Completion of Unraveled Term Rewriting Systems toward Program Inversion of Injective Functions*

Naoki Nishida and Masahiko Sakai

Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan
nishida@is.nagoya-u.ac.jp sakai@is.nagoya-u.ac.jp

Abstract. Given a constructor term rewriting system defining injective functions, the inversion compiler proposed by Nishida, Sakai and Sakabe generates a confluent conditional term rewriting system, and unravels the conditional system into an unconditional term rewriting system. In general, the unconditional system is not confluent and thus not computationally equivalent to the conditional system. In this paper, we slightly modify Knuth-Bendix completion procedure in order to transform the unraveled systems into ones that are computationally equivalent to the corresponding conditional systems if the procedure halts successfully. We also show a way to apply the procedure to rewrite systems evaluated by the innermost strategy, and apply it to examples of program inversion.

1 Introduction

Given a constructor TRS, the inversion compiler proposed in [25, 24] first generates an inverse CTRS as an intermediate result, and then transforms the CTRS into a TRS that is equivalent to the CTRS with respect to inverse computation. The first phase of the compiler is local inversion. For every constructor TRS, the first phase generates an inverse CTRS that completely represents the inverse relation of the reduction relation represented by the constructor TRS. The second phase employs (a variant of) Ohlebusch’s unraveling [26]. Unravelings are transformations based on Marchiori’s approach [15], that transform CTRSs into TRSs. Note that we call all variants of Marchiori’s unravelings unravelings because they satisfy the condition [15, 16] of being unravelings. Unfortunately, the compiler cannot always generate TRSs that are computationally equivalent to the intermediate CTRSs due to a character of unravelings. The character is that computation by the generated TRSs sometimes produces some garbage normal forms that represent dead ends of wrong choices at inverse-computation branches. Note that it is decidable whether or not a normal form is a solution or garbage. This problem arises even if functions are injective. Thus, the inversion compiler is less applicable to practical first-order functional programs. It

* This work is partly supported by MEXT. KAKENHI #18500011, and Kayamori Foundation of Informational Science Advancement.
is easy to translate the functional programs to constructor TRSs, but difficult to translate the resultant TRSs of the compiler back into functional programs. For example, consider the following functional program in Standard ML where $\text{Snoc}(xs, y)$ produces the list obtained from $xs$ by adding $y$ as the last element:

$$\text{fun Snoc( [], y ) = [y]}$$
$$\mid \text{Snoc( (x::xs), y ) = x :: Snoc( xs, y );}$$

We can easily translate the above program into the following constructor TRS:

$$R_1 = \begin{cases} 
\text{Snoc(nil, y) } \rightarrow \text{cons(y, nil)} \\
\text{Snoc(cons(x, xs), y) } \rightarrow \text{cons(x, Snoc(xs, y))}
\end{cases}$$

The compiler inverts $R_1$ into the following CTRS in the first phase:

$$\text{Inv}(R_1) = \begin{cases} 
\text{InvSnoc(cons(y, nil)) } \rightarrow \langle\text{nil, y}\rangle \\
\text{InvSnoc(cons(x, ys)) } \rightarrow \langle\text{cons(x, xs), y}\rangle \leftarrow \text{InvSnoc(ys) } \rightarrow \langle\text{x, y}\rangle
\end{cases}$$

Here, we write the tuple of $n$ terms $t_1, \ldots, t_n$ as $\langle t_1, \ldots, t_n \rangle$ that can be represented as terms by introducing an $n$-ary constructor. To simplify discussions, we omit describing special rules, which are in the form of $\text{invF(F}(x_1, \ldots, x_n)) \rightarrow \langle x_1, \ldots, x_n \rangle$ [24, 25] because they are useless for our setting and not harmful for the result in this paper. The special rules are necessary for inverse computation of term rewriting systems but not for inverse computation of call-by-value systems. The compiler unravels the above CTRS into the following TRS in the second phase:

$$\text{U}(\text{Inv}(R_1)) = \begin{cases} 
\text{InvSnoc(cons(y, nil)) } \rightarrow \langle\text{nil, y}\rangle \\
\text{InvSnoc(cons(x, ys)) } \rightarrow U_1(\text{InvSnoc(ys)}, x, ys) \\
U_1(\langle x, y \rangle, x, ys) \rightarrow \langle\text{cons(x, xs), y}\rangle
\end{cases}$$

Here, $[t_1, t_2, \ldots, t_n]$ abbreviates the list $\text{cons}(t_1, \text{cons}(t_2, \cdots, \text{cons}(t_n, \text{nil})\cdots))$. The term $\text{Snoc}(a, b, c)$ has a unique normal form $[a, b, c]$ but $\text{InvSnoc}(a, b, c)$ has two normal forms, a solution $\langle[a, b, c]\rangle$ of inverse computation and a garbage normal form $U_1(U_1(\text{InvSnoc(nil)}, c, nil), b, [c]), a, [b, c])$. To fix this problem, it has been shown in [22] that the transformation in [31] is suitable as the second phase of the compiler, in the sense of producing convergent systems. By the transformation, we obtain the following convergent TRS instead of $\text{U}(\text{Inv}(R_1))$:

$$\text{InvSnoc}(\langle x, y, \perp \rangle, z) \rightarrow \langle\text{nil, y}\rangle$$
$$\text{InvSnoc}(\langle x, y \rangle, \perp) \rightarrow \text{InvSnoc}(\langle x, ys \rangle, \{\text{InvSnoc}(ys, \perp)\})$$
$$\text{InvSnoc}(\langle y, nil \rangle, z) \rightarrow \{\text{nil, y}\}$$
$$\text{InvSnoc}(\langle x, ys \rangle, \{\langle x, y \rangle\}) \rightarrow \langle\text{cons(x, xs), y}\rangle$$
$$\{\langle x \rangle\} \rightarrow \{x\}$$
$$\text{InvSnoc}(\langle x, zs \rangle) \rightarrow \langle\text{InvSnoc}(xs, \perp)\rangle$$
$$\cup\{c(x_1, \ldots, x_n) \rightarrow \langle\text{c(x_1, \ldots, x_n)\rangle} \mid c \text{ is a constructor, } n \geq 1 \}$$

where $\{\}$ and $\perp$ are special function symbols not in the original signature. In this system, the term $\text{InvSnoc}([a, b, c], \perp)$ has a unique normal form $\langle\langle a, b, c\rangle\rangle$. 

2
However, it is difficult to translate the convergent TRS into a functional program. In this example, it appears to be easy to translate from the CTRS $\text{Inv}(R_1)$ into a functional program directly because we can easily determine an appropriate priority of conditional rules in $\text{U}(\text{Inv}(R_1))$. However, such a direct translation is difficult in general because we cannot decide which rules have priority of the application to terms. This problem cannot be solved by the restricted compiler in [1] because $R_1$ is out of the scope of [1].

Roughly speaking, by adding the rule $U_1(\text{InvSnoc}(\text{nil}), x, \text{nil}) \rightarrow \langle \text{nil}, x \rangle$ that is obtained from the critical pair between the first and second rules of $\text{U}(\text{Inv}(R_1))$, the garbage normal form of $\text{InvSnoc}([a, b, c])$ can be reduced to the solution. This added rule provides a path from the wrong branch of inverse computation to the correct branch. Due to this rule, the new TRS is confluent. Therefore, completion is expected to solve the inconfuence of TRSs obtained by the inversion compiler.

In this paper, we present a framework that makes unravelings useful not only in analyzing properties but also in modifying programs (unraveled TRSs, especially inverse programs), by applying the Knuth-Bendix completion procedure. We also show that some CTRSs can be well translated into functional programs in the framework. More precisely, we illustrate all of the following:

- under the call-by-value evaluation of operationally terminating deterministic CTRSs, simulation-completeness on the innermost reduction is preserved by Ohlebusch’s unraveling;
- the completion procedure can modify the unraveled TRSs (evaluated by the innermost reduction) of CTRSs into (innermost-)convergent TRSs that are computationally equivalent to the CTRSs if the procedure halts successfully;
- to deal with TRSs whose termination cannot be proved by any LPO, we employ the completion with termination provers, following the approach in [34].

As a practical example, we apply the completion procedure to the unraveled TRSs obtained by the inversion compiler [25] from injective functions, and show that translating the TRSs obtained by the procedure into functional programs is not difficult. Note that we do not consider sorts; however, the framework in this paper is easily extended to many-sorted systems.

This paper is organized as follows. In Section 2, we provide basic notations on term rewriting. In Section 3, we show that applying the completion procedure to the unraveled TRSs of CTRSs produces convergent TRSs which are computationally equivalent to the CTRSs. In Section 4, we apply the unraveling and completion to the CTRSs generated by the partial inversion compiler from injective functions. In Section 5, we describe some related work. In Section 6, we conclude this paper.

2 Preliminaries

Here, we will review the following basic notations of term rewriting [3, 27].
Throughout this paper, we use $\mathcal{V}$ as a countably infinite set of variables. The set of all terms over a signature $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. The set of all variables appearing in either of terms $t_1, \ldots, t_n$ is represented by $\text{Var}(t_1, \ldots, t_n)$. The identity of terms $s$ and $t$ is denoted by $s \equiv t$. For a term $t$ and a position $p$ of $t$, the notation $t|_p$ represents the subterm of $t$ at position $p$. The function symbol at the root position $\varepsilon$ of $t$ is denoted by $\text{root}(t)$. The notation $C[t_1, \ldots, t_n]|_{p_1, \ldots, p_n}$ represents the term obtained by replacing $\Box$ at position $p_i$ of an $n$-hole context $C$ with term $t_i$ for $1 \leq i \leq n$. The domain and range of a substitution $\sigma$ are denoted by $\text{Dom}(\sigma)$ and $\text{Ran}(\sigma)$, respectively. The application $\sigma(t)$ of substitution $\sigma$ to $t$ is abbreviated to $t\sigma$.

An (oriented) conditional rewrite rule over $\mathcal{F}$ is a triple $(l, r, c)$, denoted by $l \rightarrow r \leftarrow c$, such that $l$ is a non-variable term in $\mathcal{T}(\mathcal{F}, \mathcal{V})$, $r$ is a term in $\mathcal{T}(\mathcal{F}, \mathcal{V})$, and $c$ is of form of $s_1 \rightarrow t_1 \land \cdots \land s_n \rightarrow t_n$ ($n \geq 0$) of terms $s_i$ and $t_i$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. In particular, the conditional rewrite rule $l \rightarrow r \leftarrow c$ is said to be a (unconditional) rewrite rule if $n = 0$, and we may abbreviate it to $l \rightarrow r$. We sometimes attach a unique label $\rho$ to a rule $l \rightarrow r \leftarrow c$ by denoting $\rho : l \rightarrow r \leftarrow c$, and we use the label to refer to the rule. To simplify notations, we may write labels instead of the corresponding rules. Let $R$ be a finite set of conditional rewrite rules over $\mathcal{F}$. The rewrite relation of $R$ is denoted by $\rightarrow_R$. To specify the applied position $p$ and rule $\rho$, we write $\rightarrow^p_R$ or $\rightarrow^{[p, \rho]}_R$. An (oriented) conditional rewriting system (CRS, for short) over a signature $\mathcal{F}$ is a finite set of conditional rewrite rules over $\mathcal{F}$. A conditional rewrite rule $\rho : l \rightarrow r \leftarrow s_i$ is called deterministic if $\text{Var}(r) \subseteq \text{Var}(l, t_1, \ldots, t_k)$ and $\text{Var}(s_i) \subseteq \text{Var}(l, t_1, \ldots, t_{i-1})$ for $1 \leq i \leq k$. The CRS $R$ is called a deterministic CRS (a DCRS for short) if all rules in $R$ are deterministic. Operational termination of DCRSs is such that no infinite reductions exist in existing rewrite engines [14].

We denote the innermost reduction of an operationally terminating DCRS $R$ by $\rightarrow^\ast_R$. Note that the innermost reduction is not well-defined for every CRS [8]. However, the innermost reduction of operationally terminating CRSs is well-defined.

Throughout this paper, we assume that a signature $\mathcal{F}$ consists of a set $\mathcal{D}$ of defined symbols and a set $\mathcal{C}$ of constructors: $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$. Let $R$ be a CRS over $\mathcal{F}$. The sets $\mathcal{D}_R$ and $\mathcal{C}_R$ of all defined symbols and all constructors of $R$ are defined as $\mathcal{D}_R = \{ \text{root}(l) \mid l \rightarrow r \leftarrow c \in R \}$ and $\mathcal{C}_R = \mathcal{F} \setminus \mathcal{D}_R$, respectively. We suppose that $\mathcal{D}_R \subseteq \mathcal{D}$ and $\mathcal{C}_R \subseteq \mathcal{C}$. Terms in $\mathcal{T}(\mathcal{C}, \mathcal{V})$ are called constructor terms. The CRS $R$ is called a constructor system if every rule $f(t_1, \ldots, t_n) \rightarrow r \leftarrow c$ in $R$ satisfies $\{ t_1, \ldots, t_n \} \subseteq \mathcal{T}(\mathcal{C}, \mathcal{V})$.

We use the notion of context-sensitive reduction in [13]. The context-sensitive reduction of the context-sensitive TRS $(R, \mu)$ of a CRS $R$ and a replacement map $\mu$ is denoted by $\rightarrow^{(R, \mu)}$. The innermost reduction of $\rightarrow^{(R, \mu)}$ is denoted by $\rightarrow^{\ast(\mu)}$.

Let $t_i \rightarrow r_i$ ($i = 1, 2$) be two rules whose variables have been renamed such that $\text{Var}(l_1, r_1) \cap \text{Var}(l_2, r_2) = \emptyset$. Let $p$ be a position in $l_1$ such that $l_1|_p$ is not a variable and let $\theta$ be a most general unifier of $l_1|_p$ and $l_2$. This determines a critical pair $(r_1\theta, (l_1\theta)[r_2\theta]|_p)$. If $p = \varepsilon$, then the critical pair is called an overlay.
If two rules give rise to a critical pair, we say that they overlap. We denote the set of critical pairs constructed by rules in a TRS $R$ by $\text{CP}(R)$. We also denote the set of critical pairs between rules in $R$ and another TRS $R'$ by $\text{CP}(R,R')$. Moreover, $\text{CP}_e(R)$ denotes the set of overlaps of $R$.

Let $R$ and $R'$ be CTRSs such that normal forms are computable, and $T$ be a set of terms. Roughly speaking, $R'$ is computationally equivalent to $R$ with respect to $T$ if there exist mappings $\phi$ and $\psi$ such that if $R$ terminates on a term $s \in T$ admitting a unique normal form $t$, then $R'$ also terminates on $\phi(s)$ and for any of its normal forms $t'$, we have $\psi(t') = t$ [31]. In this paper, we assume that $\phi$ and $\psi$ are the identity mappings.

Let $\rightarrow$ and $\Rightarrow$ two binary relations on terms, and $T'$ and $T''$ be sets of terms. We say that $\rightarrow = \Rightarrow$ in $T' \times T''$ if $\rightarrow \cap (T' \times T'') = \Rightarrow \cap (T' \times T'')$. Especially, we say that $\rightarrow = \Rightarrow$ in $T'$ if $T' = T''$.

### 3 Completion to Unraveled TRSs

In this section, we show that applying the completion procedure to the unraveled TRSs of CTRSs produces convergent TRSs that are computationally equivalent to the CTRSs. To adapt to call-by-value computation, we show simulation-completeness of the unraveling for DCTRSs with respect to innermost reduction.

#### 3.1 Unraveling for DCTRSs

We first give the definition of Ohlebusch’s unraveling [26]. Here, given a finite set $X$ of variables, we denote by $\overline{X}$ the sequence of variables in $X$ without repetitions (in some fixed order).

**Definition 1 ([26]).** Let $R$ be a DCTRS over a signature $\mathcal{F}$. For every conditional rewrite rule $\rho : l \rightarrow r \Leftarrow s_1 \rightarrow t_1 \land \ldots \land s_k \rightarrow t_k$, let $|\rho|$ denote the number of conditions in $\rho$ (that is, $|\rho| = k$). For every conditional rule $\rho \in R$, we prepare $k$ “fresh” function symbols $U_1^\rho, \ldots, U_{|\rho|}^\rho$, called $U$ symbols, in the transformation. Here, the word “fresh” means that every $U_i^\rho$ is not in $\mathcal{F}$. We transform $\rho$ into a set $\mathcal{U}(\rho)$ of $k + 1$ unconditional rewrite rules as follows:

$$\mathcal{U}(\rho) = \left\{ l \rightarrow U_1^\rho(s_1, \overline{X_1}), \ U_2^\rho(t_1, \overline{X_1}) \rightarrow U_2^\rho(s_2, \overline{X_2}), \ldots, \ U_{|\rho|}^\rho(t_k, \overline{X_k}) \rightarrow r \right\}$$

where $X_i = \text{Var}(l, t_1, \ldots, t_{i-1})$. The system $\mathcal{U}(R) = \bigcup_{\rho \in R} \mathcal{U}(\rho)$ is an unconditional TRS over the extended signature $\mathcal{F}_U = \mathcal{F} \cup \{ U_i^\rho \mid \rho \in R, 1 \leq i \leq |\rho| \}$.

An unraveling $U$ is simulation-complete for a DCTRS $R$ over a signature $\mathcal{F}$ if “for all terms $s$ and $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $s \sim_R t$ if and only if $s \sim_{U(R)} t$. Roughly speaking, computational equivalence is equivalent to the combination of simulation-completeness and normal-form uniqueness. The unraveling $U$ is not simulation-complete for every DCTRS [27]. This is because the unraveled TRSs of CTRSs are approximations of the CTRSs. In [15], we can find a counterexample against
the simulation-completeness of unravelings. To avoid this difficulty on non-“simulation-completeness” of \( U \), a restriction to the rewrite relations of the unraveled TRSs is shown in [30], which is done by the following context-sensitive condition the replacement map \( \mu \) such that \( \mu(U^R) = \{ 1 \} \) for every \( U^R \) in Definition 1. We denote the context-sensitive TRS \((U(R), \mu)\) by \( U_{cs}(R) \). We consider \( U_{cs} \) as an unraveling from CTRSs to context-sensitive TRSs.

**Theorem 2 (simulation-completeness [30]).** For every DCTRS \( R \) over a signature \( F \), \( U_{cs} \) is simulation-complete, that is, \( \rightarrow^*_{R} = \rightarrow^*_{U_{cs}(R)} \) in \( T(F, V) \).

### 3.2 Call-by-Value Evaluation

To adapt computation of DCTRSs to call-by-value evaluation of functional programs, we define an innermost-like reduction of DCTRSs. This notion removes the context-sensitivity for simulation-completeness from the corresponding reduction.

For a binary relation \( \rightarrow \) on terms, the binary relation \( \rightarrow^{(n)}_{i} \) is defined as \( \{(s, t) \mid s \xrightarrow{R} t \in NF \rightarrow \} \) where \( NF \rightarrow \) is the set of normal forms with respect to \( \rightarrow \). Let \( R \) be an operationally terminating DCTRS. The n-level innermost-like reduction \( \rightarrow^{(n)}_{(\rightarrow R)} \) is defined as follows:

\[
\begin{align*}
\rightarrow^{(n)}_{(\rightarrow R)} = & \emptyset, \\
\rightarrow^{(n+1)}_{(\rightarrow R)} = & \rightarrow^{(n)}_{(\rightarrow R)} \cup \{ (C[l\sigma], C[r\sigma]) \mid l \rightarrow r \Leftarrow s_1 \rightarrow t_1 \land \cdots s_k \rightarrow t_k \in R, \forall i, s_i\sigma \rightarrow^{(n)}_{(\rightarrow R)} t_i \} .
\end{align*}
\]

The innermost-like reduction \( \rightarrow^{(\rightarrow R)} \) of \( R \) is defined as \( \bigcup_{i \geq 0} \rightarrow^{(i)}_{(\rightarrow R)} \). Note that if \( R \) is a TRS, then the innermost-like reduction of \( R \) is just the ordinary innermost reduction.

For operationally terminating DCTRSs, it is clear that Theorem 2 holds for \( \rightarrow^{(\rightarrow R)} \) and \( \rightarrow^{(U_{cs}(R))} \).

**Theorem 3.** For every operationally terminating DCTRS \( R \) over a signature \( F \), \( U_{cs} \) is simulation-complete with respect to the innermost-like reduction, that is, \( \rightarrow^{(\rightarrow R)} = \rightarrow^{(U_{cs}(R))} \) in \( T(F, V) \).

Moreover, the context-sensitive constraint can be removed. Notice that the innermost reduction of operationally terminating CTRSs is well-defined.

**Theorem 4.** Let \( R \) be a DCTRS over a signature \( F \). Then, \( \rightarrow^{(\rightarrow U(R))} = \rightarrow^{(U_{cs}(R))} \) in \( T(F, V) \times T(F, U, V) \).

Therefore, when evaluating terms by innermost reductions, we can treat \( U(R) \) without the context-sensitive constraint determined by \( U \).

For confluent and operationally terminating DCTRSs, we have the following simulation-completeness.

**Corollary 5.** Let \( R \) be a confluent and operationally terminating DCTRS over a signature \( F \). Then, \( U \) is simulation-complete for \( R \) with respect to the innermost reduction, that is, \( \rightarrow^{(\rightarrow R)} = \rightarrow^{(U(R))} \) in \( T(F, V) \).
3.3 Applying Completion to Unraveled TRSs

In this subsection, we apply the completion procedure to the unraveled TRSs of CTRSs in order to transform them into convergent TRSs that are computationally equivalent to the CTRSs.

First, we introduce the completion procedure. For a binary relation \( \rightarrow \) on terms, the binary relation \( \rightarrow^* \) is defined as \( \{ (s, t) \mid s \rightarrow^* t \in \text{NF}_\rightarrow \} \). Here, \( \text{NF}_\rightarrow \) is the set of normal forms with respect to \( \rightarrow \).

**Definition 6 (Knuth-Bendix completion procedure [3]).** Let \( \mathcal{F} \) be a signature, \( E \) be a finite set of equations, and \( \rightarrow \) be a reduction order. Let \( E_0 = E \), \( R_0 = \emptyset \) and \( i = 0 \). Next, we apply the following steps:

1. (Orientation) select an equation \( s \approx t \in E_i \) such that \( s \not\rightarrow t \);
2. (Composition) \( R' := \{ l \rightarrow r' \mid l \rightarrow r \in R_i, r \rightarrow^*_{R_i \cup \{s \leftarrow t\}} r' \} \);
3. (Deduction) \( E' := (E_i \setminus \{ \{s \approx t\} \}) \cup \text{CP}(\{s \leftarrow t\}, R' \cup \{s \leftarrow t\}) \);
4. (Collapse) \( R_{i+1} := \{ s \leftarrow t \} \cup \{ l \rightarrow r \mid l \rightarrow r \in R', l \not\rightarrow s \} \};
5. (Simplification & Deletion)
   \[
   E_{i+1} := \{ s'' \approx t'' \mid s' \approx t' \in E', s' \rightarrow^*_{R_{i+1}} s'' \neq t'' \rightarrow^*_{R_{i+1}} t' \};
   \]
6. if \( E_{i+1} \neq \emptyset \) then \( i := i + 1 \) and go to step 1.

We denote an application of each cycle (1–5) to \( (E_i, R_i) \) by \( (E_i, R_i) \vdash (E_{i+1}, R_{i+1}) \). Note that the procedure does not always halt. Suppose that the procedure halts successfully at \( i = k \) (hence \( E_k = \emptyset \)). Then, \( R_k \) is convergent and non-overlapping, and \( R_k \) satisfies \( \rightarrow_E = \rightarrow_R \) [3]. Note that when there is no rule to select at the Orientation step, the procedure halts in failure. At steps 2 and 5 in the procedure, implementations often use the innermost reductions \( \rightarrow^*_{R \cup \{s \leftarrow t\}} \) and \( \rightarrow^*_R \) for \( \rightarrow^*_{R \cup \{s \leftarrow t\}} \) and \( \rightarrow^*_{R_{i+1}} \), respectively.

In the remainder of this subsection, we show the correctness of the TRSs obtained by the completion, that is, for a given TRS, the TRS obtained through the completion procedure is computationally equivalent to the given one.

The usual purpose of the completion is to generate TRSs that are equivalent to given equation sets. In contrast to the usual purpose, we would like the completion to modify unraveled TRSs \( \cup(R) \) to executable programs (convergent TRSs) that are computationally equivalent to the original CTRSs \( R \). For this reason, we start the completion procedure from \( (\text{CP}(\cup(R)), \{ l \leftarrow r \in \cup(R) \mid \exists l' \leftarrow r' \in \cup(R), l \not\rightarrow s \}) \) where \( \cup(R) \subseteq \rightarrow \). Moreover, consistency of the normal forms of \( \cup(R) \) (that is, they are also normal forms of the modified system) is necessary for preserving computational equivalence of \( R \). For this requirement, we add the side condition “root(s) is a U symbol” to the Orientation step:

1. (Orientation\(^1\)) select an equation \( s \approx t \in E_i \) such that \( s \not\rightarrow t \) and root(s) is a U symbol;

\(^1\) The encompassment quasi-order \( \supseteq \) on terms is defined as follows: \( l \supseteq s \) if there exists a subterm \( l' \) of \( l \) and a substitution \( \theta \) such that \( l' \equiv s\theta \).
Due to the side condition of the ORIENTATION step, and the basic character of the completion procedure [3], the completion procedure produces TRSs that are computationally equivalent to the input TRSs.

As we have already described, we would like to modify systems on a call-by-value interpretation. The innermost reduction is necessary for modeling some primitive functions used in some examples of program inversion (R in Subsection 4.2). Since such TRSs are not confluent but innermost-confluent, the completion procedure fails in modifying those TRSs. To solve this problem, we show a sufficient condition where the completion works well for innermost-convergent systems that are not confluent.

**Theorem 7 (correctness).** Let $R$ be an operationally terminating DCTRS over a signature $F$, and $\triangleright$ be a reduction order such that $\bigcup(R) \subseteq \triangleright$. Let $E_0 = CP_{c}(U(R))$, $R_0 = \{ l \rightarrow r \in U(R) \mid \exists l' \rightarrow r \in U(R), l \triangleright l' \}$, and $R'$ be a TRS obtained by the completion procedure from $(E_0, R_0)$ with $\triangleright$. Then, (1) $\text{NF}_{\bigcup(R)}(F, V) = \text{NF}_{R'}(F, V)$, (2) $R'$ is innermost-convergent, and (3) $\text{tr} \triangleright_{\bigcup(R)} = \text{tr} \triangleright_{R'}$ in $T(F, V)$.

The following condition is necessary for the completion procedure halting successfully: for every term $s \in T(F, V)$, its normal form over $F$ with respect to the innermost reduction is unique, that is, for all normal forms $t_1$ and $t_2$, if $t_1 \triangleright_{\bigcup(R)} s \triangleright_{\bigcup(R)} t_2$, then $t_1 \equiv t_2$. Note that if $R$ is confluent, then $\bigcup(R)$ satisfies this property. If $t_1 \triangleright_{\bigcup(R)} s \triangleright_{\bigcup(R)} t_2$ and $t_1 \not\equiv t_2$, then the additional side condition prevents the $t_1 \not\equiv t_2$ from being joinable.

**Example 8.** Consider the inconvergent TRS $\bigcup(\text{Inv}(R_1))$ in Section 1 again. Given the LPO $\triangleright_{\text{lpo}}$ determined by the precedence $>$ with $\text{InvSnoc} > \bigcup_1 \triangleleft \text{cons} > \text{nil} > (\ )$, we obtain the following convergent TRS by the completion procedure (in 4 cycles):

$$R_2 = \begin{cases} 
\text{InvSnoc}(\text{cons}(x, y)) \rightarrow U_1(\text{InvSnoc}(y), x, y) \\
U_1((x, y), x, y) \rightarrow \{\text{cons}(x, x), y\} \\
U_1(\text{InvSnoc}(\text{nil}), x, \text{nil}) \rightarrow \{\text{nil}, y\}
\end{cases}$$

Unfortunately, the completion procedure does not always halt even if the inputs are restricted to unraveled TRSs. For example, the completion does not halt for the unraveled TRS obtained from Example 7.1.5 in [27] although there exists an appropriate convergent TRS which is equivalent to the unraveled TRS, and which is available from completion by hand.

### 3.4 Translation Back into Functional Programs

In this subsection, we informally discuss translations from convergent TRS $R_2$ into functional programs in Standard ML. It is difficult to translate $\text{Inv}(R_1)$ or $\bigcup(\text{Inv}(R_1))$ into functional programs because deciding a priority of rewrite rules is difficult in general. On the other hand, we do not have to consider such
a priority for \( R_2 \) that is computationally equivalent to \( \text{Inv}(R_1) \) because \( R_2 \) is not only confluent but also non-overlapping.

We abbreviate \((t)\) to \( t \) in translating rewrite systems into functional programs. Note that we do not have to abbreviate \((t)\) to \( t \) in the term rewriting framework because the existence of tuple symbols \((.,),\ldots\) in unraveled rules make reduction sequences correspond to call-by-value evaluations.

The \( U \) symbols \( U_1^p \) introduced by the unraveling are often considered to express \text{let}, \text{if} or \text{case} clauses in functional programming languages. In the rewrite rules of \( R_2 \), the \( U \) symbol \( U_1 \) plays the role of a \text{case} clause as follows:

\[
\text{case } \text{InvSnoc}( \text{ys} ) \text{ of } (\text{xs}, \text{y}) \Rightarrow (\text{x}::\text{xs}, \text{y}) \\
| \text{InvSnoc}(\text{[]} ) \Rightarrow (\text{[]}, \text{y})
\]

where the pattern \( \text{InvSnoc}(\text{[]} ) \) is not well-formed in the syntax of Standard ML. It is natural to write this fragment by introducing the extra \text{case} clause for \( \text{ys} \) as follows:

\[
\text{case } \text{ys} \text{ of } \text{[]} \Rightarrow (\text{[]}, \text{y}) \\
| \_ \Rightarrow (\text{case } \text{InvSnoc}( \text{ys} ) \text{ of } (\text{xs}, \text{y}) \Rightarrow (\text{x}::\text{xs}, \text{y}))
\]

Thus, we translate the TRS \( R_2 \) into the following program:

\[
\text{fun } \text{InvSnoc}(\text{x}::\text{ys}) = \\
\text{case } \text{ys} \text{ of } \text{[]} \Rightarrow (\text{[]}, \text{x}) \\
| \_ \Rightarrow (\text{case } \text{InvSnoc}(\text{ys}) \text{ of } (\text{xs}, \text{y}) \Rightarrow (\text{x}::\text{xs}, \text{y}));
\]

Other approaches to translations are possible. For example, we can consider \( U_1 \) as the composition of \text{if} and \text{let} clauses or as a “local function” defined in \text{InvSnoc}.

### 3.5 Completion with Termination Provers

In this subsection, we show an example where the completion consults with termination provers.

Consider the following unraveled TRS:

\[
R_3 = U(\text{Inv}(R_1)) \cup \left\{ \begin{array}{c}
\text{InvSnocRev}(\text{nil}) \rightarrow (\text{nil}) \\
\text{InvSnocRev}(\text{y}) \rightarrow U_2(\text{InvSnoc}(\text{y}), \text{y}) \\
U_2((\text{z}, \text{x}), \text{y}) \rightarrow U_3(\text{InvSnocRev}(\text{z}), \text{x}, \text{y}, \text{z}) \\
U_3((\text{x}_1), \text{x}, \text{y}, \text{z}) \rightarrow (\text{cons}(\text{x}, \text{x}_1))
\end{array} \right\}
\]

In contrast to the case of \text{InvSnoc}, there is no LPO \( \succ_{\text{lopo}} \) with \( R_3 \subseteq \succ_{\text{lopo}} \). Detecting such a path-based reduction order (LPO, RPO and so on) in advance may be impossible or there might be no such path-based order. Thus, analyzing the dependencies of defined symbols is necessary to prove the termination of \( R_3 \).

To achieve this kind of analysis, we introduce termination provers to the completion procedure [34]. We modify \text{ORIENTATION}, following the approach shown in [34]:

9
1. (Orientation\(^{1}\)) select an equation \(s \approx t \in E(i)\) such that \(\bigcup_{j=0}^{i} R(j) \cup \{s \to t\}\) is terminating, and \(\text{root}(s)\) is a \(U\) symbol;

In the case of innermost reduction, it is enough to check innermost-termination of \(\bigcup_{j=0}^{i} R(j) \cup \{s \to t\}\). This setting enables us to employ existing termination provers at each Orientation step.

In this mechanism, the TRS \(R_3\) is modified by the procedure (in 2 cycles) into a convergent TRS as follows:

\[
R_2 \cup \left\{
\begin{array}{l}
\text{InvSnocRev}(v) \to U_2(\text{InvSnoc}(v), v) \\
U_2((w, x), v) \to U_3(U_2(\text{InvSnoc}(w), w), v, w, x) \\
U_3((xs), v, w, x) \to \{\text{cons}(x, xs)\} \\
U_3(\text{InvSnoc}(\text{nil}), x, \text{nil}) \to \{\text{nil}, x\}
\end{array}
\right.
\]

4 Application to Program Inversion of Injective Functions

In this section, we apply the unraveling \(U\) and the completion procedure to CTRSs generated by the partial inversion compiler [24]. First, we briefly introduce the feature of inverse systems for injective functions. Next, we show the results of experiments by an implementation of the framework. We employ the partial inversion \(\text{Inv}\) in [24] that generates a partial inverse CTRS from a pair of a given constructor TRS and a specification that we do not describe in detail here. For a defined symbol \(F\), the defined symbol \(\text{Inv}F\) introduced by \(\text{Inv}\) represents a full inverse of \(F\). We assume that a constructor TRS defines a main injective function, and that the specification requires a full inverse of the main function.

4.1 Inverse CTRSs of Injective Functions

In this subsection, we first define injectivity of TRSs, and then give sufficient condition for input constructor TRSs whose inverse CTRSs generated by \(\text{Inv}\) are convergent.

We define injective TRSs as follows.

**Definition 9 ([22]).** Let \(R\) be a convergent constructor TRS. A defined symbol \(F\) of \(R\) is called injective (with respect to normal forms) if the binary relation \(\{(s_1, \ldots, s_n), t) \mid s_1, \ldots, s_n, t \in \text{NF}_R(F, V), F(s_1, \ldots, s_n) \xrightarrow{R} t\}\) is an injective mapping. The TRS \(R\) is called injective (with respect to normal forms) if all of its defined symbols are injective.

For example, the TRS \(R_1\) in Section 1 is injective. Note that every injective TRS is non-erasing [22].

The following defined symbol \(\text{Reverse}\) computes the reverses of given lists:

\[
R_4 = \left\{
\begin{array}{l}
\text{Reverse}(xs) \to \text{Rev}(xs, \text{nil}), \quad \text{Rev}(\text{nil}, ys) \to ys \\
\text{Rev}(\text{cons}(x, xs), ys) \to \text{Rev}(xs, \text{cons}(x, ys))
\end{array}
\right.
\]
The inverse TRS of the above TRS is generated as follows:

\[ \bigcup (\text{Inv}(R_4)) = \{ \cdots , \text{InvRev}(z) \rightarrow U_4(\text{InvRev}(z), z), \cdots \} \]

**Reverse** is injective but **Rev** is not. Thus, **R** is not injective. In this case, the TRS \( \bigcup (\text{Inv}(R_4)) \) is not terminating. For this reason, we restrict ourselves to injective functions whose inverse TRSs are terminating. In [22], a sufficient condition has been shown for the full inversion compiler in [25] to generate convergent inverse CTRSs from injective TRSs. The condition is also effective for the partial inversion compiler \( \text{Inv} \).

**Theorem 10.** Let \( R \) be a non-erasing innermost-convergent constructor TRS. If \( F \in D_R \) is injective, then for all normal forms \( s \), \( t_1 \) and \( t_2 \in NF_{\text{Inv}(R)}(F, V) \), if \( t_1 \rightarrow^*_{\bigcup (\text{Inv}(R))} \text{Inv}(F)(s) \rightarrow^*_{\bigcup (\text{Inv}(R))} t_2 \), then \( t_1 \equiv t_2 \). Suppose that for every rule \( F(u_1, \ldots, u_n) \rightarrow r \) in \( R \), if \( r \) is not a variable, then the root symbol of \( r \) does not depend \(^2\) on \( F \). Then, the CTRS \( \text{Inv}(R) \) is operationally terminating, and the TRS \( \bigcup (\text{Inv}(R)) \) is terminating.

Note that \( \bigcup (\text{Inv}(R)) \) is not always confluent even if \( \text{Inv}(R) \) is confluent. When a constructor TRS \( R \) does not satisfy the condition in Theorem 10 for preserving termination, we may check the innermost termination of \( \bigcup (\text{Inv}(R)) \).

### 4.2 Experiments

In this section, we report the results of applying an implementation of our approach based on Theorem 7 and the **ORIENTATION**\(^3\) step to several samples.

In [10], the results of the experiments for the inversion compiler LRinv [10, 11] running on 15 samples\(^3\) are shown where LRinv succeeds in inverting all of the examples. Those examples are written in the scheme script Gauche: 5 scripts on “list manipulation” (**snoc.fct**, **snocrev.fct**, **reverse.fct**, and so on), 3 on “number manipulation”, 4 on “encoding and decoding” (**treelist.fct**, and so on), and 2 on “printing and parsing”. The inverse TRSs of the scripts **snoc.fct**, **snocrev.fct** and **reverse.fct** correspond to the TRSs \( \bigcup (\text{Inv}(R_1)) \), \( R_3 \) and \( \bigcup (\text{Inv}(R_4)) \), respectively. None of the constructor TRSs corresponding to the scripts **reverse.fct**, **unbin.fct**, **treepath.fct**, **pack.fct** and **pack-bin.fct** are injective. The CTRSs obtained by \( \text{Inv} \) from them are not operationally terminating. We do not treat those examples in these experiments.

In the examples, there is a special primitive operator \( \text{du} \) defined as follows: \( \text{du}(x) = (x, x) \), \( \text{du}(x, x) = (x) \), and \( \text{du}(x, y) = (x, y) \) if \( x \neq y \). We encode this operator as the following TRS:

\[
R_{\text{du}} = \begin{cases} 
\text{Du}(x) \rightarrow (x, x), & \text{Du}(y, y) \rightarrow \text{EqChk}(\text{EQ}(x, y)), \\
\text{EqChk}(x) \rightarrow (x), & \text{EqChk}(\text{EQ}(x, y)) \rightarrow (x, y), \\
\text{EQ}(x, x) \rightarrow \langle x \rangle & 
\end{cases}
\]

\(^2\) An \( n \)-ary symbol \( G \) of \( R \) depends on a symbol \( F \) if \((G, F)\) is in the transitive closure of the relation \( \{ (G', F') | G'(\cdot \cdot \cdot) \rightarrow C[F'(\cdot \cdot \cdot)] \in R \} \).

\(^3\) Unfortunately, the site shown in [10] is not accessible now. The examples are also described briefly as functional programs in [11], and some of the detailed programs can be found in [11].
This TRS is terminating. Since $R_{du}$ has no overlay, $R_{du}$ is locally innermost-confluent, and hence, $R_{du}$ is innermost-confluent [12]. Under the innermost reduction, the TRS can simulate computation of $du$. The operator $du$ is an inverse of itself [10, 11]. Therefore, the TRS $R_{du}$ is also an inverse system of itself. For this reason, the inversion compiler produces $Du$ itself from $Du$, and replaces InvDup in other generated systems with $Du$.

Table 1 summarizes the results of the experiments for our approach running on 10 of the 15 examples mentioned, which are easily translated to TRSs$^4$. The implementation of the completion procedure is based on the programs shown in [3]. In our implementation, we use the following weight $w$ for equations: $w(s \rightarrow t) = (\text{size}(s) + \text{depth}(s)) \times 2 + (\text{size}(t) + \text{depth}(t))$ where $s > t$, and $\text{size}(u)$ and $\text{depth}(u)$ are the term-size and term-depth of $u$, respectively. The weight $w$ is one of weights that come from experience. At every Orientation step, the implementation selects an equation whose weight is the minimum in equations $s \rightarrow t$ such that $(\bigcup_{i=0}^{j} R_{(i)} \cup \{s \rightarrow t\})$ is innermost terminating. The implementation is written in Standard ML of New Jersey, and it was executed under OS Vine Linux 4.2, on an Intel Pentium 4 CPU at 3 GHz and 1 GByte of primary memory. The implementation consults with AProVE 1.2 [6] by system call in SML/NJ as a termination prover at the Orientation step that checks the innermost termination. The implementation checks the innermost termination of input TRSs before the completion procedure starts. The timeout for checking termination is 300 seconds. The second column labeled with “[1]” shows whether the input TRS of the example is in the class of [1] in which the corresponding inverse TRS is non-overlapping. In that case, the implemented procedure only checks innermost-termination of the inverse TRS. The third column labeled with “Th. 10” shows whether the input TRS satisfies the conditions in Theorem 10. The information “not applicable” in the fourth column says that no appropriate LPO could be found and then the completion procedure cannot be applied. The rightmost column shows the average time of 5 trials.

In the experiments, the procedure failed in modifying treelist.fct because of a timeout in pre-checking the innermost termination. For the intermediate CTRS $R_5$ generated from treelist.fct, the improvement of $U$ shown in [21] is effective. Note that the improvement proposed for the variant $T$ of $U$ is also applicable to $U$ where $T$ shown in [5, 24, 25] is obtained by setting $X_i = \text{Var}(l, l_1, \ldots, l_{i-1}) \cap \text{Var}(r, t_i, s_{i+1}, t_{i+1}, \ldots, s_k, t_k)$ in Definition 1. $U$ unravels one of the conditional rules into 5 rules but the improvement of $U$ unravels the rule into 4. Surprisingly, the completion succeeds in modifying the TRS obtained by the improvement unraveling (see the result on the bottom line of Table 1).

Remark that in other examples, there is no difference between applications of $U$ and the improvement of $U$.

We tried to prove by TTT [9] innermost-termination of $U(R_5)$ and the TRS obtained by applying the improvement of $U$ to $R_5$. The results are the same with AProVE 1.2. Moreover, AProVE 1.2 succeeded in proving termination of

---

$^4$ The detail of the experiments will be available from the following URL:
[http://www.trs.cm.is.nagoya-u.ac.jp/repius/experiments/](http://www.trs.cm.is.nagoya-u.ac.jp/repius/experiments/)
Table 1. the results of the experiments

<table>
<thead>
<tr>
<th>example</th>
<th>[1]</th>
<th>Th.10</th>
<th>completion</th>
<th>result (cycles)</th>
<th>call termination provers</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>du (primitive)</td>
<td></td>
<td></td>
<td>success</td>
<td>0 cycle</td>
<td>1 times</td>
<td>0.64 s</td>
</tr>
<tr>
<td>snoc.fct</td>
<td></td>
<td>✓</td>
<td>success</td>
<td>1 cycle</td>
<td>2 times</td>
<td>2.12 s</td>
</tr>
<tr>
<td>snocrev.fct</td>
<td>✓</td>
<td></td>
<td>success</td>
<td>2 cycles</td>
<td>3 times</td>
<td>4.61 s</td>
</tr>
<tr>
<td>double.fct</td>
<td></td>
<td></td>
<td>success</td>
<td>1 cycle</td>
<td>2 times</td>
<td>2.32 s</td>
</tr>
<tr>
<td>mirror.fct</td>
<td></td>
<td></td>
<td>success</td>
<td>2 cycles</td>
<td>3 times</td>
<td>4.10 s</td>
</tr>
<tr>
<td>zip.fct</td>
<td>✓</td>
<td>✓</td>
<td>success</td>
<td>0 cycle</td>
<td>1 times</td>
<td>1.04 s</td>
</tr>
<tr>
<td>inc.fct</td>
<td>✓</td>
<td></td>
<td>success</td>
<td>1 cycle</td>
<td>2 times</td>
<td>2.53 s</td>
</tr>
<tr>
<td>octbin.fct</td>
<td>✓</td>
<td></td>
<td>success</td>
<td>0 cycle</td>
<td>1 times</td>
<td>1.28 s</td>
</tr>
</tbody>
</table>
| treelist.fct
                              ✓   | fail | (0 cycle) | timeout at 1st time | timeout |
| print-sexp.fct  |     |       | success    | 6 cycles        | 7 times                  | 35.53 s |
| print-xml.fct   |     |       | success    | 2 cycles        | 3 times                  | 14.02 s |
| treelist.fct\^1 | ✓   |       | success    | 4 cycles        | 5 times                  | 40.92 s |

\^1The improvement [21] of $U$ is applied instead of $U$.

both $\mathbb{U}(R_5)$ and the TRS obtained by applying the improvement of $U$ to $R_5$, and TTT did not in either of those TRSs.

5 Related Work

Completion procedures are used for solving word problems or for transforming equations to equivalent convergent systems. As far as we know, there is no application of completion to program modification, and there is no program transformation based on unravelings in order to produce computationally equivalent systems. The method in this paper does not always succeed for every confluent and operationally terminating DCTRSs while the latest transformation [31] based on Viry’s approach always succeeds. However, the method succeeded for all examples on program inversion we tried, except for functions that call non-injective functions such as ones including accumulators. The inverse relations of many-to-one functions are not functions. For this reason, many inversion compilers generate programs in extended languages to express the relations, or programs that require extended interpreters for executing [28, 23, 10, 24, 25, 11, 7, 1]. Even treating only injective functions, frameworks have to be developed to generate programs written in the same language of the input programs. As far as we know, the essences of inversion in many inversion compilers are very similar. The difference is an arrangement to the inverted programs that are an intermediate result of whole inversion.

The inversion compiler LRInv has been proposed for injective functions written in a functional language [10, 7, 11]. This compiler translates source programs into programs in a grammar language, and then inverts the grammar programs.
into inverse grammar programs. To eliminate nondeterminism in the inverse programs, their compiler applies LR parsing to the inverse programs. The classes for which LR parsing and the completion procedure work successfully are not well known, which is a difficulty for comparing LRinv and our method.

Semi-inversion is a more general notion of partial inversion [18, 19]. In this approach, intermediate semi-inverse programs are translated back into programs written in the original functional language if the domain of rules are disjoint. However, the presence of overlapping domains may be undecidable in general. The semi-inversion transformation has been extended for higher-order functions [20].

Bidirectional transformation is an application of program inversion [17]. It employs partial inversion techniques to produce inverses. Programs written in the language for bidirectional transformation are linear (Affine) and in treeless form. Similar techniques have been developed as view updating in the field of database systems.

6 Conclusion

In this paper, we have shown that in some cases, unravelings are of great use in translating rewrite systems generated by the inversion compiler into functional programs. We expect that the completion is useful for some cases in transforming unraveled TRSs into ones that are computationally equivalent to the original CTRSs if the procedure halts successfully. One of future work is to find a condition of the class for which the completion halts, and then to compare this framework with the transformation based on Viry’s approach [31].

All the functional programs in this paper run successfully on Standard ML of New Jersey while they may have the warning “match nonexhaustive”. We succeeded in removing the context-sensitive constraint from the unraveled TRSs generated by the inversion compiler. This removal plays an important role in simplifying the discussion. As future work, we plan to extend the partial inversion compiler for functions with accumulators such as Rev.

This paper has shown an application of unravelings in program transformations. To apply the completion (or other transformation techniques), the unraveled TRSs must be simulation-complete for the original CTRSs. For this reason, unravelings and their simulation-completeness are worth being discussed.

Acknowledgement

We are grateful to Germán Vidal for improving this paper. We also thank Tsubasa Sakata for his discussing correctness of the completion procedure on innermost reduction.

References

25. Nishida, N., Sakai, M., Sakabe, T.: Generation of inverse computation programs of
AAECC 12(1-2) (2001) 73–116
29. Rosu, G.: From conditional to unconditional rewriting. In: Revised selected papers
30. Schernhammer, F., Gramlich, B.: On proving and characterizing operational ter-
82–85
32. Toyana, Y.: How to prove equivalence of term rewriting systems without induction.
34. Wehrman, I., Stump, A., Westbrook, E.M.: Slothrop: Knuth-Bendix completion
287–296