x \in \mathcal{V} \), (2) \( \theta(f(s_1, \ldots, s_n)) = f(\theta(s_1), \ldots, \theta(s_n)) \). We usually identify \( \theta \) and \( \theta(s) \). We denote \( \theta(s) \) instead of \( \theta(s) \). The domain of a substitution \( \theta \) (denoted by dom\((\theta)\)) is defined by \( \text{dom}(\theta) = \{ x \in \mathcal{V} \mid \theta(x) \neq x \} \). Consider special (indexed) constants \( \square_i \) \( (i \geq 1) \) called holes such that \( \square_i \notin \mathcal{F} \) and \( \text{domain} (\theta) \). An (indexed) context \( C \) is an element of \( T(\mathcal{F} \cup \{ \square_i \mid i \geq 1 \}, \mathcal{V}) \). \( C[s_1, \ldots, s_n] \) is the result of \( C \) replacing \( \square_i \) by \( s_i \) \( \forall i \in \{1, \ldots, n\} \) (indexed replacement). A context \( C \) with precisely one hole is denoted by \( C[\square] \). The set of contexts is denoted by \( T^C(\mathcal{F}, \mathcal{V}) \); its subset \( T^C(\mathcal{F} \cup \{ \square_i \mid 1 \leq i \leq n \}, \mathcal{V}) \) is denoted by \( T^C_n(\mathcal{F}, \mathcal{V}) \). \( T^C(\mathcal{F}) \) and \( T^C_n(\mathcal{F}) \) are defined in the same way as \( T(\mathcal{F}) \).

A pair \( \langle l, r \rangle \) of terms is a rewrite rule if \( l \notin \mathcal{V} \) and \( \mathcal{V}(l) \supseteq \mathcal{V}(r) \). We usually write the rewrite rule \( \langle l, r \rangle \) as \( l \rightarrow r \). A term \( s \) \textit{reduces} to a term \( t \) by \( \mathcal{R} \) (denoted by \( s \rightarrow_\mathcal{R} t \)) if there exists a context \( C[\square] \), a substitution \( \theta \) and a rewrite rule \( l \rightarrow r \) in \( \mathcal{R} \) such that \( s = C[\theta] \) and \( t = C[r] \). The reflexive transitive closure of \( \rightarrow_\mathcal{R} \) is denoted by \( \rightarrow^*_\mathcal{R} \), the transitive closure by \( \rightarrow^\ast_\mathcal{R} \), and the equivalence closure by \( \leftrightarrow^\ast_\mathcal{R} \). A term \( s \) is in \textit{normal form} when \( s \rightarrow_\mathcal{R} t \) for no term \( t \). \( \text{NF}(\mathcal{R}) \) denotes the set of terms in normal form. An equation is a pair of terms; we usually write an equation \( l \approx r \). For a set \( \mathcal{E} \) of equations, we write \( s \leftrightarrow t \) if there exists a context \( C[\square] \), a substitution \( \theta \), and an equation \( l \approx r \) in \( \mathcal{E} \) such that \( s = C[\theta] \) and \( t = C[r] \) or \( s = C[\theta] \) and \( t = C[r] \). The reflexive transitive closure of \( \leftrightarrow \) is denoted by \( \leftrightarrow^\ast \).

A rewrite rule \( l \rightarrow r \) is \textit{left-linear} when \( l \) is linear; a \( \mathcal{R} \) \textit{is left-linear} if every rewrite rule in \( \mathcal{R} \) is left-linear. We assume that the set \( \mathcal{F} \) of function symbols is divided into two disjoint sets—the set \( \mathcal{F}_d \) of defined function symbols and the set \( \mathcal{F}_c \) of constructor symbols. Elements of \( T(\mathcal{F}_c, \mathcal{V}) \) are called constructor terms. A rewrite rule \( l \rightarrow r \) is a constructor rule if \( l = f(l_1, \ldots, l_n) \) for some \( f \in \mathcal{F}_d \) and \( l_1, \ldots, l_n \in T(\mathcal{F}_c, \mathcal{V}) \). A \( \mathcal{R} \) is a \textit{constructor system} (CS for short) if every rewrite rule is a constructor rule. Without loss of generality, we assume \( \mathcal{F}_i = \{ \text{root}(l) \mid l \rightarrow R \} \) for a given \( \mathcal{R} \). A \( \mathcal{R} \) is \textit{confluent}, or has the Church–Rosser property, (CR(\( \mathcal{R} \)) if, for any terms \( s, s_1, s_2 \), \( s \rightarrow_\mathcal{R} s_1 \) and \( s \rightarrow_\mathcal{R} s_2 \) imply that there exists a term \( t \) such that \( s_1 \rightarrow_\mathcal{R} t \) and \( s_2 \rightarrow_\mathcal{R} t \). Note that CR(\( \mathcal{R} \)), \( s, t \in \text{NF}(\mathcal{R}) \) and \( s \leftrightarrow^*_{\mathcal{R}} t \) imply \( s = t \). A \( \mathcal{R} \) is \textit{strongly normalizing} (SN(\( \mathcal{R} \)) if there exists no infinite reduction \( s_1 \rightarrow_{\mathcal{R}} s_2 \rightarrow_{\mathcal{R}} s_3 \rightarrow_{\mathcal{R}} \cdots \).

3. Transformation by templates

This section introduces our framework of program transformation in which programs are formalized by TRSs. Let us start with some motivating examples.

Example 3.1 A program that computes the summation of a list is specified by the following TRS \( \mathcal{R}_{\text{sum}} \), in which the natural numbers 0, 1, 2, \ldots are expressed as \( 0, s(0), s(s(0)), \ldots \).
For example, we have transformation template: the examples can be generalized to a more abstract similarly.

Note that the associativity of the function \(\sum\) and the property \(\cdot\) are satisfied: all the conditions of a pattern variable is satisfied:

\[
\mathcal{H} = \{ g(b, u_1) \Rightarrow g(u_1, b), g(g(u_2, v_2), w_2) \Rightarrow g(u_2, g(v_2, w_2)) \}
\]

We are now going to introduce a formal definition of a “transformation template”.

**Definition 3.4** Let \(\mathcal{F}\) be a set of pattern variables (disjoint from \(\mathcal{F}\) and \(\mathcal{V}\)) where each pattern variable \(p \in \mathcal{F}\) has its arity (denoted by \(\text{arity}(p)\)). A pattern is a term in \(T(\mathcal{F} \cup \mathcal{F}, \mathcal{V})\). A TRANS pattern \(\mathcal{P}\) is a set of rewriting rules over patterns. A hypothesis \(\mathcal{H}\) is a set of equations over patterns. A transformation template (or just template) is a triple \((\mathcal{P}, \mathcal{P}', \mathcal{H})\) of two TRANS patterns \(\mathcal{P}, \mathcal{P}'\) and a hypothesis \(\mathcal{H}\). For patterns \(s, t\), we define \(s \rightarrow \mathcal{P} t, s \leftrightarrow \mathcal{H} t\), etc. similarly for terms.

To achieve program transformation using templates, we need a mechanism to specify how a template is applied to a concrete TRS. For this, we use a variant of the notion of tree homomorphism—or we call this a tree homomorphism.

**Definition 3.5** Let \(\varphi\) be a mapping from \(\mathcal{F} \cup \mathcal{V}\) to \(T(\mathcal{F} \cup \mathcal{F}, \mathcal{V})\). We say \(\varphi\) is a term homomorphism if the following conditions are satisfied:

1. \(\varphi(p) \in T_{\text{arity}(p)}(\mathcal{F})\) for any \(p \in \text{dom}_{\mathcal{F}}(\varphi)\),
2. \(\varphi(x) \in \mathcal{V}\) for any \(x \in \text{dom}_{\mathcal{V}}(\varphi)\),
3. \(\varphi\) is injective on \(\text{dom}_{\mathcal{V}}(\varphi)\), i.e., for any
x, y ∈ dom\(\varphi\), if \(x \neq y\) then \(\varphi(x) \neq \varphi(y)\),
where \(\text{dom}_{\mathcal{A}}(\varphi) = \{ p \in \mathcal{A} \mid \varphi(p) \neq p[\Box_1, \ldots, \Box_{\text{arity}(p)}]\}\) and \(\text{dom}_{\mathcal{Y}}(\varphi) = \{ x \in \mathcal{Y} \mid \varphi(x) \neq x \}\). A term homomorphism \(\varphi\) is extended to a mapping over \(T(\mathcal{F} \cup \mathcal{X}, \mathcal{Y})\) as follows:

\[
\varphi(s) = \begin{cases} 
\varphi(x) & \text{if } s = x \in \mathcal{Y} \\
f(\varphi(s_1), \ldots, \varphi(s_n)) & \text{if } s = f(s_1, \ldots, s_n), \ f \in \mathcal{F} \\
\varphi(p)(\varphi(s_1), \ldots, \varphi(s_n)) & \text{if } s = p(s_1, \ldots, s_n), \ p \in \mathcal{X}.
\end{cases}
\]

Note that \(\varphi(s)\) is a pattern for any pattern \(s\) and term homomorphism \(\varphi\). For a term homomorphism \(\varphi\) and a rewrite rule \(l \rightarrow r\) (an equation \(s \approx t\) over patterns, \(\varphi(l) \rightarrow \varphi(r)\) (resp. \(\varphi(l) \rightarrow \varphi(r)\)) of term homomorphisms. To show this, we extend the property of term homomorphisms is proved in term homomorphism preserves reduction. This (I)

\(\varphi(t) = \varphi(f(t_1, \ldots, t_n)\varphi(\theta)) = \varphi(f(t_1, \ldots, t_n))\varphi(\theta) = \varphi(t)\varphi(\theta).
\]

(3) \(t = p(t_1, \ldots, t_n)\) with \(p \in \mathcal{X}\). Then,

\[
\varphi(t) = \varphi(p(t_1, \ldots, t_n)) = \varphi(p(t_1, \theta)) = \varphi(s_1) = \varphi(t_1)(\varphi(\theta)) = \varphi(t)\varphi(\theta).
\]

(\(\varphi(p) = \emptyset\) \(\Box\))

\[L_{\text{Lemma 3.7}}\] Let \(t\) be a pattern, \(C[\ ]\) a context, and \(\varphi\) a term homomorphism. Then, \(\varphi(C[t]) = \varphi(C)[\varphi(t), \ldots, \varphi(t)]\).

(Proof) The proof proceeds by induction on the size of \(C[\ ]\).

(1) \(C[\ ] = \emptyset\). Trivial.

(2) \(C[\ ] = f(s_1, \ldots, C'[\ldots, s_n])\) with \(f \in \mathcal{F}\). Then,

\[
\varphi(f(s_1, \ldots, C'[\ldots, s_n])) = f(\varphi(s_1), \ldots, \varphi(C'[\ldots, s_n])) = f(\varphi(s_1), \ldots, \varphi(C'[\varphi(t), \ldots, \varphi(t)] \ldots, \varphi(s_n)))
\]

(3) \(C[\ ] = p(s_1, \ldots, s_n)\) with \(s_i = C'[\ ]\) and \(p \in \mathcal{X}\). Then,

\[
\varphi(p(s_1, \ldots, C'[\ldots, s_n])) = \varphi(p)(\varphi(s_1), \ldots, \varphi(C'[\ldots, s_n])) = \varphi(p)(\varphi(s_1), \ldots, \varphi(C'[\varphi(t), \ldots, \varphi(t)] \ldots, \varphi(s_n)))
\]

(Proof) Suppose \(s \rightarrow_{p} t\). Then, there exists a context \(C[\ ]\), a substitution \(\theta\), and a rewrite rule pattern \(l \rightarrow r\) in \(\mathcal{P}\) such that \(s = C[\theta]\) and \(r = C[r\theta]\). Also, \(\varphi(l) = \mathcal{Y}(l) \subseteq \text{dom}(\varphi)\) by
\[ \forall (P) \land \forall (R). \text{ Then,} \\
\varphi(s) = \varphi(C)[\varphi(\theta), \ldots, \varphi(\theta)] \\
\text{(by Lemma 3.7)} \\
= \varphi(C)[\varphi(l)(\varphi(\theta), \ldots, \varphi(l)\varphi(\theta))] \\
\text{(by Lemma 3.6)} \\
\text{\textsuperscript{*}}_R \varphi(C)[\varphi(r)(\varphi(\theta), \ldots, \varphi(r)\varphi(\theta))] \\
\text{(by Lemma 3.6)} \\
= \varphi(C)[\varphi(r\theta), \ldots, \varphi(r\theta)] \\
\text{(by Lemma 3.7)} \\
= \varphi(t). \\
\]

It can be shown that \( s \mapsto_R t \) implies \( \varphi(s) \mapsto \varphi(t) \) in a similar way.

The TRS transformation by a template is defined as follows.

**Definition 3.9** Let \( \langle P, P', H \rangle \) be a template. A TRS \( R \) is transformed into \( R' \) by \( \langle P, P', H \rangle \) if there exists a term homomorphism \( \varphi \) such that \( R = \varphi(P) \cup R_{\text{com}} \) and \( R' = \varphi(P') \cup R_{\text{com}} \) for some TRS \( R_{\text{com}} \).

Note that the hypothesis \( H \) is not used in the definition of the transformation, but it will be needed later when we discuss the correctness of the transformation.

**Example 3.10** Let \( R_{\text{sum}}, R'_{\text{sum}} \) be the TRSs in Example 3.1, and \( \langle P, P', H \rangle \) the template given in Example 3.3. Then, the following term homomorphism \( \varphi \) satisfies \( R_{\text{sum}} = \varphi(P) \) and \( R'_{\text{sum}} = \varphi(P') \).

\[
\begin{align*}
\varphi &= \{ f \mapsto \text{sum}(\overline{1}), \quad u_1 \mapsto x_1, \quad u_6 \mapsto x_6, \\
g \mapsto \overline{(1, \overline{2})}, \quad v_1 \mapsto y_1, \quad v_6 \mapsto y_6, \\
f_1 \mapsto \text{sum1}(\overline{1}, \overline{2}), \quad u_2 \mapsto x_2, \quad u_6 \mapsto x_6, \\
a \mapsto [], \quad v_3 \mapsto x_3, \quad u_7 \mapsto x_7, \\
b \mapsto 0, \quad u_3 \mapsto y_1, \quad u_7 \mapsto y_8, \\
c \mapsto \overline{1}, \overline{2}, \quad u_4 \mapsto x_4, \quad w_8 \mapsto z_8, \\
d \mapsto \text{sum}(\overline{2}), \quad u_5 \mapsto x_5. 
\end{align*}
\]

Thus, the TRS \( R_{\text{sum}} \) is transformed into \( R'_{\text{sum}} \) by \( \langle P, P', H \rangle \) where \( R_{\text{com}} = \emptyset \).

**Example 3.11** Let \( R_{\text{cat}}, R'_{\text{cat}} \) be the TRSs in Example 3.2, and \( \langle P, P', H \rangle \) the template given in Example 3.3. Then, the following term homomorphism \( \varphi \) satisfies \( R_{\text{cat}} = \varphi(P) \) and \( R'_{\text{cat}} = \varphi(P') \).

\[
\begin{align*}
\varphi &= \{ f \mapsto \text{cat}(\overline{1}), \quad u_1 \mapsto x_1, \quad u_6 \mapsto x_6, \\
g \mapsto \text{app}(\overline{1}, \overline{2}), \quad v_1 \mapsto y_1, \quad v_6 \mapsto y_6, \\
f_1 \mapsto \text{cat1}(\overline{1}, \overline{2}), \quad u_2 \mapsto x_2, \quad u_6 \mapsto x_6, \\
a \mapsto [], \quad v_3 \mapsto x_3, \quad u_7 \mapsto x_7, \\
b \mapsto [], \quad u_3 \mapsto x_3, \quad u_7 \mapsto x_8, \\
c \mapsto \overline{1}, \overline{2}, \quad u_4 \mapsto x_4, \quad v_8 \mapsto z_8, \\
d \mapsto \text{sum}(\overline{2}), \quad u_5 \mapsto x_5. 
\end{align*}
\]

Thus, the TRS \( R_{\text{cat}} \) is transformed into \( R'_{\text{cat}} \) by \( \langle P, P', H \rangle \) where \( R_{\text{com}} = \emptyset \).

Readers can easily observe from these examples that \( R_{\text{sum}} \) and \( R_{\text{cat}} \) are respectively transformed into \( R'_{\text{sum}} \) and \( R'_{\text{cat}} \) in the same way. A question naturally arises from this observation: does the template guarantee the correctness of all the transformations done by that template?

In the next section, we will discuss the criteria for the templates for the correct transformation and try to give a definite answer to this question.

### 4. Correctness of the templates

This section discusses how the correctness of program transformation using templates is validated, i.e., when the equivalence of the input and output programs of program transformations are guaranteed. Intuitively, a program transformation from one program to another is correct if these programs compute the same answer for any input data. In term rewriting, this notion is formalized in the following way.

**Definition 4.1** Let \( \mathcal{F} \) be a set of function symbols such that \( \mathcal{F}_e \subseteq \mathcal{F} \subseteq \mathcal{F} \). Two TRSs, \( R \) and \( R' \), are said to be equivalent for \( \mathcal{F} \) (notation, \( \mathcal{R} \approx_{\mathcal{F}} \mathcal{R}' \)), if for any ground term \( s \in T(\mathcal{F}) \) and ground constructor term \( t \in T(\mathcal{F}_e) \), \( s \mapsto_R t \) if and only if \( s \mapsto_{R'} t \) holds.

At this juncture, we need to make a short remark about the definition of the equivalence of TRSs. In a program transformation from \( R \) to \( R' \), one cannot generally expect \( s \mapsto_R t \) if \( s \mapsto_{R'} t \) for all ground terms \( s \in T(\mathcal{F}) \) and ground constructor term \( t \in T(\mathcal{F}_e) \). This is because one TRS may use some subfunctions that the other may not have. This is why the equivalence of TRSs is defined with respect to a set \( \mathcal{F} \) of function symbols. Intuitively, the functions in \( \mathcal{F} \) are those originally required to compute by the TRSs in comparison.

**Example 4.2** Let us consider \( R_{\text{sum}} \) and \( R'_{\text{sum}} \) in Example 3.1. Then, \( \text{sum1}(\_), s(0) \mapsto_{\text{sum}} s(0) \in T(\mathcal{F}_e) \), but \( \text{sum1}(\_), s(0) \) is in a normal form of \( R_{\text{sum}} \), because \( R_{\text{sum}} \) has no rewrite rules for \( \text{sum1} \). Thus, \( R_{\text{sum}} \not\approx_{\mathcal{F}} R'_{\text{sum}} \) for any \( \mathcal{F} \) containing \( \text{sum1} \). Rather, we should consider the equivalence of these TRSs by setting \( \mathcal{F} = \{ \text{sum}, +, \_, :, [], s, 0 \} \); indeed, in that case we can prove \( R_{\text{sum}} \approx_{\mathcal{F}} R'_{\text{sum}} \).

Although whether two TRSs are equivalent cannot generally be decided, it is known that two TRSs are equivalent when there exists
an equivalent transformation from one to the other for some restricted class of TRSs. Let us now simplify and improve this technique for our framework.

For a set \( \mathcal{G} \) of function symbols, we speak of a TRS \( \mathcal{R} \) (or a set \( \mathcal{E} \) of equations) over \( \mathcal{G} \) when all rewrite rules (resp. equations) consist of terms in \( T(\mathcal{G}, \mathcal{F}) \).

**Definition 4.3** Let \( \mathcal{R}_0 \) be a left-linear CS over \( \mathcal{F}_0 \) and \( \mathcal{E} \) be a set of equations over \( \mathcal{F}_0 \). An equivalent transformation sequence under \( \mathcal{E} \) is a sequence \( \mathcal{R}_0, \ldots, \mathcal{R}_n \) of TRSs (over \( \mathcal{F}_0, \ldots, \mathcal{F}_n \), respectively) such that \( \mathcal{R}_{k+1} \) is obtained from \( \mathcal{R}_k \) by applying one of the following inference rules:

1. **Introduction**
   \[ \mathcal{R}_{k+1} = \mathcal{R}_k \cup \{ f(x_1, \ldots, x_n) \rightarrow r \} \]
   provided that \( f(x_1, \ldots, x_n) \rightarrow r \) is a left-linear constructor rewrite rule such that \( f \notin \mathcal{F}_k \) and \( r \in T(\mathcal{F}_k, \mathcal{F}') \). We put \( \mathcal{F}_{k+1} = \mathcal{F}_k \cup \{ f \} \).

2. **Addition**
   \[ \mathcal{R}_{k+1} = \mathcal{R}_k \cup \{ l \rightarrow r \} \]
   provided \( l \stackrel{\ast}{\rightarrow}^{\mathcal{R}_k, \mathcal{E}} r \) holds.

3. **Elimination**
   \[ \mathcal{R}_{k+1} = \mathcal{R}_k \setminus \{ l \rightarrow r \} \]
   This is the case, we write \( \mathcal{R}_k \Rightarrow \mathcal{R}_{k+1} \).

(In the Addition and Elimination rules, \( \mathcal{F}_{k+1} \subseteq \mathcal{F}_k \) provided that \( \mathcal{R}_{k+1} \) is a TRS over \( \mathcal{F}_{k+1} \).) The reflexive transitive closure of \( \Rightarrow \) is denoted by \( \Rightarrow^* \). We indicate the rule of \( \Rightarrow \) by \( \Rightarrow^T \), \( \Rightarrow^A \), or \( \Rightarrow^E \). Finally, we say there exists an equivalent transformation from \( \mathcal{R} \) to \( \mathcal{R}' \) under \( \mathcal{E} \) when there exists an equivalent transformation sequence \( \mathcal{R}_0 \Rightarrow^* \mathcal{R}_1 \Rightarrow^* \cdots \Rightarrow^* \mathcal{R}' \) under \( \mathcal{E} \).

To state the criteria for the equivalence of TRSs based on equivalence transformation, we introduce some standard notions related to proving the inductive theorem proving in what follows.

Suppose \( \mathcal{F}_0 \subseteq \mathcal{G} \subseteq \mathcal{F} \). A TRS \( \mathcal{R} \) is sufficiently complete for \( \mathcal{G} \) (\( SC(\mathcal{R}, \mathcal{G}) \)) when for any ground term \( s \in T(\mathcal{G}) \) there exists \( t \in T(\mathcal{F}_0) \) such that \( s \Rightarrow^* \mathcal{R} t \). A substitution \( \theta \) is ground on \( \mathcal{G} \) if \( \theta(x) \in T(\mathcal{G}) \) for any \( x \in \text{dom}(\theta) \). An equation \( s \approx t \) is a ground substitution on \( \mathcal{G} \) such that \( \mathcal{V}(s) \cup \mathcal{V}(t) \subseteq \text{dom}(\theta_g) \), \( \theta_g \stackrel{s \approx t}{\rightarrow}^{\mathcal{R}_A} \theta_g \) holds. For a set \( \mathcal{E} \) of equations, we write \( \mathcal{R} \vdash^* \mathcal{G} \rightarrow^* \mathcal{E} \) when \( \mathcal{R}, \mathcal{G} \vdash^* \mathcal{E} \) for any \( s \approx t \) in \( \mathcal{E} \).

**Theorem 4.4** Let \( \mathcal{G} \) and \( \mathcal{G}' \) be sets of function symbols such that \( \mathcal{F}_0 \subseteq \mathcal{G} \subseteq \mathcal{F} \). Let \( \mathcal{R} \) be a left-linear CS over \( \mathcal{G} \), \( \mathcal{E} \) a set of equations over \( \mathcal{G} \), and \( \mathcal{R}' \) a TRS over \( \mathcal{G}' \). Suppose that \( \mathcal{R}, \mathcal{G} \vdash^* \mathcal{E} \) and \( \mathcal{E} \) has an equivalent transformation from \( \mathcal{R} \) to \( \mathcal{R}' \). Then, \( \text{CR}(\mathcal{R}) \land SC(\mathcal{R}, \mathcal{G}) \land SC(\mathcal{R}', \mathcal{G}') \) imply \( \mathcal{R} \approx_{\mathcal{G} \setminus \mathcal{G}'} \mathcal{R}' \).

**(Proof)** Suppose \( \mathcal{R} \Rightarrow^* \mathcal{R}_1 \Rightarrow^* \mathcal{R}_A \Rightarrow^* \mathcal{R}' \). We first show some properties of \( \mathcal{R}_1 \). Let \( \mathcal{R}_0 = \mathcal{R} \) and \( \mathcal{R}_i \Rightarrow^* \mathcal{R}_i \cup \{ f(x_1, \ldots, x_n) \rightarrow r \} = \mathcal{R}_{i+1} \).

Then, \( \text{SC}(\mathcal{R}_i, \mathcal{F}_i) \) implies \( \text{SC}(\mathcal{R}_{i+1}, \mathcal{F}_i \cup \{ f \}) \) by the definition of the Introductions rule. Thus, by our assumption \( \text{SC}(\mathcal{R}, \mathcal{G}) \), it easily follows by induction on the length of \( \mathcal{R} \Rightarrow^* \mathcal{R} \), that

\[ \text{SC}(\mathcal{R}_i, \mathcal{F}_i) \]

for all \( i \) such that \( \mathcal{R} \Rightarrow^* \mathcal{R}_i \). Thus, we may assume w.l.o.g. \( \text{SC}(\mathcal{R}_1, \mathcal{F}_1) \) because we may ignore any function symbols not appearing even in \( \mathcal{R}_1 \). It is clear that \( \mathcal{R} \subseteq \mathcal{R}_1 \) by the definition of the Introduction rule. Also, from \( \text{CR}(\mathcal{R}_0) \) and the fact that each introduced rewrite rule \( f(x_1, \ldots, x_n) \rightarrow r \) is a term, and an equation \( \mathcal{R}_i \) and \( \mathcal{R} \), it follows that \( \text{CR}(\mathcal{R}_1) \) using the commutativity of TRSs. Thus, for \( \mathcal{R}_1 \), we have (1) \( \text{SC}(\mathcal{R}_1, \mathcal{F}_1) \), (2) \( \mathcal{R} \subseteq \mathcal{R}_1 \), and (3) \( \text{CR}(\mathcal{R}_1) \). We next show that \( \mathcal{R} \Rightarrow^* \mathcal{R} \). (i.e., for any \( s, t \in T(\mathcal{G}) \), \( s \Rightarrow^* \mathcal{R} t \) if \( s \Rightarrow^* \mathcal{R} t \).

(1) \( \mathcal{R} \Rightarrow^* \mathcal{R} \) on \( T(\mathcal{G}) \). (i.e., for any \( s, t \in T(\mathcal{G}) \), \( s \Rightarrow^* \mathcal{R} t \) if \( s \Rightarrow^* \mathcal{R} t \).

(2) \( \mathcal{R} \Rightarrow^* \mathcal{R} \) on \( T(\mathcal{G}) \). (i.e., for any \( s, t \in T(\mathcal{G}) \), \( s \Rightarrow^* \mathcal{R} t \) if \( s \Rightarrow^* \mathcal{R} t \).

(3) \( \mathcal{R} \Rightarrow^* \mathcal{R} \) on \( T(\mathcal{G}) \). (i.e., for any \( s, t \in T(\mathcal{G}) \), \( s \Rightarrow^* \mathcal{R} t \) if \( s \Rightarrow^* \mathcal{R} t \).

(4) \( \mathcal{R} \Rightarrow^* \mathcal{R} \) on \( T(\mathcal{G}) \). (i.e., for any \( s, t \in T(\mathcal{G}) \), \( s \Rightarrow^* \mathcal{R} t \) if \( s \Rightarrow^* \mathcal{R} t \).
by our assumption $R, G \vdash_{\text{nd}} E$. Thus, by $R \subseteq R_I$, $C[\theta^g] \vdash_{R} C[\theta^g]$ holds. Hence, $\vdash_{R} \subseteq \vdash_{E}$ on $F$. It is easy to see by the definition of the Addition rule that $\vdash_{R} = \vdash_{E \cup R_A}$ on $T(F, Y)$. Hence, $\vdash_{R} \subseteq \vdash_{E \cup R_A} \subseteq \vdash_{R}$. 

(3) $\vdash_{R} = \vdash_{R'}$ on $T(G)$ (i.e., for any $s, t \in T(G)$, $s \equiv_{R} t$ if $s \equiv_{R'} t$.)

(2) It easily follows from item 2 and the definition of the Elimination rule. 

(1) Suppose that $s \equiv_{R} t$ where $s, t \in T(G)$. From SC($R', G'$), there exist ground constructor terms $s', t' \in T(F, E)$ such that $s \equiv_{R} s'$ and $t \equiv_{R} t'$. Since we have already shown $\vdash_{R} \equiv_{R'}$ on $T(G)$, it follows that $s' \equiv_{R_I} s$ and $t' \equiv_{R_I} t$. From CR($R_I$) and $T(F, E) \subseteq \text{NF}(R_I)$, $s' = t'$ holds. This means $s \equiv_{R} s' \equiv_{R} t' \equiv_{R} t$. 

From 1, 3, and $T(G) \subseteq T(G)$, $T(F, E)$, it follows that $\vdash_{R} = \vdash_{R'}$ on $T(G \cap G')$. Finally, we show $R \vdash_{g \cap g'} R'$. Suppose $s \in T(G \cap G')$, $t \in T(F, E)$, and $s \equiv_{R} t$. From SC($R', G'$), there exists a constructor term $t' \in T(F, E)$ such that $s \equiv_{R} t'$. By $\vdash_{R} = \vdash_{R'}$, $s \equiv_{R} t'$. Then, by CR($R$) and $T(F, E) \subseteq \text{NF}(R)$, it follows that $t = t'$. Hence, $s \equiv_{R} t' = t$. Conversely, suppose $s \in T(G \cap G')$, $t \in T(F, E)$. From SC($R, G$), there exists a constructor term $t' \in T(F, E)$ such that $s \equiv_{R} t'$. By $\vdash_{R} = \vdash_{R'}$, $s \equiv_{R} t'$. Then, by CR($R$) and $T(F, E) \subseteq \text{NF}(R)$, it follows that $t = t'$. Hence, $s \equiv_{R} t' = t$. 

Example 4.5 Let $R_{\text{sum}}, R'_{\text{sum}}$ be the TRSs in Example 3.1. Let $E$ be the following set of equations.

$$E \{ \begin{align*}
+ (0, x) &\approx + (x, 0) \\
+ (0, x, y, z) &\approx + (x, + (y, z))
\end{align*} \}$$

Note that any equation in $E$ is an inductive consequence of $R_{\text{sum}}$ for $G = \{ \text{sum}, +, \cdot, [ ], s, 0 \}$, i.e., $R, G \vdash_{\text{nd}} E$.

We now demonstrate an equivalent transformation from $R_{\text{sum}}$ to $R'_{\text{sum}}$ under $E$. Let $R_0 = R_{\text{sum}}$.

(1) Let $R_1 = R_0 \cup \{ \text{sum}_1(x, y) \rightarrow + (y, \text{sum}(x)) \}$. Clearly, $R_0 \Rightarrow R_1$ by the Introduction rule.

(2) Let $R_2 = R_1 \cup \{ \text{sum}(x) \rightarrow \text{sum}_1(x, 0) \}$. Here, we have

$$\begin{align*}
\text{sum}(x) &\leftarrow R_1, + (0, \text{sum}(x)) \\
&\leftarrow R_1, \text{sum}(x, 0)
\end{align*}$$

Thus, $R_1 \Rightarrow R_2$ by the Addition rule.

(3) Let $R_3 = R_2 \cup \{ \text{sum}_1([], x) \rightarrow x \}$. Then, we have

$$\begin{align*}
\text{sum}_1([], x) &\rightarrow R_2, + (x, \text{sum}([])) \\
&\rightarrow R_2, + (x, 0) \\
&\rightarrow E, + (0, x) \\
&\rightarrow R_2, x
\end{align*}$$

Thus, $R_2 \Rightarrow R_3$ by the Addition rule.

(4) Let $R_4 = R_3 \cup \{ \text{sum}_1(x, y, z) \rightarrow \text{sum}_1(y, + (x, z)) \}$. Then, we have

$$\begin{align*}
\text{sum}_1(x, y, z) &\rightarrow R_3, + (z, \text{sum}(x y)) \\
&\rightarrow R_3, + (z, + (x, \text{sum}(y))) \\
&\rightarrow E, + (z, + (x, y)) \\
&\rightarrow R_3, \text{sum}_1(y, + (x, z))
\end{align*}$$

Thus, $R_3 \Rightarrow R_4$ by the Addition rule.

(5) Finally, applying the Elimination rule three times to $R_4$, we obtain $R_{\text{sum}}$.

Thus, there exists an equivalent transformation from $R_{\text{sum}}$ to $R_{\text{sum}}'$ under $E$. It is easily shown that $R_{\text{sum}}$ is confluent and sufficiently complete for $G$ and that $R_{\text{sum}}'$ is sufficiently complete for $G \cup \{ \text{sum}_1 \}$. Therefore, from Theorem 4.4, it follows that $R_{\text{sum}} \simeq_{g} R_{\text{sum}}'$.

For the TRS transformation in Example 3.2, it is easily observed that the correctness of the transformation can be proved exactly in the same way. Thus, one may naturally expect that such manual transformations can be conducted at the template level. A naïve method of proving the correctness of a template $(P, P', H)$ is to find an equivalent transformation (or $P$ to $P'$) under $H$ similar to TRSs. This naïve method, however, does not work because $P \Rightarrow P'$ under $H$ does not imply $\varphi(P) \Rightarrow \varphi(P')$ under $\varphi(H)$ in general. For example, suppose $P = \{ p(x) \rightarrow a(b) \}$ and $P' = P \cup \{ q(x) \rightarrow b \}$. Then, $P \Rightarrow P'$. However, $\varphi(P) \nRightarrow \varphi(P')$ when $\varphi = \{ p \rightarrow f(\square_1), q \rightarrow f(\square_2), a \rightarrow 0, b \rightarrow 1 \}$. The key idea in the proof of Theorem 4.4 is the preservation of the Church-Rosser property under the Introduction rule. In the example above, $\varphi(P')$ does not have the Church-Rosser property even though $P'$ does. Thus, in order to preserve the correctness of each step, in particular the Introduction step in equivalence transformation,
some restrictions on the term homomorphism \( \varphi \) are necessary. This leads us to the notion of CS homomorphisms.

Similar to the set \( \mathcal{F} \) of function symbols, we assume that the set \( \mathcal{A} \) of pattern variables is divided into two disjoint sets—the set \( \mathcal{A}_d \) of defined pattern variables and \( \mathcal{A}_c \) of constructor pattern variables. All the notions concerning constructor TRSs are then naturally extended to TRS patterns.

**Definition 4.4.6** A term homomorphism \( \varphi \) is a CS homomorphism if for any defined pattern variable \( p, \varphi(p) = f(\bar{a}_1, \ldots, \bar{a}_n) \) for some defined function symbol \( f \) and mutually distinct indexes \( i_1, \ldots, i_n \), and (2) \( \text{root}(\varphi(p)) = \text{root}(\varphi(q)) \) implies \( p = q \) for any defined pattern variables \( p \) and \( q \).

We now propose inference rules to develop a template for correct transformations. Once a template is obtained with these inference rules then the correctness of all the TRS transformations obtained by this template are verified automatically.

**Definition 4.4.7** Let \( \mathcal{P}_0 \) be a left-linear CS pattern over \( \mathcal{A}_0 \) and \( \mathcal{H}_u \) a set of equations over \( \mathcal{A}_u \). An equivalent transformation sequence under \( \mathcal{H}_u \) is a sequence \( \mathcal{P}_0, \ldots, \mathcal{P}_{n_u} \) of CS patterns (over \( \mathcal{A}_0, \ldots, \mathcal{A}_u \), respectively) such that \( \mathcal{P}_{n_u} \) is obtained from \( \mathcal{P}_k \) by applying one of the following inference rules:

1. **Introduction**
   \[ \mathcal{P}_{k+1} = \mathcal{P}_k \cup \{ p(x_1, \ldots, x_n) \rightarrow r \} \]
   provided \( p(x_1, \ldots, x_n) \rightarrow r \) is a left-linear rewrite rule such that \( p \in \mathcal{A}_d \setminus \mathcal{A}_k \) and \( r \in T(\mathcal{A}_k \cap \mathcal{A}_d, \mathcal{F}) \). We put \( \mathcal{X}_{k+1} = \mathcal{X}_k \cup \{ p \} \).

2. **Addition**
   \[ \mathcal{P}_{k+1} = \mathcal{P}_k \cup \{ l \rightarrow r \} \]
   provided \( l \vdash_{\mathcal{P}_k \cup \mathcal{H}_u} r \) holds.

3. **Elimination**
   \[ \mathcal{P}_{k+1} = \mathcal{P}_k \setminus \{ l \rightarrow r \} \]

(In Addition and Elimination rules, \( \mathcal{X}_{k+1} \) can be any set of pattern variables such that \( \mathcal{X}_{k+1} \subseteq \mathcal{X}_k \) provided that \( \mathcal{P}_{k+1} \) is a TRS pattern over \( \mathcal{X}_{k+1} \).) Similar to the equivalence transformation of TRSs, we say there exists an equivalent transformation from \( \mathcal{P} \) to \( \mathcal{P}' \) under \( \mathcal{H} \) when there exits an equivalent transformation sequence \( \mathcal{P} \vdash_{\mathcal{P}_1} \cdots \vdash_{\mathcal{P}_{n_u}} \mathcal{P}' \) under \( \mathcal{H} \).

**Definition 4.4.8** A template \( \langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle \) is developed if there exists an equivalent transformation from \( \mathcal{P} \) to \( \mathcal{P}' \) under \( \mathcal{H} \).

**Definition 4.4.9** Let \( \mathcal{G} \) be a set of function symbols such that \( \mathcal{F}_c \subseteq \mathcal{G} \subseteq \mathcal{F} \) and \( \mathcal{R} \) a left-linear CS over \( \mathcal{G} \). Then, \( \mathcal{R} \) is transformed into a TRS \( \mathcal{R}' \) w.r.t. \( \mathcal{G} \) by a template \( \langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle \) if there exists a CS homomorphism \( \varphi \) such that \( \mathcal{R} = \varphi(\mathcal{P}) \cup \mathcal{R}_{\text{com}}, \mathcal{R}' = \varphi(\mathcal{P}') \cup \mathcal{R}_{\text{com}}, \mathcal{R}, \mathcal{G} \vdash_{\text{ind}} \varphi(\mathcal{H}) \), and \( \text{root}(\varphi(p)) \notin \mathcal{F}_d(\mathcal{R}_{\text{com}}) \) for any \( p \in \mathcal{A}_d \setminus \mathcal{A}_d(\mathcal{P}) \).

To give criteria for the correctness of the TRS transformation with the developed templates, we now prepare a couple of lemmas.

**Lemma 4.4.10** Let \( \varphi \) be a CS homomorphism such that \( \text{root}(\varphi(p)) \notin \mathcal{F}_d(\mathcal{R}_{\text{com}}) \) for any \( p \in \mathcal{A}_d \setminus \mathcal{A}_d(\mathcal{P}) \). If \( \mathcal{P} \vdash \mathcal{P}' \) then \( \varphi(\mathcal{P}) \cup \mathcal{R}_{\text{com}} \vdash \varphi(\mathcal{P}') \cup \mathcal{R}_{\text{com}} \).

(Proof) Suppose \( \mathcal{P}' = \mathcal{P} \cup \{ p(x_1, \ldots, x_n) \rightarrow r \} \). It is easy to see by the definition of CS homomorphism and the Introduction rule, and the assumption that \( \text{root}(\varphi(p)) \) does not occur in \( \varphi(\mathcal{P}) \cup \mathcal{R}_{\text{com}} \), that \( \varphi(p(x_1, \ldots, x_n)) \) is linear, and that all the function symbols in \( \varphi(r) \) occur in \( \varphi(\mathcal{P}) \cup \mathcal{R}_{\text{com}} \).

**Lemma 4.4.11** Let \( \varphi \) be a CS homomorphism. If \( \mathcal{P} \vdash \mathcal{P}' \), then \( \varphi(\mathcal{P}) \cup \mathcal{R}_{\text{com}} \vdash \varphi(\mathcal{P}') \cup \mathcal{R}_{\text{com}} \).

(Proof) This immediately follows from Proposition 3.8.

Thus, we arrive at:

**Theorem 4.4.12** Let \( \mathcal{G} \) and \( \mathcal{G}' \) be sets of function symbols such that \( \mathcal{F}_c \subseteq \mathcal{G}, \mathcal{G}' \subseteq \mathcal{F} \). Let \( \mathcal{R} \) be a left-linear CS over \( \mathcal{G} \), \( \mathcal{E} \) a set of equations over \( \mathcal{G} \), and \( \mathcal{R}' \) a TRS over \( \mathcal{G}' \). Suppose that \( \mathcal{R} \) is transformed into a TRS \( \mathcal{R}' \) w.r.t. \( \mathcal{G} \) by a developed template \( \langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle \). Then, \( \text{CR}(\mathcal{R}) \land \text{SC}(\mathcal{R}, \mathcal{G}) \land \text{SC}(\mathcal{R}', \mathcal{G}') \) imply \( \mathcal{R} \simeq_{\mathcal{G} \leftrightarrow \mathcal{G}'} \mathcal{R}' \).

(Proof) By Lemmas 4.4.10 and 4.4.11, and Theorem 4.4.

**Example 4.4.13** We now demonstrate how a template is developed. Let \( \langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle \) be the transformation template in Example 3.3. We show there exists an equivalent transformation from \( \mathcal{P} \) to \( \mathcal{P}' \) under \( \mathcal{H} \).

1. Let \( \mathcal{P}_0 = \mathcal{P} \).
2. Let \( \mathcal{P}_1 = \mathcal{P}_0 \cup \{ f_1(u, v) \rightarrow g(v, f(u)) \} \). Here, \( f_1 \) is a fresh pattern variable. Thus, \( \mathcal{P}_0 \vdash \mathcal{P}_1 \) by the Introduction rule.
3. Let \( \mathcal{P}_2 = \mathcal{P}_1 \cup \{ f(u) \rightarrow f_1(u, b) \} \). Here, we have

\[
f(u) \leftarrow f_1(u, b).
\]
Thus, $\mathcal{P}_1 \Rightarrow \mathcal{P}_2$ by the Addition rule.

(4) Let $\mathcal{P}_3 = \mathcal{P}_2 \cup \{f_1(a, u) \rightarrow u\}$. Here, we have

$$f_1(a, u) \rightarrow_{\mathcal{P}_2} g(u, f(a)) \rightarrow_{\mathcal{P}_3} g(u, b) \rightarrow_{\gamma} g(b, u) \rightarrow_{\mathcal{P}_4} u$$

Thus, we have $\mathcal{P}_2 \Rightarrow \mathcal{P}_3$ by the Addition rule.

(5) Let $\mathcal{P}_4 = \mathcal{P}_3 \cup \{f_1(c(u, v), w) \rightarrow f_1(v, g(w, u))\}$. Here, we have

$$f_1(c(u, v), w) \rightarrow_{\mathcal{P}_3} g(w, f(c(u, v))) \rightarrow_{\mathcal{P}_4} g(w, g(u, f(v))) \rightarrow_{\gamma} g(g(w, u), f(v)) \rightarrow_{\mathcal{P}_4} f_1(v, g(w, u))$$

Thus, we have $\mathcal{P}_3 \Rightarrow \mathcal{P}_4$ by the Addition rule.

(6) Finally, applying the Elimination rules three times to $\mathcal{P}_4$, we obtain $\mathcal{P}'$.

Thus, the template $\langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle$ is developed.

5. Program transformation system RAPT

This section introduces the program transformation system RAPT (Rewriting-based Automated Program Transformation system) that implements our framework.

The source code for RAPT consists of about 5,000 lines and is written in the Standard ML of New Jersey, which is an implementation of the strongly typed functional programming language ML.

RAPT transforms a given many-sorted TRS according to a given transformation template and output it when the equality between the input TRS and obtained TRS is verified. Note that given transformation templates have to be developed to guarantee the correctness of transformation. The extension to many-sorted frameworks is crucial to verifying sufficient completeness; all of our results in the previous sections can easily be adapted to many-sorted frameworks.

Figure 1 shows an example of the input of RAPT. The \texttt{FUNCTIONS} and \texttt{RULES} sections describe input many-sorted TRS and INPUT, OUTPUT and HYPOTHESIS sections give the transformation template. Figure 1 describes the type information for the $R_{sum}$, the TRS $R_{sum}$, and the template that appears in Example 3.3.

The TRS transformation and the verification of the correctness of the transformation is conducted in RAPT by the following six phases:

(1) \textbf{Validation of Input TRS}
Checking whether the input TRS is a left-linear constructor system and well-typed.

(2) \textbf{Precedence Detection}
Verifying whether the input TRS terminates using lexicographic path ordering via precedence detection$^{[4]}$.

(3) \textbf{Proving Confluence and Sufficient Completeness}
Verifying whether the input TRS is confluent and sufficiently complete. Since the termination of the input TRS has already been verified, RAPT only checks the joinability of critical pairs and quasi-reducibility of the TRS.

(4) \textbf{TRS Pattern Matching}
Finding CS homomorphism from the TRS pattern to TRS, using the pattern matching algorithm Match$^{[4]}$.

(5) \textbf{Verification of Hypothesis}
Applying the CS homomorphism to the hypothesis of the template and proving whether it is inductive consequences of the input TRS using rewriting induction$^{[4]}$.

(6) \textbf{Validation of the Output TRS}
Checking whether the output TRS terminates, is left-linear, well-typed, and sufficiently complete.

Figure 2 describes the dependencies between the six phases. If these six phases are successfully passed then RAPT produces output TRSs. The correctness of the transformation is guaranteed in the sense of Definition 4.1, provided that the template is developed.

The following is the output of RAPT for the run of Figure 1.

\[ \begin{array}{l}
\texttt{[ sum(u) } \rightarrow \texttt{ f1(u, 0() ),} \\
\texttt{ f1(nil(), u) } \rightarrow \texttt{ u,} \\
\texttt{ f1(cons(u, v), w) } \rightarrow \texttt{ f1(v, +(v, w) ),} \\
\texttt{ + (0(), u) } \rightarrow \texttt{ u,} \\
\texttt{ +(s(v), w) } \rightarrow \texttt{ s(+ (v, w) ) ]} \\
\end{array} \]

The above TRS corresponds to the output TRS $R'_{sum}$.

6. Examples

This section introduces several developed templates and describes examples of transformations obtained by these templates. Each transformation can be done in less than 100
msec with RAPT.

6.1 A transformation from recursive form to iterative form

We considered transformations from recursive programs to iterative in Section 3. The template \( \langle P, P', H \rangle \) in Example 3.3 is obtained by generalizing such transformations. We have seen that the transformation of the TRS \( R_{\text{sum}} \) and \( R_{\text{cat}} \). Let us try to transform other TRSs using \( \langle P, P', H \rangle \).

**Example 6.1** The following TRS \( R_{\text{rev}} \) specifies a program which computes the reverse of input lists:

\[
\begin{align*}
\text{R}_{\text{rev}}: & \\
\text{rev}[[]] & \rightarrow [] \\
\text{rev}(x::xs) & \rightarrow \text{app}(\text{rev}(xs), x : []) \\
\text{app}([], ys) & \rightarrow ys \\
\text{app}(x::xs, ys) & \rightarrow x : \text{app}(xs, ys)
\end{align*}
\]

Let \( P_{\text{com}} \) be the the common part of \( P \) and \( P' \). The TRS pattern \( P \) does not match \( R_{\text{rev}} \) even though the TRS pattern \( P \setminus P_{\text{com}} \) matches the first two rules of \( R_{\text{rev}} \). Hence, the template \( \langle P_1, P'_1, H \rangle = \langle P \setminus P_{\text{com}}, P' \setminus P_{\text{com}}, H \rangle \) can be used to transform \( R_{\text{rev}} \) into the following \( R'_{\text{rev}} \).
The transformation. In fact, the template
developed may produce incorrect transformations. The
template by moving the common part
templates for

Here, the right-hand side of the third rule of
is not a normal form. By normalizing such
terms, one obtains the preferred \( R'_\text{rev} \).

\[ \begin{align*}
R'_\text{rev} &= \begin{cases}
\text{rev}(xs) \rightarrow \text{rev1}(xs, [\,]) \\
\text{rev1}(\text{[]} ; ys) \rightarrow ys \\
\text{rev1}(x : xs, ys) \rightarrow \text{rev1}(xs, x : ys) \\
\text{app}(\text{[]} ; ys) \rightarrow ys \\
\text{app}(x : xs, ys) \rightarrow x : \text{app}(xs, ys)
\end{cases}
\]

<table>
<thead>
<tr>
<th>Example 6.2</th>
<th>The following TRS ( \mathcal{R}_\text{rev} ) specifies the program that computes a logical multiplication of all elements of an input boolean list.</th>
</tr>
</thead>
</table>
| \[ \begin{align*}
\text{rev}(xs) & \rightarrow \text{rev1}(xs, [\,]) \\
\text{rev1}(\text{[]} ; ys) & \rightarrow ys \\
\text{rev1}(x : xs, ys) & \rightarrow \text{rev1}(xs, x : ys) \\
\text{app}(\text{[]} ; ys) & \rightarrow ys \\
\text{app}(x : xs, ys) & \rightarrow x : \text{app}(xs, ys)
\end{align*} \] |

| \[ \begin{align*}
\text{forall}(\text{[]} ) & \rightarrow \text{true} \\
\text{forall}(x : ys) & \rightarrow \text{and}(x, \text{forall}(ys)) \\
\text{and}(true, true) & \rightarrow \text{true} \\
\text{and}(true, false) & \rightarrow \text{false} \\
\text{and}(false, true) & \rightarrow \text{false} \\
\text{and}(false, false) & \rightarrow \text{false}
\end{align*} \] |

Using the template \( \langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle \), \( \mathcal{R}_\text{rev} \) is transformed to the following TRS \( \mathcal{R}'_\text{rev} \):

\[ \begin{align*}
\text{forall}(u) & \rightarrow \text{fl}(u, true) \\
\text{fl}(\text{[]} , u) & \rightarrow u \\
\text{fl}(u, w; v) & \rightarrow \text{fl}(v, and(w, u)) \\
\text{and}(true, true) & \rightarrow \text{true} \\
\text{and}(true, false) & \rightarrow \text{false} \\
\text{and}(false, true) & \rightarrow \text{false} \\
\text{and}(false, false) & \rightarrow \text{false}
\end{align*} \]

Then, we can easily show that there exists an equivalent transformation from \( \mathcal{P} \) to \( \mathcal{P}' \) under \( \mathcal{H} \) developing the template \( \langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle \) in a similar way. Thus, the template \( \langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle \) is developed so that the correctness of transformations by \( \langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle \) is guaranteed.

Using the template \( \langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle \), \( \mathcal{R}_\text{sum} \) is transformed into \( \mathcal{R}'_\text{sum} \) where \( \mathcal{R}_\text{com} = \{ + (0, y) \rightarrow y, + (s(x), y) \rightarrow s(+ (x, y)) \} \). Also, \( \mathcal{R}_\text{cat} \) is transformed into \( \mathcal{R}'_\text{cat} \), where \( \mathcal{R}_\text{com} = \{ \text{app}(\text{[]} ; y) \rightarrow y, \text{app}(x : xs, ys) \rightarrow x : \text{app}(xs, ys) \} \). The TRS \( \mathcal{R}'_\text{rev} \) is transformed into the following \( \mathcal{R}'_\text{rev} \) where \( \mathcal{R}_\text{com} = \{ \text{app}(\text{[]} ; ys) \rightarrow ys, \text{app}(x : xs, ys) \rightarrow x : \text{app}(xs, ys) \} \):

\[ \begin{align*}
\text{rev}(xs) & \rightarrow \text{rev1}(xs, [\,]) \\
\text{rev1}(\text{[]} ; ys) & \rightarrow ys \\
\text{rev1}(x : xs, ys) & \rightarrow \text{rev1}(xs, x : ys) \\
\text{app}(\text{[]} ; ys) & \rightarrow ys \\
\text{app}(x : xs, ys) & \rightarrow x : \text{app}(xs, ys)
\end{align*} \]

6.2 Fusion transformation

Fusion transformation\(^{5, 19}\) is one of the most recognized techniques for transforming programs.

The following template \( \langle \mathcal{P}_\text{fus1}, \mathcal{P}'_\text{fus1}, \mathcal{H}_\text{fus1} \rangle \) describes a simple fusion transformation:

\[ \begin{align*}
\mathcal{P}_\text{fus1} &= \{ f(u, v, w) \rightarrow \text{g}(h(u, v), w) \\
& \quad \text{g}(a, u) \rightarrow b(u) \\
& \quad \text{g}(c(u, v), w) \rightarrow \text{e}(u, \text{g}(v, w)) \\
& \quad \text{h}(a, u) \rightarrow \text{r}(u) \\
& \quad \text{h}(c(u, v), w) \rightarrow \text{c}(d(u), \text{h}(v, w))
\end{align*} \]

\[ \begin{align*}
\mathcal{P}'_\text{fus1} &= \{ f(a, u, v) \rightarrow \text{g}(r(u), v) \\
& \quad \text{f}(c(u, v), w, z) \rightarrow \text{e}(d(u), \text{f}(v, w, z)) \\
& \quad \text{g}(a, u) \rightarrow \text{b}(u) \\
& \quad \text{g}(c(u, v), w) \rightarrow \text{e}(u, \text{g}(v, w)) \\
& \quad \text{h}(a, u) \rightarrow \text{r}(u) \\
& \quad \text{h}(c(u, v), w) \rightarrow \text{c}(d(u), \text{h}(v, w))
\end{align*} \]

\[ \mathcal{H}_\text{fus1} = \emptyset \]

Note that a hypothesis is not necessary to develop the above template.

We now demonstrate that there exists an equivalent transformation form \( \mathcal{P}_\text{fus1} \) to \( \mathcal{P}'_\text{fus1} \) under \( \mathcal{H}_\text{fus1} \), i.e., \( \langle \mathcal{P}_\text{fus1}, \mathcal{P}'_\text{fus1}, \mathcal{H}_\text{fus1} \rangle \) is developed.

1. Let \( \mathcal{P}_0 = \mathcal{P}_\text{fus1} \).
(2) Let \( \mathcal{P}_1 = \mathcal{P}_0 \cup \{ f(a, u, v) \rightarrow g(r(u), v) \} \).
Here, we have

\[
\begin{align*}
f(a, u, v) & \rightarrow_{\mathcal{P}_0} g(h(a, u), v) \\
& \rightarrow_{\mathcal{P}_0} g(r(u), v).
\end{align*}
\]

Thus, \( \mathcal{P}_0 \Rightarrow \mathcal{P}_1 \) by the Addition rule.

(3) Let \( \mathcal{P}_2 = \mathcal{P}_1 \cup \{ f(c(u, v), w, z) \rightarrow e(d(u), f(v, w, z)) \} \). Here, we have

\[
\begin{align*}
f(c(u, v), w, z) & \rightarrow_{\mathcal{P}_1} g(h(c(u, v), w), z) \\
& \rightarrow_{\mathcal{P}_1} g(d(u), h(v, w), z) \\
& \rightarrow_{\mathcal{P}_1} e(d(u), g(h(v, w), z)) \\
& \leftarrow_{\mathcal{P}_1} e(d(u), f(v, w, z)).
\end{align*}
\]

Thus, \( \mathcal{P}_1 \Rightarrow \mathcal{P}_2 \) by the Addition rule.

(4) Applying the Elimination rule, we obtain

\[
\mathcal{P}'_{\text{fus2}} = \mathcal{P}_2 \setminus \{ f(u, v, w) \rightarrow g(h(u), v, w) \}.
\]

Thus, the template \( (\mathcal{P}_{\text{fus1}}, \mathcal{P}'_{\text{fus1}}, \mathcal{H}_{\text{fus1}}) \) is developed.

**Example 6.3** The following TRS \( \mathcal{R}_{\text{lenapp}} \) specifies the program that computes the sum of the lengths of two lists.

\[
\mathcal{R}_{\text{lenapp}} = \left\{ \begin{array}{l}
\text{lenapp}(x, y) \rightarrow \text{len(app}(x, y)) \\
\text{len}([ ]) \rightarrow 0 \\
\text{len}(x:y) \rightarrow \text{s(len(y)))} \\
\text{app}([ ], y) \rightarrow y \\
\text{app}(x:y, z) \rightarrow x:\text{app}(y, z)
\end{array} \right.
\]

Using the template \( (\mathcal{P}_{\text{fus1}}, \mathcal{P}'_{\text{fus1}}, \mathcal{H}_{\text{fus1}}) \), the TRS \( \mathcal{R}_{\text{lenapp}} \) is transformed into the following TRS \( \mathcal{R}'_{\text{lenapp}} \) where \( \mathcal{R}_{\text{com}} = \emptyset \).

\[
\mathcal{R}'_{\text{lenapp}} = \left\{ \begin{array}{l}
\text{lenapp}([ ], u) \rightarrow \text{len}(u) \\
\text{lenapp}(w; v, w) \rightarrow \text{s(lenapp}(v, w)) \\
\text{len}([ ]) \rightarrow 0 \\
\text{len}(w;v) \rightarrow \text{s(len}(v)) \\
\text{app}([ ], u) \rightarrow u \\
\text{app}(w;v, w) \rightarrow u:\text{app}(v, w)
\end{array} \right.
\]

**Example 6.4** The following TRS \( \mathcal{R}_{\text{ones+}} \) specifies the program that returns a list whose elements are all \( s(0) \) and its length is the sum of two given natural numbers.

\[
\mathcal{R}_{\text{ones+}} = \left\{ \begin{array}{l}
\text{ones+}(x, y) \rightarrow \text{ones}(+(x, y)) \\
\text{ones}(0) \rightarrow [ ] \\
\text{ones}(s(x)) \rightarrow s(0):\text{ones}(x) \\
+(0, x) \rightarrow x \\
+(s(x), y) \rightarrow s(+ (x, y))
\end{array} \right.
\]

Using the template \( (\mathcal{P}_{\text{fus1}}, \mathcal{P}'_{\text{fus1}}, \mathcal{H}_{\text{fus1}}) \), the TRS \( \mathcal{R}_{\text{ones+}} \) is transformed into the following TRS \( \mathcal{R}'_{\text{ones+}} \) where \( \mathcal{R}_{\text{com}} = \emptyset \).

\[
\mathcal{R}'_{\text{ones+}} = \left\{ \begin{array}{l}
\text{ones+}(0, u) \rightarrow \text{ones}(u) \\
\text{ones+}(s(v), w) \rightarrow s(0):\text{ones+}(v, w) \\
\text{ones}(0) \rightarrow [ ] \\
\text{ones}(s(x)) \rightarrow s(0):\text{ones}(x) \\
+(0, u) \rightarrow u \\
+(s(v), w) \rightarrow s(+ (v, w))
\end{array} \right.
\]

Through the above examples, the template \( (\mathcal{P}_{\text{fus1}}, \mathcal{P}'_{\text{fus1}}, \mathcal{H}_{\text{fus1}}) \) can produce simple fusion transformations.

Let us consider another template. The following template \( (\mathcal{P}_{\text{fus2}}, \mathcal{P}'_{\text{fus2}}, \mathcal{H}_{\text{fus2}}) \) produces fusion transformations that are slightly more complicated.

\[
\mathcal{P}_{\text{fus2}} = \begin{array}{l}
h(u) \rightarrow f(g(u)) \\
f(a) \rightarrow b \\
f(c(u, v)) \rightarrow d(r(u), f(v)) \\
g(u) \rightarrow g1(u, a) \\
g1(b, v) \rightarrow k(v) \\
g1(d(u, v), w) \rightarrow g1(v, e(p(u), w))
\end{array}
\]

\[
\mathcal{P}'_{\text{fus2}} = \begin{array}{l}
h(u) \rightarrow h1(u, b) \\
h1(a, v) \rightarrow k(v) \\
h1(c(u, v), w) \rightarrow h1(v, e(p(r(u)), w)) \\
f(a) \rightarrow b \\
f(c(u, v)) \rightarrow d(r(u), f(v)) \\
g(u) \rightarrow g1(u, a) \\
g1(b, v) \rightarrow k(v) \\
g1(d(u, v), w) \rightarrow g1(v, e(p(u), w))
\end{array}
\]

\[
\mathcal{H}_{\text{fus2}} = \begin{array}{l}
f(g1(u, v)) \approx g1(f(u), f(v))
\end{array}
\]

We now show that there exists an equivalent transformation form \( \mathcal{P}_{\text{fus1}} \) under \( \mathcal{H}_{\text{fus1}} \), i.e., \( (\mathcal{P}_{\text{fus2}}, \mathcal{P}'_{\text{fus2}}, \mathcal{H}_{\text{fus2}}) \) is developed.

(1) Let \( \mathcal{P}_0 = \mathcal{P}_{\text{fus2}} \).

(2) Let \( \mathcal{P}_1 = \mathcal{P}_0 \cup \{ h1(u, v) \rightarrow g1(f(u), v) \} \).
Here, \( h1 \) is a fresh pattern variable.
Thus, \( \mathcal{P}_0 \Rightarrow \mathcal{P}_1 \) by the Introduction rule.

(3) Let \( \mathcal{P}_2 = \mathcal{P}_1 \cup \{ h1(a, v) \rightarrow k(v) \} \).
Here, we have

\[
\begin{align*}
h1(a, v) & \rightarrow_{\mathcal{P}_1} g1(f(a), v) \\
& \rightarrow_{\mathcal{P}_1} g1(b, v) \\
& \rightarrow_{\mathcal{P}_1} k(v)
\end{align*}
\]

Thus, \( \mathcal{P}_1 \Rightarrow \mathcal{P}_2 \) by the Addition rule.

(4) Let \( \mathcal{P}_3 = \mathcal{P}_2 \cup \{ h1(c(u, v), w) \rightarrow h1(v, e(p(r(u)), w)) \} \).
Here, we have

\[
\begin{align*}
h1(c(u, v), w) & \rightarrow_{\mathcal{P}_2} h1(v, e(p(r(u)), w))
\end{align*}
\]
Finally, applying the Elimination rules (5) Let $P_2 \Rightarrow P_3$ by the Addition rule.

Thus, $P_2 \Rightarrow P_3$ by the Addition rule.

(6) Finally, applying the Elimination rules twice to $P_4$, we obtain $P'_{fus2}$.

Thus, the template $(P_{fus2}, P'_{fus2}, H_{fus2})$ is developed.

**Example 6.5** The following TRS $R_{mdqr}$ is transformed into the TRS $R'_{mdqr}$ by the template $(P_{fus2}, P'_{fus2}, H_{fus2})$ where $R_{com} = \{\text{double}(0) \rightarrow 0, \text{double}(x) \rightarrow s(s(\text{double}(x)))\}$.  

\[
\begin{align*}
\text{mapdoubleqrev}(xs) & \rightarrow \text{mapdouble}(\text{qrev}(xs)) \\
\text{mapdouble}([ ]) & \rightarrow [ ] \\
\text{mapdouble}(x:xs) & \rightarrow \text{double}(x) \text{mapdouble}(xs) \\
\text{double}(0) & \rightarrow 0 \\
\text{double}(s(x)) & \rightarrow s(s(\text{double}(x))) \\
\text{qrev}([ ]) & \rightarrow \text{qrev1}([ ]) \\
\text{qrev1}(x;ys) & \rightarrow ys \\
\text{qrev1}(x:xs;ys) & \rightarrow \text{qrev1}(x;xs;ys)
\end{align*}
\]

\[
\begin{align*}
\text{mapdoubleqrev}(u) & \rightarrow \text{h1}(u;[ ]) \\
\text{h1}(u;[ ];v) & \rightarrow v \\
\text{h1}(u;v;w) & \rightarrow \text{h1}(u,v;\text{double}(u):w) \\
\text{mapdouble}([ ]) & \rightarrow [ ] \\
\text{mapdouble}(u:v) & \rightarrow \text{double}(u) \text{mapdouble}(v) \\
\text{qrev}(u) & \rightarrow \text{qrev1}(u;[ ]) \\
\text{qrev1}([ ];v) & \rightarrow v \\
\text{qrev1}(u;v;w) & \rightarrow \text{qrev1}(v,u;w) \\
\text{double}(0) & \rightarrow 0 \\
\text{double}(s(x)) & \rightarrow s(s(\text{double}(x)))
\end{align*}
\]

Through the example above, the function mapdoubleqrev is fused by the developed template $(P_{fus2}, P'_{fus2}, H_{fus2})$.

**7. Conclusion**

We presented a rewriting framework for transforming programs with templates in this paper. To guarantee the correctness of transformation within our framework, we introduced a notion of developed templates which are constructed via the step-by-step transformations of TRS patterns. We then showed that in any transformation of programs using the developed templates the correctness of transformation could be verified automatically.

We also described the RAPT system, which implements our framework. RAPT transforms a term rewriting system according to a specified program transformation template and automatically verifies the correctness of the transformation. Examples of the developed transformation templates and their application to the transformation of program were also given.

Another implementation of program transformation using templates is the MAG system, which is based on lambda calculus$^{(9)}$. The correctness of transformation in MAG system is based on Huet and Lang’s framework$^{(11)}$. MAG supports transformations that include modifications of expressions and matching with the help of hypothesis; its target also includes higher-order programs. RAPT does not handle such refinements, and cannot deal with most of the transformations presented by de Moor and Sittampalam$^{(8),(15)}$. The difference between MAG and RAPT, on the other hand, lies in the approach to verifying the hypothesis. Since such hypotheses are generally different in each transformation, one needs to verify them in all transformations. MAG system users usually need to verify the hypothesis by explicit induction in every different transformation. In contrast to this, RAPT proves the hypothesis automatically without needing the help of users. To the best of our knowledge, the program-transformation systems based on templates described in the literature have rarely corporated with automated theorem-proving techniques in the verification of hypotheses. RAPT involves an interesting integration of program-transformation and automated theorem-proving techniques.

**References**

3) Y. Chiba and T. Aoto. RAPT: A program


(Received May 02, 2006)
(Accepted August 01, 2006)

**Yuki China** received his M.S. from Tohoku University in 2005. He is currently a doctoral student at the same university. He is a member of JSSST.

**Takahito Aoto** received his M.S. and Ph.D. from the Japan Advanced Institute for Science and Technology (JAIST). He was at JAIST from 1997 to 1998 as an associate, at Gunma University from 1998 to 2002 as an assistant professor, and at Tohoku University from 2003 to 2004 as a lecturer. He has been at Tohoku University since 2004 as an associate professor. His current research interests are term rewriting and non-classical logics. He is a member of IPSJ, JSSST, EATCS, and ASL.
Yoshihito Toyama received his B.E. from Niigata University in 1975, and his M.E. and D.E. from Tohoku University in 1977 and 1990. He worked as a Research Scientist at NTT Laboratories from 1977 to 1993, and as a Professor at the Japan Advanced Institute of Science and Technology (JAIST) from 1993 to 2000. Since April 2000, he has a professor at the Research Institute of Electrical Communication (RIEC) of Tohoku University. His research interests include term rewriting systems, program theory, and automated theorem proving. He is a member of IEICE, IPSJ, JSSST, ACM, and EATCS.