RAPT: A Program Transformation System based on Term Rewriting

Yuki Chiba, Takahito Aoto, and Yoshihito Toyama

Research Institute of Electrical Communication, Tohoku University, Japan
{chiba, aoto, toyama}@nue.riec.tohoku.ac.jp

Abstract. Chiba et al. (2005) proposed a framework of program transformation by template and automated verification of its correctness based on term rewriting. This paper describes a design and implementation of RAPT which implements this framework. RAPT transforms a term rewriting system according to a specified program transformation template. Presupposing the program transformation template is developed, the correctness of the transformation is automatically verified so that the transformation keeps the relationship between initial ground terms and their normal forms.

1 Introduction

Chiba et al. [1] proposed a framework of program transformation by template and automated verification of its correctness based on term rewriting. In their framework, programs and program schemas are given by term rewriting systems (TRS, for short) and TRS patterns. A program transformation template consists of input and output TRS patterns and a hypothesis which is a set of equations the input TRS has to satisfy to guarantee the correctness of transformation. To automate the program transformation, they introduced a notion of term pattern matching problem and presented a sound and complete algorithm that solves this problem.

We say a program transformation is correct when the input and output programs of the transformation are equivalent. To formalize the equality of programs, they have defined the equivalence of two TRSs by that of relationships between ground terms and constructor ground terms. They introduced a notion of developed templates and a simple method to construct such templates without explicit use of induction. They then showed that in any program transformation by developed templates the correctness of the transformation is guaranteed if (1) the instantiation of hypothesis are inductive consequences of the input TRS, and (2) the input and the output TRSs satisfy some properties such as confluence and sufficient completeness.

This paper describes a design and implementation of RAPT (Rewriting-based Automated Program Transformation system) which implements this framework. RAPT transforms a TRS according to a specified program transformation template. Based on the rewriting induction proposed by Reddy [8], RAPT automatically verifies whether the input TRS satisfies the hypothesis of the transformation template. It also verifies conditions imposed to the input TRS and generated TRS by utilizing standard techniques in term rewriting. Thus, presupposing the program transformation template is developed, the correctness of the transformation is automatically verified so that the transformation keeps the relationship between initial ground terms and their normal forms.

The rest of the paper is organized as follows. In the next section, we give some motivating examples of program transformation by templates based on term rewriting. In Section 3, we explain some backgrounds needed to understand what RAPT performs. In Section 4, we describe an overview of RAPT and phases of RAPT. For each of these phases, we explain the algorithm and heuristics employed in Section 5. In Section 6, we present a way to polish transformation templates. We conclude our work in Section 7.

2 Transformation by templates

In this section, we describe a framework of program transformation by template based on term rewriting through some motivating examples.

Example 1. A program that computes the summation of a list is specified by the following TRS $\mathcal{R}_{sum}$, in which the natural numbers $0, 1, 2, \ldots$ are expressed as $\mathsf{0}, \mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{s}(\mathsf{0})), \ldots$.

$$\mathcal{R}_{sum} \begin{cases} \mathsf{sum}([\,]) & \to \mathsf{0} \\ \mathsf{sum}(x : y) & \to +(x, \mathsf{sum}(y)) \\ +(\mathsf{0}, x) & \to x \\ +(\mathsf{s}(x), y) & \to \mathsf{s}(+(x, y)) \end{cases}$$

This $\mathcal{R}_{sum}$ computes the summation of a list using a recursive call. For instance, $\mathsf{sum}(1 : (2 : (3 : (4 : (5 : [\,]))))) \xrightarrow{*}_{\mathcal{R}_{sum}} +(1, +(2, +(3, +(4, +(5, \mathsf{sum}([\,])))))) \xrightarrow{*}_{\mathcal{R}_{sum}} 15$.

Using the well-known transformation from the recursive form to the iterative (tail-recursive) form, the following different TRS $\mathcal{R}'_{sum}$ for the list summation program is obtained:

$$\mathcal{R}'_{sum} \begin{cases} \mathsf{sum}(x) & \to \mathsf{sum1}(x, \mathsf{0}) \\ \mathsf{sum1}([\,], x) & \to x \\ \mathsf{sum1}(x : y, z) & \to \mathsf{sum1}(y, +(z, x)) \\ +(\mathsf{0}, x) & \to x \\ +(\mathsf{s}(x), y) & \to \mathsf{s}(+(x, y)) \end{cases}$$

$\mathcal{R}'_{sum}$ computes the summation of a list more efficiently without the recursion. The equality of the two programs is shown using the associativity of the function $+$ and the property $+(\mathsf{0}, n) = +(n, \mathsf{0})$.

Example 2. Let us consider another example of program transformation. A program that computes the concatenation of a list of lists is specified by the following TRS $\mathcal{R}_{cat}$.

$$\mathcal{R}_{cat} \begin{cases} \mathsf{cat}([\,]) & \to [\,] \\ \mathsf{cat}(x : y) & \to \mathsf{app}(x, \mathsf{cat}(y)) \\ \mathsf{app}([\,], x) & \to x \\ \mathsf{app}(x : y, z) & \to x : \mathsf{app}(y, z) \end{cases}$$

For example, we have $\mathsf{cat}([[1, 2], [3], [4, 5]]) \xrightarrow{*}_{\mathcal{R}_{cat}} [1, 2, 3, 4, 5]$. Similarly to the Example 1, the transformation from the recursive form to the iterative form gives a more efficient TRS $\mathcal{R}'_{cat}$ as follows.

$$\mathcal{R}'_{cat} \begin{cases} \mathsf{cat}(x) & \to \mathsf{cat1}(x, [\,]) \\ \mathsf{cat1}([\,], x) & \to x \\ \mathsf{cat1}(x : y, z) & \to \mathsf{cat1}(y, \mathsf{app}(z, x)) \\ \mathsf{app}([\,], x) & \to x \\ \mathsf{app}(x : y, z) & \to x : \mathsf{app}(y, z) \end{cases}$$

Note that the associativity of the function app and the property $\mathsf{app}([\,], as) = \mathsf{app}(as, [\,])$ hold. Thus the equality of the two programs is shown similarly.

Example 3. One easily observes that these two transformations in the previous examples can be generalized to a more abstract "transformation template": the *TRS pattern* $\mathcal{P}$

$$\mathcal{P}\begin{cases}\mathsf{f}(\mathsf{a}) & \to \mathsf{b}\\ \mathsf{f}(\mathsf{c}(u,v)) & \to \mathsf{g}(\mathsf{e}(u),\mathsf{f}(v))\\ \mathsf{g}(\mathsf{b},u) & \to u\\ \mathsf{g}(\mathsf{d}(u,v),w) & \to \mathsf{d}(u,\mathsf{g}(v,w))\end{cases}$$

is transformed to the TRS pattern $\mathcal{P}'$

$$\mathcal{P}'\begin{cases}\mathsf{f}(u) & \to \mathsf{f}_1(u,\mathsf{b})\\ \mathsf{f}_1(\mathsf{a},u) & \to u\\ \mathsf{f}_1(\mathsf{c}(u,v),w) & \to \mathsf{f}_1(v,\mathsf{g}(w,\mathsf{e}(u)))\\ \mathsf{g}(\mathsf{b},u) & \to u\\ \mathsf{g}(\mathsf{d}(u,v),w) & \to \mathsf{d}(u,\mathsf{g}(v,w))\end{cases}$$

All the function symbols $\mathsf{f},\mathsf{a},\mathsf{b},\mathsf{g},\ldots$ occurring in the TRS patterns $\mathcal{P}$ and $\mathcal{P}'$ are *pattern variables*. If we match the TRS pattern $\mathcal{P}$ to a concrete TRS $\mathcal{R}$ with an instantiation for these pattern variables, we obtain a more efficient TRS $\mathcal{R}'$ by applying this instantiation to the pattern $\mathcal{P}'$. The equality of $\mathcal{R}$ and $\mathcal{R}'$ is guaranteed when the instantiation satisfies the following equations, called *hypothesis*:

$$\mathcal{H}\begin{cases}\mathsf{g}(\mathsf{b},u) & \approx \mathsf{g}(u,\mathsf{b})\\ \mathsf{g}(\mathsf{g}(u,v),w) & \approx \mathsf{g}(u,\mathsf{g}(v,w))\end{cases}$$

3 Backgrounds

In this section, we recall some notions and results of [1] needed to understand what RAPT performs. The framework in [1] is based on unsorted TRSs. But as commented and discussed in Section 6 of [1], the framework is easily adapted to many-sorted TRSs; we rather should deal with many-sorted TRSs to discuss the correctness of transformation that models realistic programs.

3.1 Many-sorted TRS and its equivalence

Intuitively, a program transformation from one program to another is correct if these programs compute the same result for any input data. So, to discuss the correctness of TRS transformations, we need a notion of equivalence of TRSs. In this subsection, we introduce a notion of equivalence of TRSs that suits to model the correctness of program transformation. Before that, let us introduce some basic notions of many-sorted TRSs and fix some notations.

Let $\mathscr{S}$ be a set of sorts. Let $\mathscr{F},\mathscr{V}$ be sets of many-sorted functions and variables over $\mathscr{S}$. The set $T(\mathscr{F},\mathscr{V})^\tau$ of *many-sorted terms of* τ is defined like this: (1) If a variable x has a sort τ then $x \in T(\mathscr{F},\mathscr{V})^\tau$. (2) If a function symbol f has a sort $\tau_1 \times \cdots \times \tau_n \to \tau$ and $t_1,\ldots,t_n$ are many-sorted terms of sorts $\tau_1,\ldots,\tau_n$, respectively, then $f(t_1,\ldots,t_n) \in T(\mathscr{F},\mathscr{V})^\tau$. The set $T(\mathscr{F},\mathscr{V})$ of all well-sorted terms is given by $T(\mathscr{F},\mathscr{V}) = \bigcup_{\tau\in\mathscr{S}} T(\mathscr{F},\mathscr{V})^\tau$. The set of variables occurring in t is denoted by $\mathscr{V}(t)$. $T(\mathscr{F})$ stands for the set of *ground* terms i.e. terms t satisfying $\mathscr{V}(t) = \emptyset$. The *many-sorted term rewriting system* (many-sorted TRS, for short) is a set of many-sorted rewrite rules $l \to r$, where l, r are

terms of the same sort satisfying $l \notin \mathscr{V}$ and $\mathscr{V}(r) \subseteq \mathscr{V}(l)$. For a set of function symbols $\mathscr{G} \subseteq \mathscr{F}$, we say a many-sorted TRS $\mathcal{R}$ is *over* $\mathscr{G}$ when any function symbol of $\mathcal{R}$ is included in $\mathscr{G}$.

A *substitution* is a mapping from $\mathscr{V}$ to $T(\mathscr{F}, \mathscr{V})$ that preserves sort. A substitution is uniquely extended to the endomorphism on $T(\mathscr{F}, \mathscr{V})$. The *rewrite relation* $\to_{\mathcal{R}}$ of a many-sorted term rewriting system $\mathcal{R}$ is the smallest relation on $T(\mathscr{F}, \mathscr{V})$ satisfying (1) $l \to_{\mathcal{R}} r$ for all $l \to r \in \mathcal{R}$, (2) $s \to_{\mathcal{R}} t$ implies $\theta(s) \to_{\mathcal{R}} \theta(t)$ for any substitution θ, and (3) $s \to_{\mathcal{R}} t$ implies $f(u_1, \ldots, s, \ldots, u_n) \to_{\mathcal{R}} f(u_1, \ldots, t, \ldots, u_n)$ for any $f \in \mathscr{F}$ and $u_1, \ldots, u_n \in T(\mathscr{F}, \mathscr{V})$. The reflexive transitive closure of $\to_{\mathcal{R}}$ is denoted by $\overset{*}{\to}_{\mathcal{R}}$. A many-sorted term t is in a *normal form* of $\mathcal{R}$ when there exists no term s such that $t \to_{\mathcal{R}} s$.

We assume that the set $\mathscr{F}$ of function symbols is divided into disjoint two sets: the set $\mathscr{F}_{\mathrm{d}}$ of *defined function symbols* and the set $\mathscr{F}_{\mathrm{c}}$ of *constructor symbols*. Terms in $T(\mathscr{F}_{\mathrm{c}}, \mathscr{V})$ are called *constructor terms*. A many-sorted TRS is a *constructor system* when for any $l \to r \in \mathcal{R}$, $l = f(l_1, \ldots, l_n)$ for some defined function symbol $f \in \mathscr{F}_{\mathrm{d}}$ and constructor terms $l_1, \ldots, l_n$. Later, we will limit our target of TRS transformation to constructor systems.

Example 4. Let us consider $\mathcal{R}$ in Example 1. We set $\mathscr{S} = \{\, \mathrm{Nat}, \mathrm{List} \,\}$, $\mathscr{F}_{\mathrm{d}} = \{\mathsf{sum}^{\mathrm{List}\to\mathrm{Nat}},\ +^{\mathrm{Nat}\times\mathrm{Nat}\to\mathrm{Nat}}\}$, $\mathscr{F}_{\mathrm{c}} = \{\mathsf{0}^{\mathrm{Nat}},\ \mathsf{s}^{\mathrm{Nat}\to\mathrm{Nat}},\ [\,]^{\mathrm{List}},\ :^{\mathrm{Nat}\times\mathrm{List}\to\mathrm{List}}\}$. Then $\mathcal{R}$ is a many-sorted constructor system.

Definition 1 (equivalence of many-sorted TRSs). *Let $\mathscr{G}$ be a set of function symbols such that $\mathscr{F}_{\mathrm{c}} \subseteq \mathscr{G} \subseteq \mathscr{F}$. Two many-sorted TRSs $\mathcal{R}$ and $\mathcal{R}'$ are said to be* equivalent for $\mathscr{G}$ *(denoted as $\mathcal{R} \simeq_{\mathscr{G}} \mathcal{R}'$) if for any ground term $s \in T(\mathscr{G})$ and ground constructor term $t \in T(\mathscr{F}_{\mathrm{c}})$, $s \overset{*}{\to}_{\mathcal{R}} t$ iff $s \overset{*}{\to}_{\mathcal{R}'} t$ holds.*

In a program transformation from $\mathcal{R}$ to $\mathcal{R}'$, one can not generally expect $s \overset{*}{\to}_{\mathcal{R}} t$ iff $s \overset{*}{\to}_{\mathcal{R}'} t$ for all ground terms $s \in T(\mathscr{F})$ and ground constructor term $t \in T(\mathscr{F}_{\mathrm{c}})$; for, 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 $\mathscr{G}$ of function symbols. Intuitively, functions in $\mathscr{G}$ are those originally requested to compute by the TRSs in comparison.

Example 5. Let us consider $\mathcal{R}$ and $\mathcal{R}'$ in Example 1. Then $\mathsf{sum1}([\,], \mathsf{s}(\mathsf{0})) \to_{\mathcal{R}'} \mathsf{s}(\mathsf{0}) \in T(\mathscr{F}_{\mathrm{c}})$, but $\mathsf{sum1}([\,], \mathsf{s}(\mathsf{0}))$ is in a normal form of $\mathcal{R}$, because $\mathcal{R}$ has no rewrite rules for $\mathsf{sum1}$. Thus $\mathcal{R} \not\simeq_{\mathscr{G}} \mathcal{R}'$ for any $\mathscr{G}$ containing $\mathsf{sum1}$. Rather, one should consider the equivalence of these TRSs by setting $\mathscr{G} = \{\mathsf{sum}, +, :, [\,], \mathsf{s}, \mathsf{0}\}$; indeed, in that case one can prove $\mathcal{R} \simeq_{\mathscr{G}} \mathcal{R}'$.

3.2 TRS transformation

We now briefly describe how TRS transformation by template is formalized in our framework. We refer to [1] for all omitted definitions.

Let $\mathscr{P}$ be a set of pattern variables (disjoint from $\mathscr{F}$ and $\mathscr{V}$) where each pattern variable $p \in \mathscr{P}$ has its arity. A *pattern* is an unsorted term in $T(\mathscr{F} \cup \mathscr{P}, \mathscr{V})$ (for $f \in \mathscr{F}$, the arity is inherited from the sort specification). A *TRS 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 $\langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle$ of two TRS patterns $\mathcal{P}$, $\mathcal{P}'$ and a hypothesis $\mathcal{H}$.

To achieve the program transformation by templates, we need a mechanism to specify how a template is applied to a concrete many-sorted TRS. For this we use a notion of *term homomorphism*, a variation of tree homomorphism [2].

Definition 2 ([1]). *Let $\langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle$ be a template. A TRS $\mathcal{R}$* is transformed into *$\mathcal{R}'$ by $\langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle$ if there exists a term homomorphism φ such that $\mathcal{R} = \varphi(\mathcal{P})$ and $\mathcal{R}' = \varphi(\mathcal{P}')$.*

This definition is adapted to many-sorted TRS in the obvious way.

A matching algorithm to find all (most general) term homomorphisms φ satisfying $\mathcal{R} = \varphi(\mathcal{P})$ from a given TRS $\mathcal{R}$ and a TRS pattern $\mathcal{P}$ is presented in [1]. This algorithm is used to transform a many-sorted TRS $\mathcal{R}$ based on a template $\langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle$: for an input of a TRS $\mathcal{R}$ and a template $\langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle$, compute a term homomorphism φ from $\mathcal{R}$ and $\mathcal{P}$ satisfying $\mathcal{R} = \varphi(\mathcal{P})$ and then output the instantiation $\mathcal{R}' = \varphi(\mathcal{P}')$ (Figure 1).

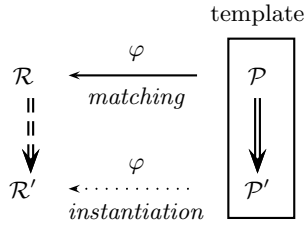

Fig. 1. TRS transformation

The hypothesis $\mathcal{H}$ is needed to discuss the correctness of the transformation.

Example 6. Let $\mathcal{R}$, $\mathcal{R}'$ be the TRSs in Example 1, and $\langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle$ the template given in Example 3. Then the following term homomorphism φ is used to transform $\mathcal{R}$ into $\mathcal{R}'$.

$$\varphi = \left\{ \begin{array}{ll} \mathsf{f} \mapsto \mathsf{sum}(\Box_1), & \mathsf{b} \mapsto 0, \\ \mathsf{g} \mapsto +(\Box_1, \Box_2), & \mathsf{c} \mapsto \Box_1{:}\Box_2, \\ \mathsf{f}_1 \mapsto \mathsf{sum1}(\Box_1, \Box_2), & \mathsf{d} \mapsto \mathsf{s}(\Box_2), \\ \mathsf{a} \mapsto [\,], & \mathsf{e} \mapsto \Box_1 \end{array} \right\}$$

Thus the TRS $\mathcal{R}$ is transformed into $\mathcal{R}'$ by $\langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle$.

3.3 Correctness of transformation

The correctness of a TRS transformation is formalized using the notion of equivalence of TRSs.

Definition 3. *Let $\mathscr{G}$ and $\mathscr{G}'$ be sets of function symbols such that $\mathscr{F}_{\mathrm{c}} \subseteq \mathscr{G}, \mathscr{G}' \subseteq \mathscr{F}$. Let $\mathcal{R}$ be a many-sorted TRS over $\mathscr{G}$ and $\mathcal{R}'$ a many-sorted TRS over $\mathscr{G}'$. A transformation from $\mathcal{R}$ to $\mathcal{R}'$ is* correct *if $\mathcal{R} \simeq_{\mathscr{G} \cap \mathscr{G}'} \mathcal{R}'$.*

We say $\langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle$ is developed if there exists an equivalent transformation [1] $\mathcal{P} \Rightarrow \mathcal{P}'$ under $\mathcal{H}$. The following is a sufficient condition to guarantee the correctness of the transformation from $\mathcal{R}$ to $\mathcal{R}'$ by a template $\langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle$ through a term homomorphism φ (Theorem 2 of [1]):

- $\mathcal{R}$ is a left-linear confluent constructor system,
- $\langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle$ is a developed template,
- φ is a CS-homomorphism,
- equations in $\varphi(\mathcal{H})$ are inductive consequences of $\mathcal{R}$ for $\mathscr{G}$, and

– $\mathcal{R}$ is sufficiently complete for $\mathscr{G}$,
– $\mathcal{R}'$ is sufficiently complete for $\mathscr{G}'$.

We note that many-sorted TRSs that model functional programs are usually left-linear constructor systems. We refer to [1] the notion of the developed template and how they are developed manually. RAPT assumes templates are developed beforehand and will be given as an input. All other conditions are verified by RAPT automatically. We note that not all conditions are decidable, and some are solved recursively by imposing a limitation and some are checked via (theoretically-proved) sufficient conditions. Since the basis of the correctness proof of [1] is the *inductionless induction*, confluence and sufficient completeness play essential roles to establish the result above.

4 Design of RAPT

RAPT transforms a many-sorted term rewriting system according to a specified program transformation template. Inputs of RAPT is a many-sorted TRS and a transformation template. The input TRS is specified by the following sections:

1. `FUNCTIONS` section which is a list of function symbols with sort declaration,
2. `RULES` section which is a list of rewrite rules over many-sorted terms.

The transformation template $\langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle$ is specified by the following sections:

1. `INPUT` section which is a list of rewrite rules of $\mathcal{P}$ over patterns,
2. `OUTPUT` section which is a list of rewrite rules of $\mathcal{P}'$ over patterns,
3. `HYPOTHESIS` section which is a list of equations of $\mathcal{H}$ over patterns.

Figure 2 shows the many-sorted TRS $\mathcal{R}_{sum}$ in Example 1 and the template $\langle \mathcal{P}$, $\mathcal{P}'$, $\mathcal{H} \rangle$ in Example 3 prepared as an input to RAPT: rules and sort declarations are separated by "`;`"; pattern variables are preceded by "`?`"; and to distinguish variables from constants, the latter are followed by "`()`" .

The first design choice of the system is to employ the *rewriting induction* [8] for inductive proving method which is used to verify whether the input TRS satisfies the hypothesis of transformation template. The second choice of the system is to restrict the input and output TRSs to be compatible with a reduction ordering. The termination of the input/output TRS is not a theoretical requirement. But to make the rewriting induction to work, we need a reduction ordering compatible with the input TRS. Also, once termination had been proved, there are effective procedures to prove confluence and sufficient completeness (for left-linear TRSs). Thus, by this choice, we can keep the design of the whole system very simple.

The TRS transformation and the verification of its correctness are conducted in RAPT by the following 6 phases:

1. Validation of Input TRS,
2. Precedence Detection,
3. Proving Confluence and Sufficient Completeness,
4. TRS Pattern Matching,
5. Verification of Hypothesis,
6. Validation of the Output TRS.

```
FUNCTIONS                              INPUT
 sum: List -> Nat;                      ?f(?a()) -> ?b();
 cons: Nat * List -> List;              ?f(?c(u,v)) -> ?g(?e(u),?f(v));
 nil: List;                             ?g(?b(),u) -> u;
 +: Nat * Nat -> Nat;                   ?g(?d(u,v),w) -> ?d(u,?g(v,w))
 s: Nat -> Nat;
 0: Nat                                OUTPUT
                                        ?f(u) -> ?f1(u,?b());
RULES                                   ?f1(?a(),u) -> u;
 sum(nil()) -> 0();                     ?f1(?c(u,v),w) -> ?f1(v,?g(w,?e(u)));
 sum(cons(x,ys)) -> +(x,sum(ys));       ?g(?b(),u) -> u;
 +(0(), x) -> x;                        ?g(?d(u,v),w) -> ?d(u,?g(v,w))
 +(s(x),y) -> s(+(x,y))
                                        HYPOTHESIS
                                        ?g(?b(),u) = ?g(u,?b());
                                        ?g(?g(u,v),w) = ?g(u,?g(v,w))
```

Fig. 2. Specification of input TRS and transformation template

In Figure 3, we describe dependencies between each phases. Solid arrows represent data flow and dotted arrows represent where informations obtained in each phase are used. If these 6 phases are successfully passed then RAPT produces output TRSs. The correctness of the transformation is guaranteed in the sense of Definition 3, provided that the template is developed. The following is the output of RAPT for the input of Figure 2:

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Output TRS(s)
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
{ [ sum(u) -> f1(u, 0()),
 f1(nil(), u) -> u,
 f1(cons(u, v), w) -> f1(v, +(w, u)),
 +(0(), u) -> u,
 +(s(v), w) -> s(+(v, w)) ]}
```

Note that, there may be more than one valid transformations; in that case, RAPT produces all valid output TRSs.

5 Inside of RAPT

In this section, we explain inside of the each phase of RAPT.

5.1 Validation of input TRS

In this phase, RAPT checks whether the input TRS is left-linear and well-typed, and from rewrite rules divides function symbols into defined function symbols and constructor symbols and checks whether the input TRS is a constructor system. The latter information will be used in Phases 3 and 4.

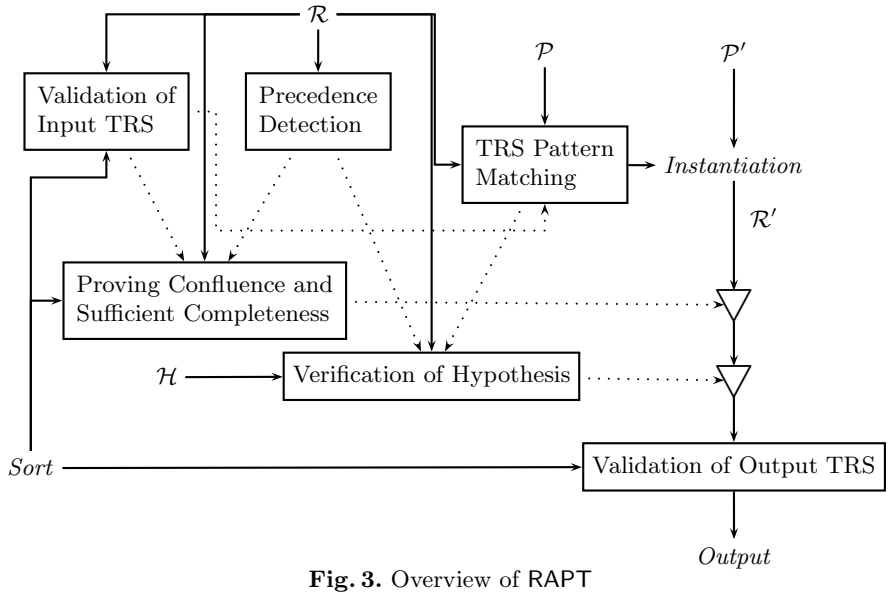

Fig. 3. Overview of RAPT

5.2 Precedence detection

In this phase, RAPT checks the input TRS is terminating by the lexicographic path ordering (LPO) [6] and (if it is the case) detects a precedence. We need not only to verify termination of the TRS but to find a reduction order used to prove inductive properties of the TRS. Although there may be a choice between which reduction orders to use, currently only the reduction order by the LPO is supported. The suitable precedence (if there exists one) for the LPO is computed based on the LPO constraint solving algorithm described in [4].

```
************************************************************
 Phase 2 (Precedence Detection)
************************************************************
++ SN Check..........
The set of precedences is
[{(cons > +), (cons > s), (cons > sum), (+ > s), (nil > 0)},
 {(sum > +), (sum > s), (+ > s), (nil > 0)},
 {(cons > +), (cons > s), (cons > sum), (+ > s), (cons > 0),
(sum > 0)},
 {(sum > +), (sum > s), (+ > s), (sum > 0)}]
O.K.
******** Phase 2 End ********
```

5.3 Proving confluence and sufficient completeness

In this phase, RAPT proves whether the input TRS is confluent and sufficiently complete.This makes use of the information of constructor symbols detected at Phase 1 and the fact that the input TRS is left-linear and terminating verified at Phases 1 and 2, respectively. For confluence, since termination of the TRS is guaranteed, it remains to check whether all critical pairs are joinable. For sufficient completeness, again since termination of the TRS is guaranteed, it suffices to check the quasi-reducibility of the TRS. We check the quasi-reducibility based

on the (many-sorted extension of) complement algorithm introduced in [7, 11] that computes the complement $C(\theta)$ of a substitution θ.

```
************************************************************
 Phase 3 (Proving Confluence and Sufficient Completeness)
************************************************************
++ CR Check..........
The set of critical pairs
is { }
                (* ..... omitted ..... *)
++ SC Check..........
++++++++++++++++++++++++++++++++++++++++
Checking whether T = { sum(y) }
is covered by S = { sum(cons(x, ys)), sum(nil()) }
++++++++++++++++++++++++++++++++++++++++
Unifiable terms of T and S are
sum(y) and sum(cons(x, ys))
M.g.u. is
[ y := cons(x, ys) ]   (* = θ *)
Checking
{ sum(y) }
Complement substitutions of
[ y := cons(x, ys) ]
are
{ [ y := nil() ] }   (* = C(θ) *)
                (* ..... omitted ..... *)
O.K.
******** Phase 3 End ********
```

5.4 TRS pattern matching

In this phase, RAPT finds a combination of rewrite rules to apply the transformation and the term homomorphism which instantiates the input pattern TRS to these rewrite rules. Using information of function symbols detected in Phase 1, RAPT also checks this term homomorphism is a CS-homomorphism.

This phase is based on the matching algorithm **Match** described in [1]. First, we reduce the matching problem of TRS and TRS pattern to matching problem of a set of pairs of term and pattern. In RAPT, pattern matching of rewrite rules are carried out in order, and use the information of matching solutions to limit next rewrite rules to perform the pattern match. Since solving the patten matching of main function usually gives information which subfunctions are used in sequel, this heuristics seems to perform the TRS matching relatively well.

```
************************************************************
 Phase 4 (TRS Pattern Matching)
************************************************************
++ TRS Match..........
(* non-deterministic choice 1 *)
Matching
 f(c(u, v)) -> g(u, f(v))
and
 sum(cons(x, ys)) -> +(x, sum(ys))
```

```
Solutions are
{g := +([]_1, []_2),
 c := cons([]_1, []_2),
 f := sum([]_1)}
Matching
 sum(a()) -> b()   (* the partial instantiation of f(a()) -> b() *)
and
 sum(nil()) -> 0()
Solutions are
{b := 0,
 a := nil}
(* non-deterministic choice 2 *)
Matching
 f(c(u, v)) -> g(u, f(v))
and
 sum(nil()) -> 0()
Solutions are
( no solutions )
(* non-deterministic choice 1-1 *)
Matching
 +(0(), u) -> u   (* the instantiation of g(b(),u) -> u *)
and
 +(0(), x) -> x
Solutions are
{ }   (* one identity term homomorphism *)
Matching
 +(d(u, v), w) -> d(u, +(v, w))
   (* the partial instantiation of g(d(u,v),w) -> d(u,g(v,w)) *)
and
 +(s(x), y) -> s(+(x, y))
Solutions are
{ d := s([]_2)}
(* non-deterministic choice 1-2 *)
Matching
 +(0(), u) -> u   (* the instantiation of g(b(),u) -> u *)
and
 +(s(x), y) -> s(+(x, y))
Solutions are
( no solutions )
O.K.
The set of solutions are
{
([],   (* unused part of TRS *)
{d := s([]_2),   (* CS-homomorphism *)
 b := 0,
 a := nil,
 g := +([]_1, []_2),
 c := cons([]_1, []_2),
 f := sum([]_1)})
}
******** Phase 4 End ********
```

5.5 Inductive theorem proving

In this phase, RAPT checks whether the input TRS satisfies the hypothesis of the template. This is done by (1) instantiating the hypothesis through the term

homomorphism found at Phase 4 and (2) proving they are inductive consequences of the input TRS, using rewriting induction. The latter uses LPO with the precedence detected at Phase 2.

The algorithm of the rewriting induction (which consists of inference rules **Simplify**, **Delete**, **Expand**) is due to [8]. Because the quasi-reducibility of the input TRS have been shown already, $>$-cover set of substitutions [8] is obtained just by selecting a subterm that is unifiable with the left-hand side of some rewrite rules.

We have used a standard heuristics to choose which inference rules to apply. That is, we first apply **Simplify** as many as possible, then apply **Delete** to delete all trivial equations, and then apply **Expand** once, and repeat this process. In **Expand**, a subterm to expand is selected via the outermost leftmost strategy.

Procedure of the rewriting induction (1) stops with success, (2) stops with failure, or (3) diverges. In RAPT, we made a bound on the number of application of **Expand** to detect the divergence.

```
************************************************************
 Phase 5 (Verification of Hypothesis)
************************************************************
++ Hypothesis Check..........
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ Inductive Theorem Proving by Rewriting Induction +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                (* ..... omitted ..... *)
Start 2 round
=============
(simplify) hs:
[ +(u, 0()) -> u ]   (* induction hypothesis *)
(simplify) old es:
[ +(+(u, v), w) = +(u, +(v, w)),
 0() = 0(),
 s(+(x, 0())) = s(x) ]
(simplify) new es:   (* lemmas to prove are simplified *)
[ +(+(u, v), w) = +(u, +(v, w)),
 0() = 0(),
 s(x) = s(x) ]
+++++
(delete) new es:   (* delete trivial equations *)
[ +(+(u, v), w) = +(u, +(v, w)) ]
+++++
(expand) e = e': +(+(u, v), w) = +(u, +(v, w))
(expand) E U E':   (* new lemmas to prove *)
[ +(x, w) = +(0(), +(x, w)),
 +(s(+(x, y)), w) = +(s(x), +(y, w)) ]
(expand) new H:   (* induction hypothesis added *)
[ +(+(u, v), w) -> +(u, +(v, w)),
 +(u, 0()) -> u ]
                (* ..... omitted ..... *)
SUCCESS in 3 round
******** Phase 5 End ********
```

5.6 Validation of output TRS

In this phase, RAPT checks whether the output TRS is (1) terminating, (2) left-linear, (3) type consistent, and (4) sufficiently complete. In (3), because the

pattern TRS $\mathcal{P}'$ for the output may contain a pattern variable not occurring in the pattern TRS $\mathcal{P}$ for the input, types may be unknown for some of function symbols in $\mathcal{R}'$. Therefore, we need to infer the type information together with the type consistency check. (4) is proved based on the fact the output TRS is terminating which is verified at (1) using LPO.

6 A guide for constructing transformation templates

In our framework, transformation templates should be developed manually. (Note that only developed templates guarantee the the correctness of transformations.) Among developed templates for the similar transformations, some template may transform more programs than another. Clearly, it is better to use more general templates to reduce the computing time for finding appropriate templates, especially when many template candidates are prepared. Below we describe how a more general template can be developed from a template for the similar transformations.

Let's look back examples in Section 2. We have constructed the template $\langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle$ by generalizing similar program transformations for $\mathcal{R}_{sum}$ and $\mathcal{R}_{cat}$. In these transformations, both $\mathcal{R}_{sum}$ and $\mathcal{R}'_{sum}$ contain rewrite rules for $+$, and both $\mathcal{R}_{cat}$ and $\mathcal{R}'_{cat}$ contains rewrite rules for app. Naturally, the TRS patterns $\mathcal{P}$ and $\mathcal{P}'$ contain the same rules; let $\mathcal{P}_{com}$ be the common part of $\mathcal{P}$ and $\mathcal{P}'$. Since these unchanged rules are not necessary to describe transformations, it may seem that $\mathcal{P}_{com}$ can be removed from templates. Moreover, by removing $\mathcal{P}_{com}$ from $\mathcal{P}$ and $\mathcal{P}'$, some more TRS transformations become possible:

Example 7. The following TRS $\mathcal{R}_{rev}$ specifies a program which computes the reverse of input lists:

$$\mathcal{R}_{rev} \begin{cases} \mathsf{rev}([\,]) & \to [\,] \\ \mathsf{rev}(x : xs) & \to \mathsf{app}(\mathsf{rev}(xs), x : [\,]) \\ \mathsf{app}([\,], ys) & \to ys \\ \mathsf{app}(x : xs, ys) & \to x : \mathsf{app}(xs, ys) \end{cases}$$

The TRS pattern $\mathcal{P}$ does not match $\mathcal{R}_{rev}$ even though the TRS pattern $\mathcal{P} \setminus \mathcal{P}_{com}$ matches the first two rules of $\mathcal{R}_{rev}$. Hence, the template $\langle \mathcal{P}_1, \mathcal{P}'_1, \mathcal{H} \rangle = \langle \mathcal{P} \setminus \mathcal{P}_{com}, \mathcal{P}' \setminus \mathcal{P}_{com}, \mathcal{H} \rangle$ can be used to transform $\mathcal{R}_{rev}$ into the following $\mathcal{R}'_{rev}$:

$$\mathcal{R}'_{rev} \begin{cases} \mathsf{rev}(xs) & \to \mathsf{rev1}(xs, [\,]) \\ \mathsf{rev1}([\,], ys) & \to ys \\ \mathsf{rev1}(x : xs, ys) & \to \mathsf{rev1}(xs, x : ys) \\ \mathsf{app}([\,], ys) & \to ys \\ \mathsf{app}(x : xs, ys) & \to x : \mathsf{app}(xs, ys) \end{cases}$$

However, remind that templates have to be *developed* to guarantee the correctness of the transformation. In fact, the template $\langle \mathcal{P} \setminus \mathcal{P}_{com}, \mathcal{P}' \setminus \mathcal{P}_{com}, \mathcal{H} \rangle$ is not developed and it may produce incorrect transformations. The situation is that rules of $\mathcal{P}_{com}$ are required to show that the template $\langle \mathcal{P}, \mathcal{P}', \mathcal{H} \rangle$ is developed.

How can we develop suitable transformation templates for $\mathcal{R}_{rev}$? In fact, this can be done by moving the common part $\mathcal{P}_{com}$ into hypothesis like this:

$$\tilde{\mathcal{P}} \begin{cases} \mathsf{f}(\mathsf{a}) & \to \mathsf{b} \\ \mathsf{f}(\mathsf{c}(u,v)) & \to \mathsf{g}(\mathsf{e}(u), \mathsf{f}(v)) \end{cases}$$

$$\tilde{\mathcal{P}}' \begin{cases} \mathsf{f}(u) & \to \mathsf{f}_1(u, \mathsf{b}) \\ \mathsf{f}_1(\mathsf{a}, u) & \to u \\ \mathsf{f}_1(\mathsf{c}(u,v), w) & \to \mathsf{f}_1(v, \mathsf{g}(w, \mathsf{e}(u))) \end{cases}$$

$$\tilde{\mathcal{H}} \begin{cases} \mathsf{g}(u, \mathsf{b}) & \approx u \\ \mathsf{g}(\mathsf{b}, u) & \approx u \\ \mathsf{g}(\mathsf{g}(u,v), w) & \approx \mathsf{g}(u, \mathsf{g}(v,w)) \end{cases}$$

Then we can easily show that there exists an equivalent transformation [1] from $\mathcal{P}$ to $\mathcal{P}'$ under $\mathcal{H}$ (See appendix). Thus, the template $\langle \tilde{\mathcal{P}}, \tilde{\mathcal{P}}', \tilde{\mathcal{H}} \rangle$ is developed so that the correctness of transformations by $\langle \tilde{\mathcal{P}}, \tilde{\mathcal{P}}', \tilde{\mathcal{H}} \rangle$ is guaranteed.

Using the template $\langle \tilde{\mathcal{P}}, \tilde{\mathcal{P}}', \tilde{\mathcal{H}} \rangle$, $\mathcal{R}_{sum}$ and $\mathcal{R}_{cat}$ are transformed into $\mathcal{R}'_{sum}$ and $\mathcal{R}'_{cat}$, respectively. The TRS $\mathcal{R}_{rev}$ is transformed into the following $\mathcal{R}''_{rev}$:

$$\mathcal{R}''_{rev} \begin{cases} \mathsf{rev}(xs) & \to \mathsf{rev1}(xs, [\,]) \\ \mathsf{rev1}([\,], ys) & \to ys \\ \mathsf{rev1}(x : xs, ys) & \to \mathsf{rev1}(xs, \mathsf{app}(x : [\,], ys)) \\ \mathsf{app}([\,], ys) & \to ys \\ \mathsf{app}(x : xs, ys) & \to x : \mathsf{app}(xs, ys) \end{cases}$$

Here, the right-hand side of the third rule of $\mathcal{R}''_{rev}$ is not a normal form. By normalizing such terms, one obtain the desired $\mathcal{R}'_{rev}$.

7 Conclusion

This paper describes the system RAPT, which implements the program transformation based on term rewriting introduced in [1]. RAPT transforms a term rewriting system according to a specified program transformation template and automatically verifies correctness of the transformation. We have described the design of RAPT and explain algorithms and heuristics employed in each phase of RAPT. We also presented a way to polish transformation templates.

Through our experiences, the computation time of RAPT is less than 100 msec in small examples as presented in this paper. The source code of RAPT consists of about 5,000 lines and is written in the Standard ML of New Jersey [10], which is an implementation of the strongly typed functional programming language ML. We are now testing other kinds of program transformation by templates such as fusion, tupling, etc. on RAPT.

Another implementation of program transformation by template is MAG system which is based on lambda calculus [3, 9]. In MAG system, the correctness of transformation is based on Huet and Lang's framework [5]. A difference between MAG and RAPT lies on the approach to the verification of hypothesis. In MAG system, users are usually need to verify the hypothesis by explicit induction. In contrast to that, RAPT proves the hypothesis automatically without help of users. To our knowledge, program transformation systems based on template in the literature rarely corporate with automated theorem proving techniques in the verification of hypothesis. RAPT shows an interesting corporation of program transformation techniques and automated theorem proving techniques.

Acknowledgments

Thanks are due to anonymous referees for useful comments and advices. This work was partially supported by a grant from Japan Society for the Promotion of Science, No. 14580357 and grants from Ministry of Education, Culture, Sports, Science and Technology, Nos. 16016202 and 17700002.

References

1. Y. Chiba, T. Aoto, and Y. Toyama. Program transformation by templates based on term rewriting. In *Proceedings of the 7th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2005)*, pages 59–69. ACM Press, 2005.
2. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 1997. `http://www.grappa.univ-lille3.fr/tata`.
3. O. de Moor and G. Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269:135–162, 2001.
4. N. Hirokawa and A. Middeldorp. Tsukuba termination tool. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *LNCS*, pages 311–320. Springer-Verlag, 2003.
5. G. Huet and B. Lang. Proving and applying program transformations expressed with second order patterns. *Acta Informatica*, 11:31–55, 1978.
6. S. Kamin and J.-J. Lévy. Two generalizations of the recursive path ordering. Unpublished manuscript, University of Illinois, 1980.
7. A. Lazrek, P. Lescanne, and J. J. Thiel. Tools for proving inductive equalities, relative completeness, and ω-completeness. *Information and Computation*, 84:47–70, 1990.
8. U. S. Reddy. Term rewriting induction. In *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *LNAI*, pages 162–177, 1990.
9. G. Sittampalam. *Higher-Order Matching for Program Transformation*. PhD thesis, Magdalen College, 2001.
10. *Standard ML of New Jersey*. `http://www.smlnj.org/`.
11. J. J. Thiel. Stop loosing sleep over incomplete data type specifications. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 76–82, 1984.

A Example of equivalent transformation

Let $\langle \tilde{\mathcal{P}}, \tilde{\mathcal{P}}', \tilde{\mathcal{H}} \rangle$ be the template appears in Section 6. We demonstrate how to develop the template $\langle \tilde{\mathcal{P}}, \tilde{\mathcal{P}}', \tilde{\mathcal{H}} \rangle$.

1. Let $\mathcal{P}_0 = \mathcal{P}$.
2. Let

$$\mathcal{P}_1 = \mathcal{P}_0 \cup \{\mathsf{f}_1(u, v) \to \mathsf{g}(v, \mathsf{f}(u))\}$$

   Here, f_1 is a fresh function symbol. Then, $\mathcal{P}_0 \Rightarrow \mathcal{P}_1$ by the Introduction rule.
3. Let $\mathcal{P}_2 = \mathcal{P}_1 \cup \{\mathsf{f}(u) \to \mathsf{f}_1(u, \mathsf{b})\}$. Because

$$\begin{aligned} \mathsf{f}(u) &\leftrightarrow_{\tilde{\mathcal{H}}} \mathsf{g}(\mathsf{b}, \mathsf{f}(u)) \\ &\leftarrow_{\mathcal{P}_1} \mathsf{f}_1(u, \mathsf{b}), \end{aligned}$$

   we have $\mathcal{P}_1 \Rightarrow \mathcal{P}_2$ by the Addition rule.
4. Let $\mathcal{P}_3 = \mathcal{P}_2 \cup \{\mathsf{f}_1(\mathsf{a}, u) \to u\}$. Because

$$\begin{aligned} \mathsf{f}_1(\mathsf{a}, u) &\to_{\mathcal{P}_2} \mathsf{g}(u, \mathsf{f}(\mathsf{a})) \\ &\to_{\mathcal{P}_2} \mathsf{g}(u, \mathsf{b}) \\ &\leftrightarrow_{\tilde{\mathcal{H}}} u, \end{aligned}$$

   we have $\mathcal{P}_2 \Rightarrow \mathcal{P}_3$ by the Addition rule.
5. Let $\mathcal{P}_4 = \mathcal{P}_3 \cup \{\mathsf{f}_1(\mathsf{c}(u, v), w) \to \mathsf{f}_1(v, \mathsf{g}(w, u))\}$. Because

$$\begin{aligned} \mathsf{f}_1(\mathsf{c}(u, v), w) &\to_{\mathcal{P}_3} \mathsf{g}(w, \mathsf{f}(\mathsf{c}(u, v))) \\ &\to_{\mathcal{P}_3} \mathsf{g}(w, \mathsf{g}(u, \mathsf{f}(v))) \\ &\leftrightarrow_{\tilde{\mathcal{H}}} \mathsf{g}(\mathsf{g}(w, u), \mathsf{f}(v)) \\ &\leftarrow_{\mathcal{P}_3} \mathsf{f}_1(v, \mathsf{g}(w, u)), \end{aligned}$$

   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 $\tilde{\mathcal{P}}'$.

Thus, the template $\langle \tilde{\mathcal{P}}, \tilde{\mathcal{P}}', \tilde{\mathcal{H}} \rangle$ is developed so that the correctness of transformations by $\langle \tilde{\mathcal{P}}, \tilde{\mathcal{P}}', \tilde{\mathcal{H}} \rangle$ is guaranteed.