Completion as Post-Process in 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 defining injective functions, the inversion compiler in [19,20] generates a confluent conditional term rewriting system defining completely the inverse relations of the injective functions, and then the compiler 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 propose a modification of Knuth-Bendix completion procedure as a post-process of the inversion compiler. Given a confluent and operationally terminating conditional system, the procedure takes the unraveled one of the conditional system as input, and it returns a convergent system that is computationally equivalent to the conditional system if it halts successfully. We also adapt the modification to the conditional systems that are not confluent but innermost-confluent. The implementation of our method succeeds in generating innermost-convergent inverse systems for all examples shown by Kawabe et al. where all main and axillary functions are injective.

Keywords: unraveling, convergence, functional programming, conditional term rewriting system

1 Introduction

Given a constructor TRS (term rewriting system), the inversion compiler proposed in [19,20] first generates a CTRS (conditional TRS) 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 a CTRS, called an *inverse system*, 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* [21]. *Unravelings* are transformations based on Marchiori's approach [14], that transform CTRSs into TRSs. Note that we call all variants of Marchiori's unravelings *unravelings* because they satisfy the condition [14,15] of being unravelings.

Unfortunately, the compiler cannot always generate TRSs that are computationally equivalent to the intermediate CTRSs due to a character of unravelings, that

[1] Email: nishida@is.nagoya-u.ac.jp
[2] Email: sakai@is.nagoya-u.ac.jp

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

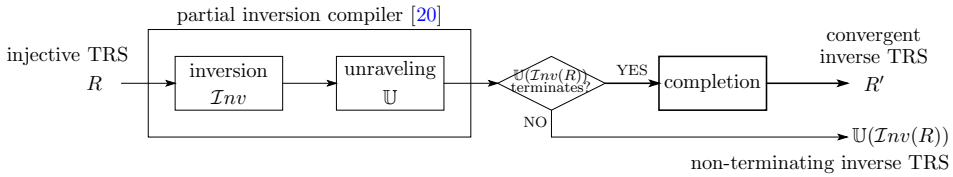

Fig. 1. Overview of the partial inversion with the completion.

is, the unraveled TRSs of CTRSs may have unexpected normal forms that represent dead ends of wrong choices at branches of evaluating conditional parts of the CTRSs. These wrong choices are captured by critical pairs of the unraveled TRSs, each of which originates two rewrite rules corresponding to the 'correct' and 'wrong' choices, where any rules looking like 'wrong choice' must be necessary elsewhere. Note that it is decidable whether a normal form is desired or unexpected: 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 compiler, this problem arises even if functions defined in given constructor TRSs are injective. For this reason, the resultant 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 the functional programs to constructor TRSs, but difficult to translate the resultant 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 CTRSs into convergent (and possibly non-overlapping) TRSs that are computationally equivalent to the CTRSs. Unfortunately, the procedure does not always halt. However, if the procedure halts successfully and the resultant convergent TRSs are non-overlapping, then the resultant systems can be easily translated back into functional programs due to non-overlappingness. We takes the modified completion procedure as a post-process into the program inversion of injective functions (Fig. 1 and Section 4). Through this approach, we show that unravelings are useful not only in analyzing properties but also in modifying programs (unraveled TRSs, especially inverse programs).

Consider the following functional program in Standard ML where $\mathsf{Snoc}(xs, y)$ 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:

$$R_1 = \{\, \mathsf{Snoc}(\mathsf{nil}, y) \to (y :: \mathsf{nil}),\quad \mathsf{Snoc}((x :: xs), y) \to (x :: \mathsf{Snoc}(xs, y)) \,\}$$

where $(t :: ts)$ abbreviates the list $\mathsf{cons}(t, ts)$. The compiler inverts R_1 into the following CTRS in the first phase[3]:

$$\mathcal{I}nv(R_1) = \begin{cases} \mathsf{InvSnoc}([y]) \to \langle \mathsf{nil}, y \rangle \\ \mathsf{InvSnoc}((x :: ys)) \to \langle (x :: xs), y \rangle \Leftarrow \mathsf{InvSnoc}(ys) \to \langle xs, y \rangle \end{cases}$$

[3] To simplify discussions, we omit describing special rules in the form of $\mathsf{Inv}F(F(x_1, \ldots, x_n)) \to \langle x_1, \ldots, x_n \rangle$ [20,19] because they are meaningless for inverse computation in dealing with call-by-value systems. The special rules are necessary only for inverse computation of normalizing computation in term rewriting.

where $[t_1, t_2, \ldots, t_n]$ abbreviates the list $\mathsf{cons}(t_1, \mathsf{cons}(t_2, \cdots, \mathsf{cons}(t_n, \mathsf{nil})\cdots))$ and 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 CTRS $\mathcal{I}nv(R_1)$ into the following TRS in the second phase:

$$\mathbb{U}(\mathcal{I}nv(R_1)) = \left\{ \begin{array}{l} \mathsf{InvSnoc}([y]) \to \langle \mathsf{nil}, y\rangle, \\ \mathsf{InvSnoc}((x :: ys)) \to U_1(\mathsf{InvSnoc}(ys), x, ys), \\ U_1(\langle xs, y\rangle, x, ys) \to \langle (x :: xs), y\rangle \end{array} \right\}$$

The introduced symbol U_1 is used for evaluating the conditional part $\mathsf{InvSnoc}(ys) \to \langle xs, y\rangle$ of the second rule in $\mathcal{I}nv(R_1)$. The term $\mathsf{Snoc}([\mathsf{a}, \mathsf{b}], \mathsf{c})$ has a unique normal form $[\mathsf{a}, \mathsf{b}, \mathsf{c}]$ but $\mathsf{InvSnoc}([\mathsf{a}, \mathsf{b}, \mathsf{c}])$ has two normal forms, a solution $\langle[\mathsf{a}, \mathsf{b}], \mathsf{c}\rangle$ of inverse computation and an unexpected normal form $U_1(U_1(U_1(\mathsf{InvSnoc}(\mathsf{nil}), \mathsf{c}, \mathsf{nil}), \mathsf{b}, [\mathsf{c}]), \mathsf{a}, [\mathsf{b}, \mathsf{c}])$. In this example, it appears to be easy to translate from the CTRS $\mathcal{I}nv(R_1)$ into a functional program directly because we can easily determine an appropriate priority of conditional rules in $\mathcal{I}nv(R_1)$. However, such a direct translation is difficult in general because we cannot decide which rules have priority of the application to terms. The restricted compiler in [1] is useless for this case because R_1 is out of the scope. 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 [18] that the transformation in [24] 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 5).

Roughly speaking, non-confluence of $\mathbb{U}(\mathcal{I}nv(R_1))$ comes from the critical pair $\langle\langle \mathsf{nil}, x\rangle, U_1(\mathsf{InvSnoc}(\mathsf{nil}), x, \mathsf{nil})\rangle$ between the first and second rules in $\mathbb{U}(\mathcal{I}nv(R_1))$. In this case, the application of the first rule is the correct choice and that of the second is the wrong, that is, $\langle \mathsf{nil}, x\rangle$ is the correct result and $U_1(\mathsf{InvSnoc}(\mathsf{nil}), x, \mathsf{nil})$ is the wrong recursive call of U_1 containing the dead end $\mathsf{InvSnoc}(\mathsf{nil})$. From this observation, by adding the rule $U_1(\mathsf{InvSnoc}(\mathsf{nil}), x, \mathsf{nil}) \to \langle \mathsf{nil}, x\rangle$, the unexpected normal form of $\mathsf{InvSnoc}([\mathsf{a}, \mathsf{b}, \mathsf{c}])$ 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. 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.

This paper illustrates all of the following:

- under the *call-by-value* evaluation of operationally terminating deterministic CTRSs, *simulation-soundness* on the innermost reduction is preserved by Ohlebusch's unraveling (Subsection 3.2);
- given a (innermost-)confluent and operationally terminating CTRS, the completion procedure takes the unraveled TRS (evaluated by the innermost reduction) as input, and returns a (innermost-)convergent TRSs that are computationally equivalent to the CTRSs if the procedure halts successfully (Subsection 3.3);

- to deal with TRSs whose termination is not provable by any reduction orders without dependency analysis (e.g., lexicographic path orders), we employ the completion with termination provers, following the approach in [28] (Subsection 3.5).

We also show that an implementation of the completion procedure succeeds in generating convergent TRSs from all the unraveled TRSs of CTRSs obtained by the inversion compiler [19] from injective functions shown by Kawabe et al. [8] where all axillary functions are also injective, and we show an informal translation of the *non-overlapping* TRSs obtained by the procedure into functional programs (Subsection 3.4). Note that we do not consider *sorts*; however, the framework in this paper is easily extended to many-sorted systems.

2 Preliminaries

Here, we will review the following basic notations of term rewriting [2,22].

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 $\mathcal{V}ar(t_1,\ldots,t_n)$. 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* ε of t is denoted by $\mathtt{root}(t)$. The notation $C[t_1,\ldots,t_n]_{p_1,\ldots,p_n}$ represents the term obtained by replacing each $\Box$ 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* σ are denoted by $\mathcal{D}om(\sigma)$ and $\mathcal{R}an(\sigma)$, respectively. The application $\sigma(t)$ of substitution σ to t is abbreviated to $t\sigma$.

An *(oriented) conditional rewrite rule* over $\mathcal{F}$ is a triple (l,r,c), denoted by $l \to r \Leftarrow c$, 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 $s_1 \to t_1 \wedge \cdots \wedge s_n \to t_n$ $(n \geq 0)$ of terms s_i and t_i in $\mathcal{T}(\mathcal{F},\mathcal{V})$. In particular, the conditional rewrite rule $l \to r \Leftarrow c$ is said to be an *(unconditional) rewrite rule* if $n = 0$, and we may abbreviate it to $l \to r$. We sometimes attach a unique label ρ to a rule $l \to r \Leftarrow c$ by denoting $\rho : l \to r \Leftarrow c$, and we use the label to refer to the rule. To simplify notations, we may write labels instead of the corresponding rules. An *(oriented) conditional rewriting system* (*CTRS*, for short) R over a signature $\mathcal{F}$ is a finite set of conditional rewrite rules over $\mathcal{F}$. The *rewrite relation* of R is denoted by $\to_R$. To specify the applied position p and rule ρ, we write $\to_R^p$ or $\to_R^{[p,\rho]}$. A conditional rewrite rule $\rho : l \to r \Leftarrow s_1 \to t_1 \cdots s_k \to t_k$ is called *deterministic* if $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l,t_1,\ldots,t_k)$ and $\mathcal{V}ar(s_i) \subseteq \mathcal{V}ar(l,t_1,\ldots,t_{i-1})$ for $1 \leq i \leq k$. The CTRS R is called a *deterministic CTRS* (a *DCTRS* for short) if all rules in R are deterministic. *Operational termination* of DCTRSs is such that no infinite reductions exist in existing rewrite engines [13]: a CTRS R is *operationally terminating* (*OP-SN*, for short) if for any terms s and t, any proof tree attempting to prove that $s \stackrel{*}{\to}_R t$ cannot be infinite.

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 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 $\mathcal{D}_R = \{\mathtt{root}(l) \mid l \to r \Leftarrow c \in R\}$ 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 CTRS

R is called a *constructor system* if every rule $f(t_1,\ldots,t_n) \to r \Leftarrow c$ in R satisfies $\{t_1,\ldots,t_n\} \subseteq \mathcal{T}(\mathcal{C},\mathcal{V})$.

We use the notion of *context-sensitive reduction* in [12]. Let $\mathcal{F}$ be a signature. A *context-sensitive condition* (*replacement mapping*) μ is a mapping from $\mathcal{F}$ 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 *context-sensitive reduction* of the *context-sensitive TRS* (R,μ) of a TRS R and a *replacement map* μ is denoted by $\to_{(R,\mu)}$: $\to_{(R,\mu)} = \{(s,t) \mid s \to_R^p t, p \in \mathcal{O}_\mu(s)\}$. The innermost reduction of $\to_{(R,\mu)}$ is denoted by $\underset{\mathrm{i}}{\to}_{(R,\mu)}$: $\underset{\mathrm{i}}{\to}_{(R,\mu)} = \{(s,t) \mid s \to_R^p t, p \in \mathcal{O}_\mu(s), (\forall q > p.\ q \in \mathcal{O}(s)$ implies that $s|_q$ is irreducible$)\}$.

Let $l_i \to r_i$ $(i = 1,2)$ be two rules whose variables have been renamed such that $\mathcal{V}ar(l_1,r_1) \cap \mathcal{V}ar(l_2,r_2) = \emptyset$. Let p be a position in l_1 such that $l_1|_p$ is not a variable and let θ be a most general unifier of $l_1|_p$ and l_2. This determines a *critical pair* $\langle r_1\theta, (l_1\theta)[r_2\theta]_p\rangle$. 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 normal forms are computable (i.e., $\to_R$ and $\to_{R'}$ are well-defined), and T be a set of terms. Roughly speaking, R' is *computationally equivalent* to R with respect to T if there exist mappings ϕ and ψ 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 $\psi(t') = t$ [24]. In this paper, we assume that ϕ and ψ are the identity mappings.

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

3 Completion to Unraveled TRSs

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

3.1 Unraveling for DCTRSs

We first give the definition of Ohlebusch's unraveling [21]. Given a finite set X of variables, we denote by $\overrightarrow{X}$ 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 $\rho : l \to r \Leftarrow s_1 \to t_1 \wedge \cdots \wedge s_k \to t_k$, let $|\rho|$ denote the number k of conditions in ρ. For every conditional rule $\rho \in R$, we prepare k 'fresh' function symbols $U^\rho_1,\ldots,U^\rho_{|\rho|}$ not in $\mathcal{F}$, called U *symbols*, in the transformation. We transform ρ into

a set $\mathbb{U}(\rho)$ of $k+1$ unconditional rewrite rules as follows:

$$\mathbb{U}(\rho) = \left\{ l \to U_1^\rho(s_1, \overrightarrow{X_1}),\ U_1^\rho(t_1, \overrightarrow{X_1}) \to U_2^\rho(s_2, \overrightarrow{X_2}),\ \cdots,\ U_k^\rho(t_k, \overrightarrow{X_k}) \to r \right\}$$

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

Note that the definition of $\mathbb{U}$ is essentially equivalent to that in [21,23].

An unraveling U is *simulation-sound* (*simulation-preserving* and *simulation-complete*, respectively) for a DCTRS R over $\mathcal{F}$ if the following holds: for all s and $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $s \xrightarrow{*}_R t$ if ('only if' and 'iff', respectively) $s \xrightarrow{*}_{U(R)} t$. Note that simulation-preservingness 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 [22]. To avoid this difficulty of non-'simulation-soundness' of $\mathbb{U}$, a restriction to the rewrite relations of the unraveled TRSs is shown in [23], which is done by the *context-sensitive* condition given by the replacement map μ such that $\mu(U_i^\rho) = \{1\}$ for every U_i^ρ in Definition 3.1. We denote the context-sensitive TRS $(\mathbb{U}(R), \mu)$ by $\mathbb{U}_{\mathrm{cs}}(R)$. We consider $\mathbb{U}_{\mathrm{cs}}$ as an unraveling from CTRSs to context-sensitive TRSs.

Theorem 3.2 ([23]) *For every DCTRS R over $\mathcal{F}$, $\xrightarrow{*}_R = \xrightarrow{*}_{\mathbb{U}_{\mathrm{cs}}(R)}$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$.*

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, called *operationally innermost reduction*. This notion removes the context-sensitivity for simulation-soundness from the corresponding reduction.

For a binary relation $\to$ on terms, the binary relation $\xrightarrow{*}!$ is defined as $\{ (s,t) \mid s \xrightarrow{*} t, t \in NF_\to \}$ where $NF_\to$ is the set of normal forms with respect to $\to$. Let R be an OP-SN DCTRS. The *n-level operationally innermost reduction* $\xrightarrow[(n),\mathrm{i}]{}_R$ is defined as follows: $\xrightarrow[(0),\mathrm{i}]{}_R = \emptyset$, and $\xrightarrow[(n+1),\mathrm{i}]{}_R = \xrightarrow[(n),\mathrm{i}]{}_R \cup \{ (C[l\sigma], C[r\sigma]) \mid l \to r \Leftarrow s_1 \to t_1 \wedge \cdots s_k \to t_k \in R, \mathcal{R}an(\sigma) \subseteq NF_{\xrightarrow[(n),\mathrm{i}]{}_R}, \forall i.\ s_i\sigma \xrightarrow[(n),\mathrm{i}]{*}!_R t_i\sigma \}$. The *operationally innermost reduction* $\xrightarrow[\mathrm{i}]{}_R$ of R is defined as $\bigcup_{i \ge 0} \xrightarrow[(i),\mathrm{i}]{}_R$. Note that if R is a TRS, then the operationally innermost reduction of R is equivalent to the ordinary innermost reduction. Note that the ordinary innermost reduction is not well-defined for every CTRS [6]. However, both the ordinary and operationally innermost reductions of OP-SN CTRSs are well-defined.

For OP-SN DCTRSs, Theorem 3.2 holds for $\xrightarrow[\mathrm{i}]{}_R$ and $\xrightarrow[\mathrm{i}]{}_{\mathbb{U}_{\mathrm{cs}}(R)}$.

Theorem 3.3 *For every OP-SN DCTRS R over $\mathcal{F}$, $\xrightarrow[\mathrm{i}]{*}_R = \xrightarrow[\mathrm{i}]{*}_{\mathbb{U}_{\mathrm{cs}}(R)}$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$.*

The proof of Theorem 3.3 follows the proof of Theorem 3.2 (see the appendix of [17]). The context-sensitive constraint is not necessary for the innermost reduction.

Theorem 3.4 *For every DCTRS R over $\mathcal{F}$, $\xrightarrow[\mathrm{i}]{*}_{\mathbb{U}(R)} = \xrightarrow[\mathrm{i}]{*}_{\mathbb{U}_{\mathrm{cs}}(R)}$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$.*

Proof. It is clear that $\xrightarrow[\mathrm{i}]{*}_{\mathbb{U}(R)} \supseteq \xrightarrow[\mathrm{i}]{*}_{\mathbb{U}_{\mathrm{cs}}(R)}$ in $\mathcal{T}(\mathcal{F},\mathcal{V})$. It follows from the notion of innermost and context-sensitive reductions that for a term reachable from terms in $\mathcal{T}(\mathcal{F},\mathcal{V})$, every term in any *irreducible positions* determined by the replacement map is a normal form. Thus, $\xrightarrow[\mathrm{i}]{}_{\mathbb{U}(R)} \subseteq \xrightarrow[\mathrm{i}]{}_{\mathbb{U}_{\mathrm{cs}}(R)}$ in $\mathcal{T}(\mathcal{F},\mathcal{V})$, and hence $\xrightarrow[\mathrm{i}]{*}_{\mathbb{U}(R)} \subseteq \xrightarrow[\mathrm{i}]{*}_{\mathbb{U}_{\mathrm{cs}}(R)}$ in $\mathcal{T}(\mathcal{F},\mathcal{V})$. □

According to Theorem 3.4, when evaluating terms by innermost reductions of $\mathbb{U}_{\mathrm{cs}}(R)$, we can treat $\mathbb{U}(R)$ without the context-sensitive constraint determined by $\mathbb{U}$.

For OP-SN DCTRSs, we have the following simulation-completeness.

Corollary 3.5 *For every OP-SN DCTRS R over $\mathcal{F}$, $\xrightarrow[\mathrm{i}]{*}_R = \xrightarrow[\mathrm{i}]{*}_{\mathbb{U}(R)}$ in $\mathcal{T}(\mathcal{F},\mathcal{V})$.*

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 Knuth-Bendix completion procedure [10,25]. Since we add an automated post-process into the inversion compiler, we here use the automated procedure instead of the ordinary completion based on inference rules [2].

Definition 3.6 Let E be a finite set of equations over $\mathcal{F}$, and $\succ$ be a reduction order. Let $E_{(0)} = E$, $R_{(0)} = \emptyset$ and $i = 0$, we apply the following steps:

1. (Orientation) select $s \approx t \in E_{(i)}$ such that $s \succ t$;
2. (Composition) $R' := \{ l \to r' \mid l \to r \in R_{(i)}, r \xrightarrow[\mathrm{i}]{*!}_{R_{(i)} \cup \{s \to t\}} r' \}$;
3. (Deduction) $E' := (E_{(i)} \setminus \{s \approx t\}) \cup CP(\{s \to t\}, R' \cup \{s \to t\})$;
4. (Collapse) $R_{(i+1)} := \{s \to t\} \cup \{ l \to r \mid l \to r \in R', l \not\supseteq s \}$;
5. (Simplification & Deletion)
   $E_{(i+1)} := \{ s'' \approx t'' \mid s' \approx t' \in E', s' \xrightarrow[\mathrm{i}]{*!}_{R_{(i+1)}} s'' \not\equiv t'' \xleftarrow[\mathrm{i}]{*!}_{R_{(i+1)}} t' \}$;
6. if $E_{(i+1)} \neq \emptyset$ then $i := i + 1$ and go to step 1.

Note that $l \supsetneq s$ if there are some $C[\,]$ and θ such that $l \equiv C[s\theta]$. 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 R_k satisfies $\overset{*}{\leftrightarrow}_E = \overset{*}{\leftrightarrow}_{R_k}$ [2]. Note that when there is no rule to select at the Orientation step, the procedure halts in failure.

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 transform unraveled TRSs $\mathbb{U}(R)$ into convergent TRSs as executable programs that are computationally equivalent to the original CTRSs R. For this reason, we start the completion procedure from $(CP(\mathbb{U}(R)), \{ l \to r \in \mathbb{U}(R) \mid \nexists l' \to r' \in \mathbb{U}(R), l \supsetneq l' \})$ 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 modified system) is necessary for preserving computational equivalence of R. For this requirement, we add the side condition '$\mathtt{root}(s)$ is a U symbol' to the Orientation step:

1. (Orientation†) *select $s \approx t \in E_{(i)}$ such that $s \succ t$ and $\mathtt{root}(s)$ is a U symbol;*

Due to the side condition of the ORIENTATION step, and the basic character of the completion procedure [2], the completion procedure produces convergent TRSs that are computationally equivalent to the input TRSs.

Theorem 3.7 *Let R be an OP-SN DCTRS over $\mathcal{F}$, and $\succ$ be a reduction order such that $\mathbb{U}(R) \subseteq \succ$. Let $E_0 = CP(\mathbb{U}(R))$, $R_0 = \{l \to r \in \mathbb{U}(R) \mid \nexists l' \to r \in \mathbb{U}(R), l \trianglerighteq l'\}$, and R' be a TRS obtained by the completion procedure from (E_0, R_0) with $\succ$. Then, (1) R' is convergent and (2) $\xrightarrow{*}{}!_{\mathbb{U}(R)} = \xrightarrow{*}{}!_{R'}$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$.*

Proof. It follows from the side condition '$\mathtt{root}(s)$ is a U symbol' of the ORIENTATION that $NF_{\mathbb{U}(R)}(\mathcal{F}, \mathcal{V}) = NF_{R'}(\mathcal{F}, \mathcal{V})$. It also follows from the correctness of the completion (Theorem 7.3.5 in [2]) that R' is convergent and $\overset{*}{\leftrightarrow}_{CP(\mathbb{U}(R))\cup\mathbb{U}(R)} \subseteq \xrightarrow{*}{}_{R'} \cdot \xleftarrow{*}{}_{R'}$. Let $\underset{1}{\to} = \{(s,t) \mid s,t \in \mathcal{T}(\mathcal{F}, \mathcal{V}),\ s \xrightarrow{+}{}!_R t\}$ and $\underset{2}{\to} = \{(s,t) \mid s,t \in \mathcal{T}(\mathcal{F}, \mathcal{V}),\ s \xrightarrow{+}{}!_{R'} t\}$. Then, we have $\underset{1}{\overset{*}{\leftrightarrow}} \subseteq \underset{2}{\overset{*}{\leftrightarrow}}$, confluence of $\underset{2}{\to}$, termination of $\underset{1}{\to}$, and $NF_1 = NF_2$ where NF_i is the set of normal forms with respect to $\to$. Therefore, it follows from Theorem 3.3 in [26] that $\underset{1}{\xrightarrow{*}} = \underset{2}{\xrightarrow{*}}$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$, and hence $\xrightarrow{*}{}!_R = \xrightarrow{*}{}!_{R'}$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. □

Note that (2) implies $NF_{\mathbb{U}(R)}(\mathcal{F}, \mathcal{V}) = NF_{R'}(\mathcal{F}, \mathcal{V})$.

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 ($R_{\mathtt{du}}$ in Subsection 4.2). Since such TRSs are not confluent but innermost-confluent, the completion procedure possibly fails in modifying those TRSs. To solve this problem and to obtain innermost-convergent TRSs instead of convergent TRSs, we show an initial setting of the completion to unraveled TRSs.

Theorem 3.8 *Let R be an OP-SN DCTRS over $\mathcal{F}$, and $\succ$ be a reduction order such that $\mathbb{U}(R) \subseteq \succ$. Let $E_0 = CP_\varepsilon(\mathbb{U}(R))$, $R_0 = \{l \to r \in \mathbb{U}(R) \mid \nexists l' \to r \in \mathbb{U}(R), l \trianglerighteq l'\}$, and R' be a TRS obtained by the completion procedure from (E_0, R_0) with $\succ$. Then, (1) R' is innermost-convergent and (2) $\underset{\mathrm{i}}{\xrightarrow{*}}{}!_{\mathbb{U}(R)} = \underset{\mathrm{i}}{\xrightarrow{*}}{}!_{R'}$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$.*

Proof. We here show the sketch of the proof. The full proof will be found in the appendix of [17]. It follows from the side condition '$\mathtt{root}(s)$ is a U symbol' of the ORIENTATION that $NF_{\mathbb{U}(R)}(\mathcal{F}, \mathcal{V}) = NF_{R'}(\mathcal{F}, \mathcal{V})$. Following the correctness proof of the ordinary completion (Section 7.3 and 7.4 in [2]), it can be shown that $\underset{\mathrm{i}}{\overset{*}{\leftrightarrow}}_{\mathbb{U}(R)} \subseteq \underset{\mathrm{i}}{\xrightarrow{*}}_{R'} \cdot \underset{\mathrm{i}}{\xleftarrow{*}}_{R'}$ and R' is innermost-convergent. The remainder of the proof is similar to the corresponding part of the proof of Theorem 3.7. □

Note that (2) implies $NF_{\mathbb{U}(R)}(\mathcal{F}, \mathcal{V}) = NF_{R'}(\mathcal{F}, \mathcal{V})$.

The following condition is necessary for the completion procedure to halt 'successfully'[4]: for every term $s \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, its normal form over $\mathcal{F}$ with respect to the innermost reduction is unique, that is, for all normal forms t_1 and $t_2 \in NF_{\mathbb{U}(R)}(\mathcal{F}, \mathcal{V})$, if $t_1 \underset{\mathrm{i}}{\overset{*}{\leftarrow}}_{\mathbb{U}(R)} s \underset{\mathrm{i}}{\overset{*}{\rightarrow}}_{\mathbb{U}(R)} t_2$, then $t_1 \equiv t_2$. Note that if R is confluent, then $\mathbb{U}(R)$ satisfies this property. If $t_1 \underset{\mathrm{i}}{\overset{*}{\leftarrow}}_{\mathbb{U}(R)} s \underset{\mathrm{i}}{\overset{*}{\rightarrow}}_{\mathbb{U}(R)} t_2$ and $t_1 \not\equiv t_2$, then the added side condition '$\mathtt{root}(s)$ is a U symbol' prevents t_1 and t_2 from being joinable.

[4] Notice that this condition is not sufficient for the procedure to halt; In other words, the procedure halts (or keeps running) 'unsuccessfully' if the input system does not satisfy this condition.

Example 3.9 Consider the non-convergent TRS $\mathbb{U}(\mathcal{I}nv(R_1))$ in Section 1 again. Given the *lexicographic path order* (LPO) $\succ_{\mathsf{lpo}}$ determined by the precedence $>$ with $\mathsf{InvSnoc} > U_1 > \mathsf{cons} > \mathsf{nil} > \langle\ \rangle$, we obtain the following convergent and non-overlapping TRS by the completion procedure (in 4 cycles):

$$R_2 = \left\{ \begin{array}{ll} \mathsf{InvSnoc}((x::ys)) \to U_1(\mathsf{InvSnoc}(ys), x, ys), & \\ U_1(\langle xs, y\rangle, x, ys) \to \langle (x::xs), y\rangle, & U_1(\mathsf{InvSnoc}(\mathsf{nil}), x, \mathsf{nil}) \to (\mathsf{nil}, x) \end{array} \right\}$$

The completion removes the rule $\mathsf{InvSnoc}(\mathsf{cons}(y, \mathsf{nil})) \to \langle \mathsf{nil}, y\rangle$ from $\mathbb{U}(\mathcal{I}nv(R_1))$ and the resultant TRS R_2 are non-overlapping. This removal is effective in translating the TRS into a functional program.

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 [22] although there exists an appropriate convergent TRS which is equivalent to the unraveled TRS.

3.4 Translation Back into Functional Programs

In this subsection, we informally discuss translations from convergent and non-overlapping TRS R_2 into functional programs in Standard ML. The translations have not been automated yet but we believe that the automation is neither so difficult nor so surprising. It is difficult to translate $\mathcal{I}nv(R_1)$ or $\mathbb{U}(\mathcal{I}nv(R_1))$ 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 $\mathcal{I}nv(R_1)$ because R_2 is not only confluent but also non-overlapping.

The U symbols U_i^ρ 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:

```
case InvSnoc( ys ) of (xs,y) => ( x::xs, y )
             | InvSnoc( [] ) => ( [], y )
```

where `InvSnoc( [] )` 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:

```
case ys of [] => ( [], y )
         | _  => (case InvSnoc( ys ) of (xs,y) => ( x::xs, y ) )
```

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

```
fun InvSnoc( (x::ys) ) =
    case ys of [] => ( [], x )
             | _ => (case InvSnoc(ys) of (xs,y) => (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 InvSnoc.

3.5 Completion with Termination Provers

In this subsection, we show an example where the completion consults with termination provers. This idea is firstly introduced in [28].

Consider the following unraveled TRS:

$$R_3 = \mathbb{U}(\mathcal{I}nv(R_1)) \cup \left\{ \begin{array}{l} \mathsf{InvSnocRev}(\mathsf{nil}) \to \langle \mathsf{nil} \rangle, \\ \mathsf{InvSnocRev}(y) \to \mathsf{U}_2(\mathsf{InvSnoc}(y), y), \\ \mathsf{U}_2(\langle z, x \rangle, y) \to \mathsf{U}_3(\mathsf{InvSnocRev}(z), x, y, z), \\ \mathsf{U}_3(\langle x_1 \rangle, x, y, z) \to \langle \mathsf{cons}(x, x_1) \rangle \end{array} \right\}$$

In contrast to the case of $\mathsf{InvSnoc}$, there is no LPO $\succ_{\mathsf{lpo}}$ with $R_3 \subseteq \succ_{\mathsf{lpo}}$. Detecting such a path-based reduction order (e.g., LPO and recursive path order) 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. We modify ORIENTATION, following the approach shown in [28]:

1. (ORIENTATION‡) *select* $s \approx t \in E_{(i)}$ *such that* $\bigcup_{j=0}^{i} R_{(j)} \cup \{s \to t\}$ *is terminating, and* $\mathtt{root}(s)$ *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 transformed by the procedure (in 2 cycles) into the following convergent and non-overlapping TRS:

$$R_2 \cup \left\{ \begin{array}{l} \mathsf{InvSnocRev}(v) \to \mathsf{U}_2(\mathsf{InvSnoc}(v), v), \\ \mathsf{U}_2(\langle w, x \rangle, v) \to \mathsf{U}_3(\mathsf{U}_2(\mathsf{InvSnoc}(w), w), v, w, x), \\ \mathsf{U}_3(\langle xs \rangle, v, w, x) \to \langle \mathsf{cons}(x, xs) \rangle, \quad \mathsf{U}_3(\mathsf{InvSnoc}(\mathsf{nil}), x, \mathsf{nil}) \to \langle \mathsf{nil}, x \rangle \end{array} \right\}$$

4 Completion as Post-Process in Program Inversion

In this section, we apply the unraveling $\mathbb{U}$ and the completion procedure to CTRSs generated by the partial inversion compiler [20], that is, we apply the completion as a *post-process* of $\mathbb{U}(\mathcal{I}nv(\cdot))$ to the unraveled TRSs. 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 $\mathcal{I}nv$ in [20] 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 $\mathsf{Inv}F$ introduced by $\mathcal{I}nv$ 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

We first define *injectivity* of TRSs [18], and then give sufficient condition for input constructor TRSs whose inverse CTRSs generated by $\mathcal{I}nv$ are convergent.

Definition 4.1 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

$\{(\langle s_1,\ldots,s_n\rangle,t) \mid s_1,\ldots,s_n,t \in NF_R(\mathcal{F},\mathcal{V}), F(s_1,\ldots,s_n) \xrightarrow{*}_R t\}$ 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 [18].

The following defined symbol Reverse computes the reverses of given lists:

$$R_4 = \left\{ \begin{array}{ll} \mathsf{Reverse}(xs) \to \mathsf{Rev}(xs,\mathsf{nil}), & \mathsf{Rev}(\mathsf{nil},ys) \to ys, \\ \mathsf{Rev}(\mathsf{cons}(x,xs),ys) \to \mathsf{Rev}(xs,\mathsf{cons}(x,ys)) & \end{array} \right\}$$

The inverse TRS of the above TRS is generated as follows:

$$\mathbb{U}(\mathcal{I}nv(R_4)) = \{\cdots,\ \mathsf{InvRev}(z) \to U_4(\mathsf{InvRev}(z),z),\ \cdots\}.$$

Reverse is injective but Rev is not. Thus, R_4 is not injective. In this case, the TRS $\mathbb{U}(\mathcal{I}nv(R_4))$ is not terminating. For this reason, we restrict ourselves to injective functions whose inverse TRSs are terminating. In [18], a sufficient condition has been shown for the full inversion compiler in [19] to generate convergent inverse CTRSs from injective TRSs. The condition is also effective for the partial inversion compiler $\mathcal{I}nv$ [20].

Theorem 4.2 *Let R be a non-erasing innermost-convergent constructor TRS. If $F \in \mathcal{D}_R$ is injective, then for all s, t_1 and $t_2 \in NF_{\mathcal{I}nv(R)}(\mathcal{F},\mathcal{V})$, $t_1 \xleftarrow[\mathrm{i}]{*}_{\mathbb{U}(\mathcal{I}nv(R))} \mathsf{Inv}F(s) \xrightarrow[\mathrm{i}]{*}_{\mathbb{U}(\mathcal{I}nv(R))} t_2$ implies $t_1 \equiv t_2$. Suppose that for every rule $F(u_1,\ldots,u_n) \to r$ in R, if r is not a variable, then the root symbol of r does not depend [5] on F. Then, the CTRS $\mathcal{I}nv(R)$ is OP-SN, and the TRS $\mathbb{U}(\mathcal{I}nv(R))$ is terminating.*

The proof of Theorem 4.2 follows the proof of Theorem 4 in [18] (see the appendix of [17]). Note that $\mathbb{U}(\mathcal{I}nv(R))$ is not always confluent even if $\mathcal{I}nv(R)$ is confluent. The first claim in Theorem 4.2 shows that given an injective TRS R, $\mathbb{U}(\mathcal{I}nv(R))$ satisfies the necessary condition (described above Example 3.9) for successful run of the completion. When a constructor TRS R does not satisfy the condition in Theorem 4.2 for preserving termination, we directly check the (innermost-)termination of $\mathbb{U}(\mathcal{I}nv(R))$. In other words, when R satisfies the second assumption in Theorem 4.2, we are free of the termination check of $\mathbb{U}(\mathcal{I}nv(R))$ that is less efficient than the check of satisfying the second assumption.

4.2 Experiments

In this section, we report the results of applying an implementation of our approach based on Theorem 3.8 and the ORIENTATION[§] step to several samples.

The implementation of the completion procedure is based on the ML programs shown in [2]. In our implementation, we use the following weight w for equations: $w(s \simeq t) = (size(s) + depth(s)) \times 2 + (size(t) + depth(t))$ where $s \succ t$, and $size(u)$ and $depth(u)$ 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 $s \approx t$

[5] An n-ary symbol G of R *depends on a symbol* F if (G,F) is in the transitive closure of the relation $\{(G',F') \mid G'(\cdots) \to C[F'(\cdots)] \in R\}$.

such that $(\bigcup_{j=0}^{i} R_{(i)}) \cup \{s \to t\}$ 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 [4] 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 in advance. The timeout for checking termination is 300 seconds in every call of the prover.

In [8], the results of the experiments for the inversion compiler LRinv [8,9] running on 15 samples [6] 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 $\mathbb{U}(\mathcal{I}nv(R_1))$, R_3 and $\mathbb{U}(\mathcal{I}nv(R_4))$, 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 $\mathcal{I}nv$ from them are not OP-SN. We excluded those non-'OP-SN' examples from experiments.

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

$$R_{\mathsf{du}} = \left\{ \begin{array}{lll} \mathsf{Du}(\langle x \rangle) \to \langle x, x \rangle, & \mathsf{Du}(\langle x, y \rangle) \to \mathsf{EqChk}(\mathsf{EQ}(x, y)), & \\ \mathsf{EqChk}(\langle x \rangle) \to \langle x \rangle, & \mathsf{EqChk}(\mathsf{EQ}(x, y)) \to \langle x, y \rangle, & \mathsf{EQ}(x, x) \to \langle x \rangle \end{array} \right\}$$

Since R_{du} has no *overlay*, R_{du} is *locally innermost-confluent*, and hence, R_{du} is *innermost-confluent* [11]. Under the innermost reduction, the TRS can simulate computation of `du`. The operator `du` is an inverse of itself [8,9]. 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.

Table 1 summarizes the results of the experiments for our approach running on 10 of the 15 examples previously mentioned, which are easily translated to TRSs [7]. The second column labeled with 'CR by [1]' shows whether the input TRS of the example is in the class shown in [1], in which the corresponding inverse TRS is orthogonal and thus confluent. In that case, the implemented procedure only checks innermost-termination of the inverse TRS. The third column labeled with 'SN by Th. 4.2' shows whether the input TRS satisfies the conditions in Theorem 4.2, that is, the corresponding inverse TRSs are terminating. The fourth column shows the results of completion ('success $\surd$', 'fail' or 'timeout') with the numbers of running 'cycles' in the sense of Definition 3.6. The numbers of cycles are equivalent to times of applying ORIENTATION. As described above, the implementation checks the innermost termination of input TRSs before the completion procedure starts. Thus, we have the results 'success (0 cycle) and 1 call of provers'. The sixth column

[6] Unfortunately, the site shown in [8] is not accessible now. The examples are also described briefly as functional programs in [9], and some of the detailed programs can be found in [9].

[7] The detail of the experiments will be available from the following URL:
`http://www.trs.cm.is.nagoya-u.ac.jp/repius/experiments/`

Table 1
the results of the experiments

example	CR by [1]	SN by Th.4.2	innermost CR&SN by completion			
			result (cycles)	call AProVE	time	¬overlap
`du` (primitive)			√ (0 cycle)	1 time	0.64 s	
`snoc.fct`		√	√ (1 cycle)	2 times	2.12 s	√
`snocrev.fct`		√	√ (2 cycles)	3 times	4.61 s	√
`double.fct`			√ (1 cycle)	2 times	2.32 s	
`mirror.fct`			√ (2 cycles)	3 times	4.10 s	
`zip.fct`	√	√	√ (0 cycle)	1 time	1.04 s	√
`inc.fct`		√	√ (1 cycle)	2 times	2.53 s	√
`octbin.fct`	√	√	√ (0 cycle)	1 time	1.28 s	√
`treelist.fct`		√	timeout (0 cycle)	timeout at 1st time	timeout	—
`print-sexp.fct`			√ (6 cycles)	7 times	35.53 s	√
`print-xml.fct`			√ (2 cycles)	3 times	14.02 s	
`treelist.fct`†		√	√ (4 cycles)	5 times	40.92 s	

†The improved transformation [16] of $\mathbb{U}$ is applied instead of $\mathbb{U}$.

shows the average time of 5 trials. The rightmost column shows whether or not the resultant TRSs are non-overlapping (*surd* means the resultant is non-overlapping). None of the resultant TRSs has overlay while part of them are overlapping.

In the experiments, the procedure failed in modifying `treelist.fct` because of a timeout in pre-checking the innermost termination [8]. For the intermediate CTRS R_5 generated from `treelist.fct`, the improvement of $\mathbb{U}$ shown in [16] is effective. Note that the improvement proposed for the variant $\mathbb{U}'$ of $\mathbb{U}$ is also applicable to $\mathbb{U}$ where $\mathbb{U}'$ shown in [3,20,19] is obtained by setting $X_i = \mathcal{V}ar(l, t_1, \ldots, t_{i-1}) \cap \mathcal{V}ar(r, t_i, s_{i+1}, t_{i+1}, \ldots, s_k, t_k)$ in Definition 3.1. $\mathbb{U}$ unravels one of the conditional rules into 5 rules but the improved transformation $\mathbb{U}'$ of $\mathbb{U}$ unravels the rule into 4. Surprisingly, the completion succeeds in modifying the TRS $\mathbb{U}'(R_5)$ (see the result on the bottom line of Table 1). Remark that in other examples, there is no difference between applications of $\mathbb{U}$ and $\mathbb{U}'$ to the inverse CTRSs.

We tried to prove by TTT [7] innermost-termination of $\mathbb{U}(R_5)$ and $\mathbb{U}'(R_5)$. The results are the same with AProVE 1.2: 'timeout' and 'success', respectively. Moreover, AProVE 1.2 succeeded in proving termination of both $\mathbb{U}(R_5)$ and $\mathbb{U}'(R_5)$, and TTT did not in either of those TRSs.

5 Comparison with Related Work

Completion procedures are used for solving word problems, for transforming equations to 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 method in this paper does not always succeed for every confluent and OP-SN DCTRSs while the latest transformation [24] based on Viry's approach [27] always succeeds. Consider the example in Section 1 again. To eliminate the unexpected normal form $U_1(U_1(U_1(\mathsf{InvSnoc}(\mathsf{nil}), \mathsf{c}, \mathsf{nil}), \mathsf{b}, [\mathsf{c}]), \mathsf{a}, [\mathsf{b}, \mathsf{c}])$ from the set of normal forms, the transformation in [24] is effective in the sense

[8] The current 'web interface' version of AProVE succeeds in proving this innermost termination.

of producing convergent systems. By the transformation, we obtain the following convergent TRS instead of $\mathbb{U}(\mathcal{I}nv(R_1))$:

$$\left\{\begin{array}{rl} \mathsf{InvSnoc}(\mathsf{cons}(y,\mathsf{nil}),z) & \to \{\langle \mathsf{nil}, y\rangle\}, \\ \mathsf{InvSnoc}(\mathsf{cons}(x,ys),\bot) & \to \mathsf{InvSnoc}(\mathsf{cons}(x,ys),\{\mathsf{InvSnoc}(ys,\bot)\}), \\ \mathsf{InvSnoc}(\mathsf{cons}(x,ys),\{\langle xs,y\rangle\}) & \to \{\langle \mathsf{cons}(x,xs),y\rangle\}, \\ \mathsf{InvSnoc}(\{xs\},z) & \to \{\mathsf{InvSnoc}(xs,\bot)\}, \quad \{\{x\}\} \to \{x\} \end{array}\right\}$$

$$\cup \{\, c(x_1,\ldots,\{x_i\},\ldots,x_n) \to \{c(x_1,\ldots,x_n)\} \mid c \text{ is a constructor, } n \geq 1 \,\}$$

where $\{\ \}$ and $\bot$ are special function symbols not in the original signature. In this system, the term $\mathsf{InvSnoc}([\mathsf{a},\mathsf{b},\mathsf{c}],\bot)$ has a unique normal form $\{\langle[\mathsf{a},\mathsf{b}],\mathsf{c}\rangle\}$. However, it is difficult to translate the convergent but *overlapping* TRS into a functional program because the system contains special symbols $\{\ \}$ and $\bot$ and *overlapping* rules.

On the other hand, the modified completion in this paper unexpectedly succeeded for all examples on program inversion we tried, except for functions that call non-injective functions such as ones including accumulators. The resultant systems of our method are not always non-overlapping (equivalently non-overlay for innermost reduction), while the results of [24] are always overlapping. One of our future works is to find a subclass in which the completion halts successfully, and then to compare this framework with the transformation [24] based on Viry's approach [27].

The inversion compiler LRinv, the closest one to the method in this paper, has been proposed for injective functions written in a functional language [8,5,9]. 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 scripts `reverse.fct`, `unbin.fct`, `treepath.fct`, `pack.fct` and `pack-bin.fct` while the method in this paper is not applicable to those scripts. From this fact, LRinv seems to be stronger than the method in this paper but there must be a plenty of room on improving the principle of inversion used in the partial inversion complier in [20]. As future work, we plan to extend the partial inversion compiler for functions with accumulators such as Rev.

Acknowledgement

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

References

[1] Almendros-Jiménez, J. M. and G. Vidal, *Automatic partial inversion of inductively sequential functions*, in: *Proc. of the 18th International Symposium on Implementation and Application of Functional*

Languages, Lecture Notes in Computer Science **4449** (2006), pp. 253–270.

[2] Baader, F. and T. Nipkow, "Term Rewriting and All That," Cambridge University Press, United Kingdom, 1998.

[3] Durán, F., S. Lucas, J. Meseguer, C. Marché and X. Urbain, *Proving termination of membership equational programs*, in: *Proc. of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation* (2004), pp. 147–158.

[4] Giesl, J., P. Schneider-Kamp and R. Thiemann, *Automatic termination proofs in the dependency pair framework*, in: *Proc. of the 3rd International Joint Conference on Automated Reasoning*, Lecture Notes in Computer Science **4130** (2006), pp. 281–286.

[5] Glück, R. and M. Kawabe, *A method for automatic program inversion based on LR(0) parsing*, Fundam. Inform. **66** (2005), no. 4, pp. 367–395.

[6] Gramlich, B., *On the (non-)existence of least fixed points in conditional equational logic and conditional rewriting*, in: *Proc. 2nd Int. Workshop on Fixed Points in Computer Science – Extended Abstracts* (2000), pp. 38–40.

[7] Hirokawa, N. and A. Middeldorp, *Tyrolean termination tool: Techniques and features*, Information and Computation **205** (2007), no. 4, pp. 474–511.

[8] Kawabe, M. and Y. Futamura, *Case studies with an automatic program inversion system*, in: *Proc. of the 21st Conference of Japan Society for Software Science and Technology*, 6C-3, 2004, pp. 1–5.

[9] Kawabe, M. and R. Glück, *The program inverter LRinv and its structure*, in: *Proc. of the 7th International Symposium on Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science **3350** (2005), pp. 219–234.

[10] Knuth, D. E. and P. B. Bendix, *Simple word problems in universal algebra*, in: *Computational Problems in Abstract Algebra* (1970), pp. 263–297.

[11] Krishna Rao, M. R. K., *Relating confluence, innermost-confluence and outermost-confluence properties of term rewriting systems*, Acta Informatica **33** (1996), no. 6, pp. 595–606.

[12] Lucas, S., *Context-sensitive computations in functional and functional logic programs*, Journal of Functional and Logic Programming **1998** (1998), no. 1.

[13] Lucas, S., C. Marché and J. Meseguer, *Operational termination of conditional term rewriting systems*, Information Processing Letters **95** (2005), no. 4, pp. 446–453.

[14] Marchiori, M., *Unravelings and ultra-properties*, in: *Proc. of the 5th International Conference on Algebraic and Logic Programming*, Lecture Notes in Computer Science **1139** (1996), pp. 107–121.

[15] Marchiori, M., *On deterministic conditional rewriting*, Computation Structures Group, Memo 405, MIT Laboratory for Computer Science (1997).

[16] Nishida, N., T. Mizutani and M. Sakai, *Transformation for refining unraveled conditional term rewriting systems*, in: *Proc. of the Sixth International Workshop on Reduction Strategies in Rewriting and Programming*, Electronic Notes in Theoretical Computer Science **174** (2007), Issue 10, pp. 75–95.

[17] 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/` (2008), the full version of this paper.

[18] Nishida, N., M. Sakai and T. Kato, *Convergent term rewriting systems for inverse computation of injective functions*, in: *Proc. of the 9th International Workshop on Termination* (2007), pp. 77–81.

[19] Nishida, N., M. Sakai and T. Sakabe, *Generation of inverse computation programs of constructor term rewriting systems*, The IEICE Trans. Inf.& Syst. **J88-D-I** (2005), no. 8, pp. 1171–1183 (in Japanese).

[20] Nishida, N., M. Sakai and T. Sakabe, *Partial inversion of constructor term rewriting systems*, in: *Proc. of the 16th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science **3467** (2005), pp. 264–278.

[21] Ohlebusch, E., *Termination of logic programs: Transformational methods revisited*, Applicable Algebra in Engineering, Communication and Computing **12** (2001), no. 1-2, pp. 73–116.

[22] Ohlebusch, E., "Advanced Topics in Term Rewriting," Springer-Verlag, 2002.

[23] Schernhammer, F. and B. Gramlich, *On proving and characterizing operational termination of deterministic conditional rewrite systems*, in: *Proc. of the 9th International Workshop on Termination* (2007), pp. 82–85.

[24] Serbanuta, T.-F. and G. Rosu, *Computationally equivalent elimination of conditions*, in: *Proc. of the 17th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science **4098** (2006), pp. 19–34.

[25] Terese, "Term Rewriting Systems," Cambridge University Press, 2003.

[26] Toyama, Y., *How to prove equivalence of term rewriting systems without induction*, Theoretical Computer Science **90** (1991), no. 2, pp. 369–390.

[27] Viry, P., *Elimination of conditions*, Journal of Symbolic Computation **28** (1999), no. 3, pp. 381–401.

[28] Wehrman, I., A. Stump and E. M. Westbrook, *Slothrop: Knuth-Bendix completion with a modern termination checker*, in: *Proc. of the 17th International Conference on Term Rewriting and Applications*, Lecture Notes in Computer Science **4098** (2006), pp. 287–296.