Completion after Program Inversion of Injective Functions

Naoki Nishida 1 Masahiko Sakai 2

Graduate School of Information Science, Nagoya University
Nagoya, Japan

Abstract
Given a constructor term rewriting system that defines injective functions, the inversion compiler proposed by Nishida, Sakai and Sakabe generates a conditional term rewriting system that defines the inverse relations of the injective functions, and then the compiler unravels the conditional system into an unconditional term rewriting system. In general, the resulting unconditional system is not (innermost-)confluent even if the conditional system is (innermost-)confluent. In this paper, we propose a modification of the Knuth-Bendix completion procedure, which is used as a post-processor of the inversion compiler. Given a confluent and operationally terminating conditional system, the procedure takes the resulting unconditional systems as input. When the procedure halts successfully, it returns convergent systems that are computationally equivalent to the conditional systems. To adapt the modified procedure to the conditional systems that are not confluent but innermost-confluent, we propose a simplified variant of the modified procedure. We report that the implementations of the procedures succeed in generating innermost-convergent inverse systems for all the examples we tried.

Keywords: unraveling, conditional term rewriting system, convergence, innermost reduction

1 Introduction

Inverse computation of an n-ary function f is, given an output v, the calculation of the possible input v1, ..., vn of f such that f(v1, ..., vn) = v. Two approaches for inverse computation are distinguished [1]: inverse interpreters [4,1] that performs inverse computation, and inversion compilers [18,28,9,25,24,7,19,20,2] that performs program inversion.

Given a constructor term rewriting system (constructor TRS), the inversion compiler proposed in [24,25] first generates a deterministic conditional TRS (DC-TRS) as an intermediate result, and then transforms the DCTRS into a TRS that is equivalent to the DCTRS with respect to inverse computation. The first phase of the compiler performs a local inversion: for every constructor TRS, the first

1 Email: nishida@is.nagoya-u.ac.jp
2 Email: sakai@is.nagoya-u.ac.jp
phase generates a DCTRS, called an inverse system, which represents the complete inverse relation for the reduction relation of 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 DCTRSs into TRSs.

Unfortunately, the compiler cannot always generate TRSs that are computationally equivalent to the corresponding DCTRSs due to a characteristic of unravelings [15,27,30,22]. The characteristic is that the unraveled TRSs of DCTRSs may have unexpected normal forms that represent dead ends of wrong choices at branches of evaluating conditional parts of the DCTRSs (see the example $\text{Inv}(R_1)$ shown later in this section). These wrong choices are captured by critical pairs of the unraveled TRSs, each of which originates from two (conditional) rewrite rules corresponding to the ‘correct’ and ‘wrong’ choices. Note that any rules looking like ‘wrong choice’ must be necessary elsewhere, and that it is decidable whether or not a normal form is expected: a normal form of the unraveled TRSs is an unexpected one if it contains an extra defined symbol introduced by the unraveling.

In program inversion by the inversion compiler [25,24], this problem arises even if all functions defined in the given constructor TRSs are injective. For this reason, the resulting TRSs do not define functions and thus the inversion compiler is less applicable to injective functions in practical functional programming languages — it is easy to translate functional programs into constructor TRSs, but difficult to translate the resulting TRSs of the compiler back into functional programs.

In this paper, we propose a modification of the Knuth-Bendix completion procedure in order to transform the unraveled TRSs of confluent and operationally terminating DCTRSs into convergent (and possibly non-overlapping) TRSs that are computationally equivalent to the DCTRSs. Unfortunately, the procedure does not always halt just as in the case of the ordinary completion procedure. However, if the procedure halts successfully and the resulting convergent TRSs are non-overlapping, then the resulting systems can be translated back into functional programs due to the non-overlapping property. When all functions defined in the input TRSs are injective, we take the modified completion procedure as a post-processor into the inversion compiler (Fig. 1 and Section 5). Through this approach, we show that unravelings are useful not only in analyzing properties of DCTRSs [15,27] but also in generating programs that can be used for computation instead of the corresponding original programs, such as program inversion of functional programs.

Consider the following functional program written in Standard ML where $\text{Snoc}(xs, y)$ produces the list obtained from $xs$ by adding $y$ as the last element:
fun Snoc( [], y ) = [y]
| Snoc( x::xs, y ) = x :: Snoc( xs, y );

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

\[ R_1 = \{ \text{Snoc}(\text{nil}, y) \rightarrow [y], \text{Snoc}(x::xs, y) \rightarrow x::\text{Snoc}(xs, y) \} \]

where nil and :: are list constructors as usual, \([t_1, t_2, \ldots, t_n]\) abbreviates the list \(t_1 :: (t_2 :: \cdots :: (t_n :: \text{nil}) \cdots)\). The compiler inverts \(R_1\) into the following DCTRS in the first phase:

\[
\mathcal{I}_{\text{inv}}(R_1) = \begin{cases} 
\text{InvSnoc}([y]) \rightarrow \langle \text{nil}, y \rangle, \\
\text{InvSnoc}(x::ys) \rightarrow \langle x::xs, y \rangle \leftarrow \text{InvSnoc}(ys) \rightarrow \langle x, y \rangle 
\end{cases}
\]

where each tuple of \(n\) terms \(t_1, \ldots, t_n\) is denoted by \(\langle t_1, \ldots, t_n \rangle\) that can be represented as terms by introducing an \(n\)-ary constructor. The compiler unravels the DCTRS \(\mathcal{I}_{\text{inv}}(R_1)\) into the following TRS in the second phase:

\[
\mathbb{U}(\mathcal{I}_{\text{inv}}(R_1)) = \begin{cases} 
\text{InvSnoc}([y]) \rightarrow \langle \text{nil}, y \rangle, \\
\text{InvSnoc}(x::ys) \rightarrow U_1(\text{InvSnoc}(ys), x, ys), \\
U_1(\langle x, y \rangle, x, ys) \rightarrow \langle x::xs, y \rangle 
\end{cases}
\]

The introduced symbol \(U_1\) is used for evaluating the conditional part \(\text{InvSnoc}(ys) \rightarrow \langle x, y \rangle\) of the second rule in \(\mathcal{I}_{\text{inv}}(R_1)\). 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 an unexpected normal form \(U_1(U_1(\text{InvSnoc}(\text{nil}), c, \text{nil}), b, [c]), a, [b, c])\). The restricted inversion compiler in [2] for generating non-overlapping systems is not applicable to this case because \(R_1\) is out of its scope. In this example, it appears to be easy to translate from the TRS \(\mathbb{U}(\mathcal{I}_{\text{inv}}(R_1))\) or the CTRS \(\mathcal{I}_{\text{inv}}(R_1)\) into a functional program because we can easily determine an appropriate priority of rules, for instance, the common first rule \(\text{InvSnoc}[y] \rightarrow \langle \text{nil}, y \rangle\) may have the highest priority. However, such a translation based on priorities of rules is difficult in general because we cannot decide which rules have priority of the application to terms. On the other hand, it is probably impossible that one transforms input systems into equivalent systems from which the compiler generates the inverse systems without overlapping.

To avoid this problem, it has been shown in [22] that the transformation in [30] is suitable as the second phase of the compiler, in the sense of producing convergent systems. However, the generated systems contain some special symbols and overlapping rules. For this reason, it is difficult to translate the convergent but overlapping TRS into a functional program (see Section 6).

Roughly speaking, non-confluence of \(\mathbb{U}(\mathcal{I}_{\text{inv}}(R_1))\) comes from the critical pair \((\langle \text{nil}, x \rangle, U_1(\text{InvSnoc}(\text{nil}), x, \text{nil}))\) between the first and second rules in \(\mathbb{U}(\mathcal{I}_{\text{inv}}(R_1))\). In this case, the application of the first rule is ‘correct’ and that of the second

---

3 To simplify discussions, we omit describing special rules in the form of \(\text{InvF}(F(x_1, \ldots, x_n)) \rightarrow \langle x_1, \ldots, x_n \rangle\) [25,24] because they are meaningless for inverse computation in dealing with functional programs on call-by-value interpretation. The special rules are necessary only for inverse computation of normalizing computation in term rewriting.
is ‘wrong’, that is, \( \langle \text{nil}, x \rangle \) is the expected result and \( U_1(\text{InvSnoc}(\text{nil}), x, \text{nil}) \) is the unexpected recursive call of \( U_1 \) containing the dead end \( \text{InvSnoc}(\text{nil}) \). From this observation, by adding the rule \( U_1(\text{InvSnoc}(\text{nil}), x, \text{nil}) \rightarrow \langle \text{nil}, x \rangle \), the unexpected 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 back to the correct branch. Due to this rule, the new TRS is confluent. This process just corresponds to the behavior of completion. Therefore, completion is expected to solve the non-confluence of TRSs obtained by the inversion compiler.

In Section 3, we propose a notion of operationally innermost reduction of DC-TRSs that corresponds to call-by-value interpretation of functional programs, and we show that simulation-completeness with respect to innermost reduction is preserved by Ohlebusch’s unraveling if the DCTRSs are restricted to functional programs having let-like structures.

In Section 4, we propose a modification of the Knuth-Bendix completion procedure, by adding a side condition to the orientation phase. Given a confluent and operationally terminating DCTRS, the modified completion procedure takes the unraveled TRSs as input. When the procedure halts successfully, it returns a convergent TRS that is computationally equivalent to the DCTRS. To obtain innermost-convergent TRSs from the unraveled TRSs of operationally terminating DCTRSs that are not confluent but innermost-confluent, we simplify the modified completion procedure by prohibiting the modified procedure to use two basic functions (composition and simplification), and by giving an additional side condition to the orientation phase. The additional condition restricts orientable equations to equations that are oriented without overlaps with other rewrite rules.

In Section 5, we first show a sufficient condition of constructor TRSs from which the inversion compiler generates (innermost-)convergent DCTRSs. Next, we describe an implementation of the modified completion procedure, and the experiments for the unraveled TRSs of DCTRSs obtained by the inversion compiler [24] from injective functions shown by Kawabe et al. [9]. Finally, we illustrate an informal translation of the non-overlapping TRSs obtained by the procedure back into functional programs.

In this paper, we do not consider sorts. However, the framework in this paper can be extended to many-sorted systems as usual. All proofs can be found in the full version of this paper [21].

## 2 Preliminaries

Here, we will review the following basic notations of term rewriting [3,27].

Throughout this paper, we use \( V \) as a countably infinite set of variables. The set of all terms over a signature \( F \) and \( V \) is denoted by \( T(F, V) \). The set of all variables appearing in the 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 \( p \). 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 each □ 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, and the application \( \sigma(t) \) of \( \sigma \) to \( t \) is abbreviated to \( \sigma t \). The **composition** \( \theta \) of substitutions \( \sigma \) and \( \theta \) is defined as \( \sigma \theta(x) = \theta(\sigma(x)) \).

Given terms \( s \) and \( t \), we write \( s \not\geq \sigma t \) if there are some \( C[\ ] \) and \( \theta \) such that \( s \equiv C[t\theta] \).

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 \( T(\mathcal{F}, \mathcal{V}) \), \( r \) is a term in \( T(\mathcal{F}, \mathcal{V}) \), and \( c \) is of the form of \( s_1 \rightarrow t_1 \land \cdots \land s_n \rightarrow t_n \) \((n \geq 0)\) with terms \( s_i \) and \( t_i \) in \( T(\mathcal{F}, \mathcal{V}) \). In particular, the conditional rewrite rule \( l \rightarrow r \Leftarrow c \) is said to be an **(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. An **(oriented) conditional rewriting system** (CRS, for short) \( R \) over a signature \( \mathcal{F} \) is a finite set of conditional rewrite rules over \( \mathcal{F} \). Note that \( R \) is a **TRS** if all rules in \( R \) are unconditional. 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 \( \rightarrow^{[\rho]}_R \). We write \( \rightarrow^\varepsilon_R \) if \( p \) is not the root position \( \varepsilon \). A conditional rewrite rule \( \rho : l \rightarrow r \Leftarrow s_1 \rightarrow t_1 \cdots s_k \rightarrow t_k \) is called **deterministic** if \( \text{Var}(r) \subseteq \text{Var}(l, s_1, \ldots, s_k) \) and \( \text{Var}(s_i) \subseteq \text{Var}(l, s_1, \ldots, t_{i-1}) \) for \( 1 \leq i \leq k \). The CTRS \( R \) is called a **deterministic CTRS** (DCTRS, for short) if all rules in \( R \) are deterministic.

A notion of **operational termination** of DCTRSs is defined via the absence of infinite well-formed proof trees in some inference system [14]: a CTRS \( R \) is **operationally terminating** (OP-SN, for short) if for any terms \( s \) and \( t \), any proof tree attempting to prove that \( s \not\rightarrow^* R t \) cannot be infinite.

Let \( \rightarrow \) be a reduction over terms in \( T(\mathcal{F}, \mathcal{V}) \). Then, the set of normal forms with respect to \( \rightarrow \) is denoted by \( NF_\rightarrow(\mathcal{F}, \mathcal{V}) \). The binary relation \( \rightarrow^\dag_\rightarrow \) is defined as \( \{ (s, t) \mid s \not\rightarrow^* \rightarrow t, t \in NF_\rightarrow(\mathcal{F}, \mathcal{V}) \} \).

Let \( R \) be a CTRS 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. Terms in \( T(\mathcal{C}_R, \mathcal{V}) \) are called **constructor terms** of \( R \). The CTRS \( R \) is called a **constructor system** if every rule \( f(t_1, \ldots, t_n) \rightarrow r \Leftarrow c \in R \) satisfies \( \{ t_1, \ldots, t_n \} \subseteq T(\mathcal{C}_R, \mathcal{V}) \).

We use the notion of **context-sensitive reduction** in [13]. A **replacement mapping** \( \mu \) is a mapping from a signature \( \mathcal{F} \) to a set of natural numbers such that \( \mu(f) \subseteq \{ 1, \ldots, n \} \) for \( n \)-ary symbols \( f \) in \( \mathcal{F} \). When \( \mu(f) \) is not defined explicitly, we assume that \( \mu(f) = \{ 1, \ldots, n \} \). The set \( \mathcal{O}_\mu(t) \) of reducible positions in \( t \) is defined as follows:

\[
\mathcal{O}_\mu(x) = \emptyset \quad \text{where } x \in \mathcal{V}, \quad \mathcal{O}_\mu(f(t_1, \ldots, t_n)) = \{ ip \mid i \in \mu(f), p \in \bigcup_{j \in \mu(f)} \mathcal{O}_\mu(t_j) \}.
\]

The context-sensitive reduction of the context-sensitive TRS \((R, \mu)\) of a TRS \( R \) and a replacement map \( \mu \) is denoted by \( \rightarrow_{(R, \mu)} \): \( \rightarrow_{(R, \mu)} = \{ (s, t) \mid s \not\rightarrow^p_{(R, \mu)} t, p \in \mathcal{O}_\mu(s) \} \). The innermost reduction of \( \rightarrow_{(R, \mu)} \) is denoted by \( \rightarrow^\dag_{(R, \mu)} \): \( \rightarrow^\dag_{(R, \mu)} = \{ (s, t) \mid s \not\rightarrow^p_{(R, \mu)} t, p \in \mathcal{O}_\mu(s), (\forall q > p, q \in \mathcal{O}_\mu(s) \text{ implies that } s|_q \text{ is irreducible}) \} \).

Let \( l_i \rightarrow r_i \) \((i = 1, 2)\) 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 the two rules are renamed versions of the same rewrite rule, we
do not consider the case $p = \varepsilon$. 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 $CP(R)$. We also denote the set of critical pairs between rules in $R$ and another TRS $R'$ by $CP(R,R')$. Moreover, $CP_\varepsilon(R)$ denotes the set of overlays of $R$.

Let $R$ and $R'$ be CTRSs such that their 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 $\psi(t') = t$ [30]. In this paper, we assume that $\phi$ and $\psi$ are the identity mappings.

Let $\rightarrow_1$ and $\rightarrow_2$ be two binary relations on terms, and $T'$ and $T''$ be sets of terms. We say that $\rightarrow_1 = \frac{1}{2} \rightarrow_2$ in $T' \times T''$ ($\rightarrow_1 \supseteq \rightarrow_2$ in $T' \times T''$, respectively) if $\rightarrow_1 \cap (T' \times T'') = \frac{1}{2} \cap (T' \times T'')$ ($\rightarrow_1 \cap (T' \times T'') \supseteq \frac{1}{2} \cap (T' \times T'')$, respectively). Especially, we say that $\rightarrow_1 = \frac{1}{2} \rightarrow_2$ in $T'$ (and $\rightarrow_1 \supseteq \frac{1}{2} \rightarrow_2$ in $T'$) if $T' = T''$.

An equation over a signature $\mathcal{F}$ is a pair $(s,t)$, denoted by $s \simeq t$, such that $s$ and $t$ are terms in $T(\mathcal{F},\mathcal{V})$. We write $s \simeq t$ for representing $s \simeq t$ or $t \simeq s$. The equational relation with respect to a set $E$ of equations is defined as $\leftrightarrow_E = \{(C[s\sigma],C[t\sigma]) \mid s \simeq t \in E\}$.

Finally, we introduce the Knuth-Bendix completion procedure [11,3,31].

**Definition 2.1** Let $E$ be a finite set of equations over a signature $\mathcal{F}$, and $\succ$ be a reduction order. Let $E(0) = E$, $R(0) = \emptyset$ and $i = 0$, we apply the following steps:

1. (Orientation) select $s \simeq t \in E(i)$ such that $s \succ t$;
2. (Composition) $R' := \{l \rightarrow r' \mid l \rightarrow r \in R(i), r \overset{*}{\rightarrow}_1 R(i) \cup \{s \rightarrow t\} \rightarrow r'\}$;
3. (Deduction) $E' := (E(i) \setminus \{s \simeq t\}) \cup CP:\{(s \rightarrow t), R' \cup \{s \rightarrow t\}\}$;
4. (Collapse) $R(i+1) := \{s \rightarrow t\} \cup \{l \rightarrow r' \mid l \rightarrow r \in R', l \not\simeq s\}$;
5. (Simplification & Deletion)

$E(i+1) := \{s'' \approx t'' \mid s' \approx t' \in E', s' \overset{*}{\rightarrow}_1 R(i+1), s'' \not\approx t'' \overset{*}{\rightarrow}_1 R(i+1) \}$

6. if $E(i+1) \neq \emptyset$ then $i := i + 1$ and go to step 1, otherwise output $R(i+1)$.

Note that the procedure does not always halt. Suppose that the procedure halts successfully at $i+1 = k$ (hence $E(k) = \emptyset$). Then, $R(k)$ is convergent, and $R(k)$ satisfies $\rightarrow_1 = \rightarrow_2 \Rightarrow R(k)$ [3]. Note that when there is no rule to select at the Orientation step, the procedure halts in failure. Note that Composition and Collapse are used for efficiency, and the resulting systems are convergent even if Composition and Collapse are skipped.

### 3 Unraveled TRSs with Call-by-Value Interpretation

In this section, we propose a notion of operationally innermost reduction of DCTRSs that corresponds to call-by-value interpretation of functional programs, and we show that simulation-completeness with respect to innermost reduction is preserved by
Ohlebusch’s unraveling if the DCTRSs are restricted to functional programs having $\mathbf{let}$-like structures.

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

**Definition 3.1** Let $R$ be a DCTRS over a signature $\mathcal{F}$. For every conditional rewrite rule $\rho : l \rightarrow r \iff s_1 \land \cdots \land s_k \rightarrow t_k$, let $|\rho|$ denote the number $k$ of conditions in $\rho$. For every conditional rule $\rho \in R$, we prepare $k$ ‘fresh’ function symbols $U_1^\rho, \ldots, U_k^\rho$ not in $\mathcal{F}$, called $U$ symbols, in the transformation. We transform $\rho$ into a set $\mathbb{U}(\rho)$ of $k + 1$ unconditional rewrite rules as follows:

$$\mathbb{U}(\rho) = \big\{ l \rightarrow U_1^\rho(s_1, \vec{X}_1), U_1^\rho(t_1, \vec{X}_1) \rightarrow U_2^\rho(s_2, \vec{X}_2), \ldots, U_k^\rho(t_k, \vec{X}_k) \rightarrow r \big\}$$

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

Note that the definition of $\mathbb{U}$ is essentially equivalent to that in [26,29].

An unraveling $U$ is simulation-sound (simulation-preserving and simulation-complete, respectively) for a DCTRS $R$ over $\mathcal{F}$ if $\rightarrow_R \subseteq \rightarrow_{\mathbb{U}(R)}$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$ ($\rightarrow_R \supseteq \rightarrow_{\mathbb{U}(R)}$ and $\rightarrow_R = \rightarrow_{\mathbb{U}(R)}$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$, respectively). Note that the simulation-preserving property is sometimes called simulation-completeness in some papers, and it is a necessary condition of being unravelings. Roughly speaking, the computational equivalence is equivalent to the combination of simulation-completeness and normal-form uniqueness. The unraveling $\mathbb{U}$ is not simulation-sound for every DCTRS [27]. To avoid this difficulty of non-simulation-soundness of $\mathbb{U}$, a restriction to the rewrite relations of the unraveled TRSs is shown in [29], which is done by the context-sensitive condition given by the replacement map $\mu$ such that $\mu(U_i^\rho) = \{ 1 \}$ for every $U_i^\rho$ in Definition 3.1. We denote the context-sensitive TRS $(\mathbb{U}(R), \mu)$ by $\mathbb{U}_{cs}(R)$. We consider $\mathbb{U}_{cs}$ as an unraveling from DCTRSs to context-sensitive TRSs.

**Theorem 3.2** ([29]) For every DCTRS $R$ over $\mathcal{F}$, $\mathbb{U}_{cs}$ is simulation-complete, that is, $\rightarrow^*_{R} = \rightarrow^{*}_{\mathbb{U}_{cs}(R)}$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$.

To apply completion procedures to unraveled TRSs, we expect that the unraveling $\mathbb{U}$ is simulation-complete without the context-sensitivity. To this end, we propose an ‘innermost-like’ reduction of DCTRSs, called operationally innermost reduction. Let $R$ be an operationally terminating (OP-SN) DCTRS. The $n$-level operationally innermost reduction $\longrightarrow_{(n),i}^* R$ is defined as follows:

- $\longrightarrow_{(0),i}^* R = \emptyset$, and
- $\longrightarrow_{(n+1),i}^* R = \longrightarrow_{(n),i}^* R \cup \{ (C[l\sigma], C[r\sigma]) | l \rightarrow r \iff s_1 \rightarrow t_1 \land \cdots, s_k \rightarrow t_k \in R, \forall u < l\sigma. u \in NF \rightarrow (\mathcal{F}, \mathcal{V}), \forall i. s_i \sigma \rightarrow_{(n),i}^* t_i \sigma \}$.  

The operationally innermost reduction $\longrightarrow_{i}^* R$ of $R$ is defined as $\bigcup_{i \geq 0} \longrightarrow_{(i),i}^* R$. Note that if $R$ is a TRS then $\longrightarrow_{i}^* R$ is equivalent to the ordinary innermost reduction. Note that
the ordinary definition of innermost reduction is not well-defined for every CTRS
[8]. However, both the ordinary and operationally innermost reductions of OP-SN
CTRSs are well-defined. \( R \) is called \textit{innermost-confluent (innermost-convergent)} if
\[ \frac{s}{R} \text{ is confluent.} \]

Let \( R \) be a DCTRS. Terms in \( \{ u_1, \ldots, u_n, t_1, \ldots, t_k \mid f(u_1, \ldots, u_n) \rightarrow r \leftarrow s_1 \rightarrow t_1 \land \cdots \land s_k \rightarrow t_k \in R \} \setminus \mathcal{V} \) are called\( \textit{patterns} \) (in \( R \)). We denote the set of patterns in \( R \) by \( \text{Pat}(R) \). It follows from the definition of \( U \) that \( \text{Pat}(R) = \text{Pat}(U(R)) \) up to variable renaming. Patterns represents structures of data by means of matching. Especially, in innermost reductions, patterns matches normal forms only.

Unfortunately, \( U_{cs} \) is not simulation-preserving for every DCTRS with respect to the normalizing innermost reduction \( \frac{s}{R} \). This is because not all normal form of \( R \) are normal form of \( U_{cs}(R) \), that is, \( NF \rightarrow_R (\mathcal{F}, \mathcal{V}) \not\subseteq NF \rightarrow_{U_{cs}(R)} (\mathcal{F}, \mathcal{V}) \). To preserve the simulation-preserving property, \( U_{cs}(R) \) must have the same pattern-matching capability with \( R \), that is, if an instance \( \rho \theta \) of a pattern \( p \) is irreducible by \( R \) then \( \rho \theta' \) is also irreducible by \( U_{cs}(R) \) for every substitution \( \theta' \) such that \( x \theta \frac{s}{R} \frac{x'}{U_{cs}(R)} x \theta' \) for all \( x \in \text{Dom}(\theta) \). When all patterns are constructor terms (that is, \( U(R) \) is a constructor system), \( R \) and \( U(R) \) have the same pattern-matching capability. However, in examples of program inversion, a primitive operator \( \text{du} \) that requires\( \textit{equality check} \) is used: \( \text{du}(\langle x \rangle) = \langle x, x \rangle \), \( \text{du}(\langle x, x \rangle) = \langle x \rangle \), and \( \text{du}(\langle x, y \rangle) = \langle x, y \rangle \) if \( x \neq y \). This operator is encoded as the following terminating TRS:

\[
R_{\text{du}} = \left\{ \begin{array}{l}
\text{Du}(\langle x \rangle) \rightarrow \langle x, x \rangle, \\
\text{Du}(\langle x, y \rangle) \rightarrow \text{EqChk}(\text{EQ}(x, y)), \\
\text{EqChk}(\langle x \rangle) \rightarrow \langle x \rangle, \\
\text{EqChk}(\text{EQ}(x, y)) \rightarrow \langle y, x \rangle, \\
\text{EQ}(x, x) \rightarrow \langle x \rangle
\end{array} \right.
\]

Note that any system containing \( \text{Du} \) is not a constructor system. Since \( R_{\text{du}} \) has no\( \textit{overlay} \), \( R_{\text{du}} \) is\( \textit{locally innermost-confluent} \), and hence, \( R_{\text{du}} \) is\( \textit{innermost-confluent} \) [12]. Under the innermost reduction, \( R_{\text{du}} \) can simulate computation of \( \text{du} \).

One of the sufficient conditions to have the same pattern-matching capability is to satisfy all of the following conditions:

\begin{itemize}
  \item all rules defining \( g \in \{ g \in \mathcal{D}_R \mid g \text{ appears in } \text{Pat}(R) \} \) are unconditional and every proper subterm of the left-hand sides is a variable, and
  \item every rule \( l \rightarrow r \leftarrow s_1 \rightarrow t_1 \land \cdots \land s_k \rightarrow t_k \) represents a\( \textit{let-like structure} \), that is, \( \text{Var}(t_i) \cap \text{Var}(l, t_1, \ldots, t_{i-1}) = \emptyset \) for \( 1 \leq i \leq k \).
\end{itemize}

We call \( R \) \textit{pattern-stable} if \( R \) satisfies all of these conditions. The\( \textit{let-like structure} \) guarantees that \( \text{Var}(t_i) \cap \{ x_1, \ldots, x_n \} = \emptyset \) for every \( U_{cs}(t_i, x_1, \ldots, x_n) \) [23,25]. Pattern-stability is essential for DCTRSs that are used for modeling functional programs with\( \textit{let-like structures} \) and equality check.

\textbf{Theorem 3.3} \textit{Let} \( R \) \textit{be a pattern-stable OP-SN DCTRS over a signature} \( \mathcal{F} \), \textit{and} \( s \) \textit{and} \( t \) \textit{be terms in} \( T(\mathcal{F}, \mathcal{V}) \). \textit{Then}, \( s \xrightarrow{i}^* R t \text{ implies } s \xrightarrow{i}^* U_{cs}(R) u \) \textit{for some} \( u \) \textit{in} \( T(\mathcal{F}, \mathcal{V}) \) \textit{such that } \( t \xrightarrow{i}^* U_{cs}(R) u \).

Pattern-stability is also a sufficient condition for simulation-soundness. On the other hand, the non-erasing property of \( R \) is another sufficient condition. Here, we
call $R$ strongly non-erasing if every rule $l \rightarrow r \leftarrow s_1 \rightarrow t_1 \land \cdots \land s_k \rightarrow t_k$ satisfies all of the following conditions [23,25]:

- $\text{Var}(l) \subseteq \text{Var}(r, s_1, t_1, \ldots, s_k, t_k)$, and
- $\text{Var}(t_i) \subseteq \text{Var}(r, s_{i+1}, t_{i+1}, \ldots, s_k, t_k)$ for $1 \leq i \leq k$.

Any U symbol is not consumed by pattern-matching. The non-erasing property guarantees that no normal form containing U symbols appears along the reduction $s \xrightarrow{R} t \in T(\mathcal{F}, \mathcal{V})$; if a normal form containing a U symbol appears in the sequence, the non-erasing property ensures that it remains in $t$.

**Theorem 3.4** Let $R$ be a pattern-stable or strongly non-erasing OP-SN DCTRS over a signature $\mathcal{F}$, and $s$ and $t$ be terms in $T(\mathcal{F}, \mathcal{V})$. Then, $s \xrightarrow{R} t$ implies $s \xrightarrow{Ucs(R)} t$.

Context-sensitivity is not necessary for innermost reduction of $Ucs(R)$.

**Theorem 3.5** For every DCTRS $R$ over $\mathcal{F}$, 
\[ \frac{s}{i} \xrightarrow{Ucs(R)} = \frac{s}{i} \xrightarrow{Ucs(R)} \quad \text{in} \quad T(\mathcal{F}_U, \mathcal{V}) \times T(\mathcal{F}, \mathcal{V}). \]

Thanks to Theorem 3.5, when evaluating terms by the innermost reduction of $Ucs(R)$, we can treat $U(R)$ without the context-sensitivity determined by $U$.

For pattern-stable OP-SN DCTRSs, we have the following simulation-soundness and weak simulation-preserving property.

**Corollary 3.6** Let $R$ be a pattern-stable OP-SN DCTRS over a signature $\mathcal{F}$, and $s$ and $t$ be terms in $T(\mathcal{F}, \mathcal{V})$. Then, 

(i) $s \xrightarrow{R} t$ implies $s \xrightarrow{U(R)} u$ for some $u$ in $T(\mathcal{F}_U, \mathcal{V})$ such that $t \xrightarrow{U(R)} u$, and 

(ii) $s \xrightarrow{U(R)} t$ implies $s \xrightarrow{U(R)} t$.

Corollary 3.6 does not mean that $s \xrightarrow{U(R)} u$ implies $s \xrightarrow{R} t$ for some $t$ such that $t \xrightarrow{U(R)} u$ and $t$ is a normal form of $R$. This weakness of the simulation-preserving property does not happen when $U(R)$ is innermost-confluent. Therefore, getting innermost-confluence is important for unraveled TRSs.

### 4 Completion of Unraveled TRSs

In this section, by adding a side condition to ORIENTATION, we propose a modification of the ordinary Knuth-Bendix completion procedure for the unraveled TRSs of convergent DCTRSs. The modified procedure transforms the unraveled TRSs into convergent TRSs that are computationally equivalent to the DCTRSs. Moreover, to adapt the modified procedure to DCTRSs that are not confluent but innermost-confluent, we add another side condition to ORIENTATION.

The usual purpose of completion procedures is to generate convergent TRSs that are equivalent to given equation sets. In contrast to the usual purpose, we expect completion procedures to transform unraveled TRSs $U(R)$ into convergent TRSs that are computationally equivalent to the original DCTRSs $R$. To this
Due to the side condition of \( \text{InvSnoc} \) resulting TRS and R
\( \forall l \mapsto r \in \mathbb{U}(R) \mid \exists l' \mapsto r' \in \mathbb{U}(R), l \nsucc l' \) where \( \mathbb{U}(R) \subseteq \succ \). Moreover, consistency of the normal forms of \( \mathbb{U}(R) \) (that is, they are also normal forms of the resulting system) is necessary for preserving computational equivalence of \( R \). For this requirement, we add the side condition ‘root\((s)\) is a U symbol’ to ORIENTATION:

1. (ORIENTATION\((i)\)) select \( s \approx t \in E(i) \) such that \( s \succ t \) and \( \text{root}\((s)\) is a U symbol;

Due to the side condition of ORIENTATION\((i)\), and due to the basic characteristic of the ordinary completion procedure \( [3] \), the modified completion procedure produces convergent TRSs that are computationally equivalent to the input TRSs when it halts successfully.

**Theorem 4.1** Let \( R \) be an OP-SN DCTRS over \( \mathcal{F} \), and \( \succ \) be a reduction order such that \( \mathbb{U}(R) \subseteq \succ \). Let \( E_0 = \text{CP}(\mathbb{U}(R)) \), \( R_0 = \{ l \mapsto r \in \mathbb{U}(R) \mid \exists l' \mapsto r' \in \mathbb{U}(R), l \nsucc l' \} \), and \( R' \) be a TRS obtained by the modified completion procedure from \( (E_0, R_0) \) with \( \succ \). Then, (1) \( R' \) is convergent, (2) \( \text{NF} \rightarrow_{\mathbb{U}(R)} (\mathcal{F}, \mathcal{V}) = \text{NF} \rightarrow_{R'} (\mathcal{F}, \mathcal{V}) \), and (3) \( \frac{*}{\succ}^{(i)}_{\mathbb{U}(R)} \) = \( \frac{*}{\succ}^{(i)}_{R'} \) in \( T(\mathcal{F}, \mathcal{V}) \).

Since \( \text{NF} \rightarrow_s (\mathcal{F}, \mathcal{V}) = \text{NF} \rightarrow_{s} (\mathcal{F}, \mathcal{V}) \) (\( S \) is either \( \mathbb{U}(R) \) or \( R \)), it holds in Theorem 4.1 that \( \frac{*}{\succ}^{(i)}_{\mathbb{U}(R)} = \frac{*}{\succ}^{(i)}_{R'} \) in \( T(\mathcal{F}, \mathcal{V}) \).

**Example 4.2** Consider the non-convergent TRS \( \mathbb{U}(\text{Inv}(R_1)) \) in Section 1 again. Given the lexicographic path order (LPO) \( \succ_{\text{LPO}} \) determined by the precedence \( \succ \) with \( \text{InvSnoc} \succ U_1 \succ :: \succ \text{nil} \succ (\ ) \), we obtain the following convergent and non-overlapping TRS by the modified completion procedure (in 4 cycles):

\[
R_2 = \begin{cases}
\text{InvSnoc}(x::ys) \mapsto U_1(\text{InvSnoc}(ys), x, ys), \\
U_1((xs, y), x, ys) \mapsto (x::xs, y), \\
U_1(\text{InvSnoc}(\text{nil}), x, \text{nil}) \mapsto (\text{nil}, x)
\end{cases}
\]

Since the procedure removes the rule \( \text{InvSnoc}([y]) \mapsto (\text{nil}, y) \) from \( \mathbb{U}(\text{Inv}(R_1)) \), the resulting TRS \( R_2 \) is non-overlapping.

Unfortunately, the modified completion procedure does not always halt even if the inputs are restricted to unraveled TRSs. For example, the modified procedure does not halt for the unraveled TRS obtained from Example 7.1.5 in [27] although there exists an appropriate convergent TRS that is computationally equivalent to the corresponding DCTRS.

Confluence of \( R \) is necessary for the modified completion procedure to halt ‘successfully’. Note that confluence of \( R \) is not sufficient for the procedure to ‘halt’. In other words, the procedure halts (or keeps running) ‘unsuccessfully’ if \( R \) is not confluent. If \( R \) is not confluent, then we have \( t_1 \overset{*}{\rightarrow}_{\mathbb{U}(R)} s \overset{*}{\rightarrow}_{\mathbb{U}(R)} t_2 \) and \( t_1 \not\equiv t_2 \) for some \( s, t_1 \) and \( t_2 \) in \( T(\mathcal{F}, \mathcal{V}) \). The added side condition ‘root\((s)\) is a U symbol’ prevents \( t_1 \) and \( t_2 \) from being joinable. From this observation, the modified procedure can be considered as a method to show confluence of \( R \): if the procedure succeeds, then \( R \) is confluent.

As stated above, we would like to transform DCTRSs on call-by-value interpretation into convergent TRSs that are computationally equivalent to the DCTRSs.
Moreover, the modified completion procedure always fails for DCTRSs that are not confluent but innermost-confluent, such as DCTRSs containing $R_{du}$.

To obtain innermost-convergent systems that are computationally equivalent to TRSs containing $R_{du}$, applying completion procedures to the TRSs appears to be effective just as in the case of convergent TRSs. However, there is a difficulty associated with innermost reduction. The difficulty is that innermost reduction is not closed under substitutions. When applying the completion procedure to $R_{du}$, the rules $Du((x, y)) \rightarrow \text{EqChk}(\text{EQ}(x, y))$ is transformed into $Du((x, y)) \rightarrow (x, y)$. Given a ground normal form $t$, the resulting system cannot simulate the reduction $Du(t, t) \xrightarrow{s} R_{du} \langle t \rangle$ due to the lack of $Du((x, y)) \rightarrow \text{EqChk}(\text{EQ}(x, y))$. To remove this troublesome problem from the modified completion procedure for innermost reduction, we prohibit the procedure to use the two operations COMPOSITION and SIMPLIFICATION, and give an additional side condition to ORIENTATION‡ as follows:

1. (ORIENTATION‡) select $s \approx t \in E(i)$ such that $s > t$, $\text{root}(s)$ is a U symbol, and $\text{CP}(\{s \rightarrow t\}, R(i) \cup \{s \rightarrow t\}) = \emptyset$;

The additional condition means that the oriented rule $s \rightarrow t$ is not overlapping with other rules in $R(i) \cup \{s \rightarrow t\}$. Thus, DEDUCTION does not add any equations to the equation set $E(i)$ but removes an equation. Since no U symbol appears in the left-hand side $l$ in $T(F, V)$ from the definition of $U$, and since the added rules are not overlapping with other rules, COLLAPSE removes no rules from the rule set $R(i)$. If $E$ has no equation of the form $s \approx s$, DELETION step removes no equations from the equation set. From this observation, the modified procedure with ORIENTATION‡ is simplified as Definition 4.3 shown later.

Before simplifying the modified procedure, we describe the relation between $U(R)$ and $S$ with respect to the innermost reduction. No rule $l \rightarrow r \in U(R)$ such that there exists a rule $l' \rightarrow r'$ with $l \triangleright l'\theta$ for some substitution $\theta$ is used in $\overline{\text{i}} U(R)$ because no instance of $l$ is an innermost redex. For this reason, we restrict the initial set of rules to $U(R) \setminus S$. Roughly speaking $U(R) \setminus S$ is the set of rules that are usable for $\overline{\text{i}} U(R)$.

**Definition 4.3** Let $R$ be an OP-SN DCTRS over $F$, and $\triangleright$ be a reduction order such that $U(R) \subseteq \triangleright$. Let $S = \{ l \rightarrow r \in U(R) \mid \not\exists l' \rightarrow r \in U(R), l \triangleright l' \}, E(0) = \{ s \approx t \mid s \approx t \in CP_\varepsilon(U(R) \setminus S), s \neq t \}, R(0) = \{ l \rightarrow r \in U(R) \setminus S \mid \not\exists l' \rightarrow r \in U(R) \setminus S, l \triangleright l' \}$, and $i = 0$, then we apply the following steps:

1. (ORIENTATION‡) select $s \approx t \in E(i)$ such that $s > t$, $\text{root}(s)$ is a U symbol, and $\text{CP}(\{s \rightarrow t\}, R(i) \cup \{s \rightarrow t\}) = \emptyset$;
2. $R(i+1) := \{s \rightarrow t\} \cup R(i)$, and $E(i+1) := E(i) \setminus \{s \approx t\}$;
3. if $E(i+1) \neq \emptyset$ then $i := i + 1$ and go to step 1, otherwise output $R(i+1)$.

We call this procedure the **simplified completion procedure**.

It is clear that $E(i) \supset E(i+1)$ for every $i \geq 0$. Therefore, the simplified completion procedure always halts. Note that the simplified procedure does not succeed for all input.
Theorem 4.4 Let \( R \) be a pattern-stable OP-SN DCTRS over \( \mathcal{F} \), and \( \succ \) be a reduction order such that \( \mathcal{U}(R) \subseteq \succ \). Let \( R' \) be a TRS obtained by the simplified completion procedure from \( R \) and \( \succ \). Let \( R' \) be a TRS obtained by the simplified completion procedure from \( R \) and \( \succ \). Then all of the following hold: (1) \( R' \) is innermost-convergent, (2) \( \text{NF} \rightarrow_{\mathcal{U}(R)} (\mathcal{F}, \mathcal{V}) = \text{NF} \rightarrow_{R'} (\mathcal{F}, \mathcal{V}) \), and (3) \( \text{NF} \rightarrow_{i}^{\ast} \mathcal{U}(R) = \text{NF} \rightarrow_{i}^{\ast} R' \) in \( T(\mathcal{F}, \mathcal{V}) \).

Note that (1) and (3) implies (2). The simplified procedure succeeds for \( \mathcal{U}(\text{Inv}(R_1)) \) as well as for Example 4.2.

Similarly to the modified completion procedure, innermost-confluence of \( R \) is necessary for the simplified completion procedure to halt ‘successfully’. Therefore, the simplified procedure is a method to show innermost-confluence of \( R \);

5 Completion after Program Inversion

In this section, we apply the modified and simplified completion procedures to DCTRSs generated by the partial inversion compiler [25], that is, we apply the procedures as a post-processor of \( \mathcal{U}(\text{Inv}(\cdot)) \) to the unraveled TRSs. First, we briefly introduce the feature of inverse systems for injective functions. Then, we show the results of experiments by an implementation of the framework.

We employ the partial inversion \( \text{Inv} \) in [25] that generates a partial inverse CTRS from a pair of a given constructor TRS and a specification, which 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 constructor TRSs define main injective functions, and that the specifications require full inverses of the main functions.

5.1 Inverse DCTRSs of Injective Functions

We first define injectivity of TRSs [22], and then give a sufficient condition for input constructor TRSs whose inverse DCTRSs generated by \( \text{Inv} \) are convergent.

Definition 5.1 Let \( R \) be a terminating and innermost-confluent constructor TRS. A defined symbol \( F \) of \( R \) is called injective (with respect to normal forms) if the binary relation \( \{ ((s_1, \ldots, s_n), t) \mid s_1, \ldots, s_n, t \in \text{NF} \rightarrow_R (\mathcal{F}, \mathcal{V}), F(s_1, \ldots, s_n) \rightarrow_R t \} \) is an injective mapping. \( R \) 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:

\[
R_4 = \begin{cases} 
\text{Reverse}(xs) \rightarrow \text{Rev}(xs, \text{nil}), \\
\text{Rev}(\text{nil}, ys) \rightarrow ys, \\
\text{Rev}(x :: xs, ys) \rightarrow \text{Rev}(xs, x :: ys)
\end{cases}
\]

\( \text{Reverse} \) is injective but \( \text{Rev} \) is not. Thus, \( R_4 \) is not injective. In this case, the inverse TRS \( \mathcal{U}(\text{Inv}(R_4)) \) is not terminating because \( \mathcal{U}(\text{Inv}(R_4)) \) contains the rule
\text{InvRev}(z) \rightarrow U_4(\text{InvRev}(z), z). \text{ 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 [24] to generate convergent inverse DCTRSs from injective TRSs. The condition is also effective for the partial inversion compiler Inv [25].}

\textbf{Theorem 5.2} \textit{Let } R \textit{ be a non-erasing, terminating and innermost-confluent constructor TRS.}

(i) \textit{If } F \in D_R \textit{ is injective, then for all } t, t_1 \textit{ and } t_2 \in NF \xrightarrow{\text{Inv}(R)} (F,V), t_1 \xrightarrow{*_{\text{Inv}(R)}} t_2 \textit{ implies } t_1 \equiv t_2.

(ii) \textit{Suppose that for every rule } F(u_1,\ldots,u_n) \rightarrow r \textit{ in } R, \textit{if } r \textit{ is not a variable then the root symbol of } r \textit{ does not depend}^4 \textit{ on } F. \textit{ If } Inv(R) \cap R = \emptyset \textit{ then the DCTRS Inv}(R) \textit{ is OP-SN.}

\textbf{Note that if the DCTRS Inv}(R) \textit{ is OP-SN then the TRS } U(Inv(R)) \textit{ is terminating [14]. Theorem 5.2 (i) shows that if Inv}(R) \textit{ is OP-SN, then Inv}(R) \textit{ has innermost-confluence that is necessary for successful runs of the simplified completion procedure. Note that Inv}(R) \textit{ is confluent if } R \textit{ is convergent [25]. When } R \textit{ does not satisfy the condition in Theorem 5.2 (ii), we directly check the termination of } U(Inv(R)). \textit{ In other words, when } R \textit{ satisfies the condition in Theorem 5.2 (ii), we are free of the termination check of } U(Inv(R)) \textit{ that is less efficient than the check of satisfying the condition.}

\textbf{5.2 Experiments}

In this subsection, we report the results of applying implementations of the modified and simplified completion procedures to 10 of 15 examples shown in [9].\textsuperscript{5} \textbf{These 15 examples are introduced for the experiments of the inversion compiler LRinv} [9,10] \textbf{where LRinv succeeds in inverting all of them. Those examples are written in the scheme script Gauche. The inverse TRSs of the scripts \texttt{snoc}, \texttt{snocrev} and \texttt{reverse} correspond to the TRSs } U(Inv(R_1)), R_3 \textbf{ and } U(Inv(R_4)), \textit{respectively. The constructor TRSs corresponding to the 5 scripts (\texttt{reverse} and so on) are not injective and the inverse TRSs obtained from them are not terminating. For this reason, we excluded those non-terminating examples from our experiments. For some examples, there exists no appropriate LPO to guarantee termination of the input TRSs. For this reason, we employ the termination check ‘}(\bigcup_{i=0}^{i} R_{(i)}) \cup \{s \rightarrow t\}\textit{ is terminating‘ instead of the input reduction orders, following the approach in [34]. The implementations are written in Standard ML of New Jersey, and they were executed under OS Vine Linux 4.2, on an Intel Pentium 4 CPU at 3 GHz and 1 GByte of primary memory. By the system call in SML/NJ, the implementations consult with AProVE 1.2 [6] as a termination prover at the ORIENTATION step.}

\footnotetext[4]{An \textit{n}-ary symbol } G \textit{ of } R \textit{ depends on a symbol } F \textit{ if } (G,F) \textit{ is in the transitive closure of the relation } \{(G',F') | G'(\cdots) \rightarrow C[F'(\cdots)] \in R\}.\textsuperscript{5} \textbf{Unfortunately, the site shown in [9] is not accessible now. The examples are also described briefly as functional programs in [10], and some of the detailed programs can be found in [10].}
implementations check termination of input TRSs in advance of the completion procedures. The timeout for checking termination is 300 seconds in every call of the prover. Note that 60 seconds timeout is enough, except for treelist.

The examples (double, mirror and print-xml) contain the special primitive operator du described in Section 4. Hence, they are not confluent but innermost-confluent. The operator du is an inverse of itself [9,10]. Thus, the TRS $R_{du}$ is also an inverse system of itself. For this reason, exceptionally, the inversion compiler does not produce any rules of InvDup but introduces Du instead of InvDup.

Due to the syntactic properties provided by the inversion compiler, all inverse DCTRSs in the experiments are pattern-stable and strongly non-erasing. Thus, the procedures in this paper are applicable to all of them.

Table 1 summarizes the results of the experiments for our approach running on 10 of the 15 examples previously mentioned, which were translated by hand into TRSs. The second column labeled with ‘CR by [2]’ shows whether the input TRS of the example is in the class shown in [2], in which the corresponding inverse TRS is orthogonal and thus confluent. In that case, the implementations only check termination of the inverse TRS. The third column labeled with ‘SN by Th. 5.2’ shows whether the input TRS satisfies the conditions in Theorem 5.2 (ii), that is, the corresponding inverse TRSs are terminating. Columns 4–6 show the results of the modified completion procedure. The fourth column shows the results of the modified completion (‘success’ or fail’) with the numbers of running ‘cycles’ in the sense of Definition 2.1, and with the average time (seconds) of 5 trials. The number of cycles is the same as the number of applications of Orientation. As described above, the implementation checks the termination of input TRSs before the completion procedure starts. Thus, we have the results ‘success (0c,· · ·) and 1 call of provers’. The sixth column labeled with ‘¬OVL’ shows whether or not the resulting TRSs are non-overlapping (√ means the resulting is non-overlapping, and ‘—’ means no resulting TRS). None of the resulting TRSs has overlays while some of them are overlapping. Columns 7–9 show the results of the simplified completion procedure.

Table 1

<table>
<thead>
<tr>
<th>example</th>
<th>CR by [2]</th>
<th>SN by Th.5.2</th>
<th>modified completion proc.</th>
<th>simplified completion proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>result (cycles, time)</td>
<td>call</td>
<td>¬OVL</td>
<td>result</td>
</tr>
<tr>
<td>du</td>
<td>fail (1c, 0.71s)</td>
<td>1</td>
<td>—</td>
<td>success (0c, 0.71s)</td>
</tr>
<tr>
<td>snoc</td>
<td>√</td>
<td>success (1c, 2.08s)</td>
<td>2</td>
<td>√</td>
</tr>
<tr>
<td>snocrev</td>
<td>√</td>
<td>success (2c, 4.29s)</td>
<td>3</td>
<td>√</td>
</tr>
<tr>
<td>double</td>
<td>fail (1c, 2.84s)</td>
<td>2</td>
<td>—</td>
<td>success (1c, 2.83s)</td>
</tr>
<tr>
<td>mirror</td>
<td>fail (3c, 5.97s)</td>
<td>3</td>
<td>—</td>
<td>success (2c, 5.96s)</td>
</tr>
<tr>
<td>zip</td>
<td>√</td>
<td>√</td>
<td>success (0c, 1.04s)</td>
<td>1</td>
</tr>
<tr>
<td>inc</td>
<td>√</td>
<td>success (1c, 2.80s)</td>
<td>2</td>
<td>√</td>
</tr>
<tr>
<td>octbin</td>
<td>√</td>
<td>success (0c, 7.33s)</td>
<td>1</td>
<td>√</td>
</tr>
<tr>
<td>treelist</td>
<td>√</td>
<td>success (4c, 159.02s)</td>
<td>5</td>
<td>fail (2c, 5.47s)</td>
</tr>
<tr>
<td>print-sexp</td>
<td>success (6c, 28.20s)</td>
<td>7</td>
<td>√</td>
<td>success (6c, 28.28s)</td>
</tr>
<tr>
<td>print-xml</td>
<td>fail (3c, 9.49s)</td>
<td>3</td>
<td>—</td>
<td>success (2c, 9.47s)</td>
</tr>
</tbody>
</table>

6 The detail will be available from “http://www.trs.cm.is.nagoya-u.ac.jp/repius/experiments/”. 

procedure, and the meaning of those columns is the same as columns 4–6.

5.3 Translation Back into Functional Programs

In general, it is difficult to decide a priority of rewrite rules. However, 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. On the other hand, every convergent constructor TRS can be easily translated back into a functional program. However, it is not easy to translate convergent TRSs that are not constructor systems, into functional programs even if the TRSs are non-overlapping. The reason is that some rules contains non-‘well-formed’ patterns in their left-hand sides, for instance, $\text{InvSnoc}(\text{nil})$ in $U(\text{Inv}(R_1))$.

In this subsection, we show a translation from $R_2$ into a SML program. Such a translation has not been automated yet but we believe that the automation is feasible.

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

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

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

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

Thus, we translate the TRS $R_2$ into the following program:

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

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

In all of the 10 examples, we succeeded in translating by hand the resulting convergent TRSs back into SML programs by means of the mechanism in this subsection although the resulting systems of double, mirror, treelist, and print-xml have overlapping.

6 Concluding Remarks

In this paper, we have shown that completion procedures are useful in generating (innermost-)convergent inverse TRSs of injective TRSs. The completion procedures can be also used for checking whether or not a (innermost-)convergent constructor TRS is injective. This is because if a given convergent constructor TRS is not injective, then the procedures never succeeds for the TRS. It is known to be undecidable
in general whether or not a function is injective [5]. In [17], however, it is shown that injectivity of linear treeless functions is decidable. On the other hand, some of the examples we mentioned in the experiments are non-linear or non-treeless while the method in this paper is not decidable.

Completion procedures are effective for solving word problems, for transforming equations into equivalent convergent systems, or for proving inductive theorems. 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 modified completion procedure in this paper does not succeed for every confluent and OP-SN DCTRSs while the latest transformation [30] based on Viry’s approach [33] always succeeds. Consider the example in Section 1 again. By the transformation in [30], we obtain the following convergent TRS instead of $U(Inv(R_1))$:  

\[
\begin{align*}
\text{InvSnoc}([y], z) &\rightarrow \{(\text{nil}, y)\}, \\
\text{InvSnoc}(x::ys, \bot) &\rightarrow \text{InvSnoc}(x::ys, \{\text{InvSnoc}(ys, \bot)\}), \\
\text{InvSnoc}(x::ys, \{(xs, y)\}) &\rightarrow \{(x::xs, y)\}, \\
\text{InvSnoc}(\{xs\}, z) &\rightarrow \{\text{InvSnoc}(xs, \bot)\}, \quad \{x\} \rightarrow \{x\}
\end{align*}
\]  

$\cup \{ c(x_1, \ldots, \{x_1\}, \ldots, x_n) \rightarrow \{c(x_1, \ldots, x_n)\} \mid c \in \{::, \langle, \rangle\} \}$

where $\{ \}$ and $\bot$ are special function symbols not in the original signature. In this system, the term $\text{InvSnoc}([a, b, c], \bot)$ has a unique normal form $\{(a, b, c)\}$. As described in Section 1, however, it is difficult to translate the convergent TRS into a functional program because the system contains special symbols $\{ \}$ and $\bot$, and overlapping rules. On the other hand, the modified completion procedure in this paper unexpectedly succeeded for all the experiments where the DCTRSs are confluent, and the resulting systems of the procedure are often non-overlapping. Moreover, for the DCTRSs that are not confluent but innermost-confluent, we proposed the simplified completion procedure but it is not yet known whether or not the transformation in [30] is applicable.

The inversion compiler LRinv, the closest one to the method in this paper, has been proposed for injective functions written in a functional language [9,7,10]. 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 makes it difficult to compare LRinv and our method. However, LRinv succeeds in generating inverse functions from the 5 scripts (reverse and so on) that we excluded from the experiments, where the main functions call non-injective functions such as the accumulator Rev. From this fact, LRinv seems to be stronger than the method in this paper but there must be plenty of room on improving the principle of inversion used in the partial inversion compiler in [25]. As future work, we plan to extend the partial inversion compiler.
for functions with accumulators such as $\text{Rev}$, and we also improve the modified and simplified completion procedures.

Acknowledgement

We are grateful to Germán Vidal and the anonymous reviewers for valuable comments for improving this paper. We also thank Tsubasa Sakata and Kazutoshi Seki for discussing correctness of the completion procedure on innermost reduction, and for helping the experiments. This work is partly supported by MEXT. KAKENHI #18500011, #20300010 and #20500008 and Kayamori Foundation of Informational Science Advancement.

References


[21] Nishida, N. and M. Sakai, *Completion as Post-Process in Program Inversion of Injective Functions*, [http://www.trs.cm.is.nagoya-u.ac.jp/~nishida/papers/](http://www.trs.cm.is.nagoya-u.ac.jp/~nishida/papers/) (2008), the full version of this paper.


