Static Dependency Pair Method in Rewriting Systems for Functional Programs with Product, Algebraic Data, and ML-Polymorphic Types

Keiichirou KUSAKARI(a), Member

SUMMARY For simply-typed term rewriting systems (STRSs) and higher-order rewrite systems (HRSs) à la Nipkow, we proposed a method for proving termination, namely the static dependency pair method. The method combines the dependency pair method introduced for first-order rewrite systems with the notion of strong computability introduced for typed λ-calculi. This method analyzes a static recursive structure based on definition dependency. By solving suitable constraints generated by the analysis, we can prove termination. In this paper, we extend the method to rewriting systems for functional programs (RFPs) with product, algebraic data, and ML-polymorphic types. Although the type system in STRSs contains only product and simple types and the type system in HRSs contains only product and simple types, our RFPs allow product types, type constructors (algebraic data types), and type variables (ML-polymorphic types). Hence, our RFPs are more representative of existing functional programs than STRSs and HRSs. Therefore, our result makes a large contribution to applying theoretical rewriting techniques to actual problems, that is, to proving the termination of existing functional programs.

key words: rewriting systems for functional programs, termination, static dependency pair method

1. Introduction

Various extensions of term rewriting systems (TRSs) [24] for handling higher-order functions have been proposed [10], [12], [14], [19], [20]. Simply-typed term rewriting systems (STRSs) introduced by Kusakari [14], and higher-order rewrite systems (HRSs) introduced by Nipkow [19] are two such extensions. In this paper, we introduce rewriting systems for functional programs (RFPs), which is an extension of TRSs with product, algebraic data, and ML-polymorphic types. For example, the typical higher-order function foldl can be represented by the following RFP $R_{\text{foldl}}$:

$$
\text{foldl}: \alpha \times \beta \rightarrow \alpha \rightarrow \text{list} \beta \rightarrow \alpha
$$

Here we suppose that the function foldl has the type:

$$
\text{foldl} : (\alpha \times \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \text{list}(\beta) \rightarrow \alpha
$$

in which $\alpha$ and $\beta$ are type variables, and list is a type constructor.

The static dependency pair method is a powerful method to prove termination, which was introduced on STRSs [16], [17], and extended to HRSs [18], [22]. The method combines the dependency pair method introduced for first-order rewrite systems [1] with the notion of strong computability introduced for typed λ-calculi [7], [23]. The static dependency pair method consists in showing the non-loopingness of each static recursion component independently, the set of static recursion components being computed through some static recursion analysis. For the RFP $R_{\text{foldl}}$, the static dependency pair method yields a single static recursion component:

$$
\text{foldl}^2 f e (\text{cons} (x, xs)) \rightarrow \text{foldl}^2 f (f (e, x)) xs
$$

To prove the non-loopingness of static recursion components, the notions of subterm criterion and reduction pair have been proposed. The subterm criterion was introduced on TRSs [9], and slightly improved by extending the subterms permitted by the criterion on STRSs [16], and extended on HRSs [18]. Reduction pairs [15] are an abstraction of weak-reduction order [1]. By using the subterm criterion, we can prove the non-loopingness of the above static recursion component from the following fact:

$$
\text{cons}(x, xs) \triangleright_{\text{sub}} xs \quad (xs \text{ is a subterm of } \text{cons} (x, xs))
$$

By recapitulating such a termination proof by the static dependency pair method, we obtain the following claim:

The function foldl is explicitly recursively defined on the third argument. Hence, the function foldl is well-defined (terminating).

This claim is an assertion of the static dependency pair method, and it may be very natural reasoning. However, it is quite difficult to verify the claim because its reduction may be affected by unanticipated behaviors of functions held in higher-order variables. Actually, the static dependency pair method is not applicable to every system. Let’s consider the additional rule for foo:

$$
\text{foo} : \alpha \times \alpha \rightarrow \alpha
$$

and let $R$ be the following RFP:

$$
R_{\text{foldl}} \cup \{ \text{foo} (x, y) \rightarrow \text{foldl} \text{ foo } y (\text{cons} (x, nil)) \}
$$

Then the RFP $R$ is not terminating because there exists the loop:

$$
\text{foo} (0, 0) \rightarrow \text{foldl} \text{ foo } 0 (\text{cons} (0, nil)) \rightarrow \text{foldl} \text{ foo } (\text{foo} (0, 0)) \rightarrow \text{foo} (0, 0).
$$

As seen above, for the non-termination of $R$, the infinite sequence through
the “second” argument of foldl is essential, but not the “third” argument. This example indicates that such a claim does not hold in general. As a class in which the static dependency pair method is sound, we founded the class of plain function-passing [16], and extended this class to the class of safe function-passing [17].

In this paper, we extend the static dependency pair method and the class of safe function-passing to RFPs, in which we can use arbitrary type constructors (algebraic data types) and type variables (ML-polymorphic types). Then we show the soundness of the static dependency pair method in the class. Since our RFPs are more representative of existing functional programs than STRSs and HRSs, and the class of safe function-passing is sufficiently expressive, our result is very practicable.

The most basic notion in the static dependency pair method is that of the static dependency pair itself. From a theoretical viewpoint, we may extend the static dependency pair method onto polymorphic settings by interpreting the static dependency pair as infinite ones in simple-type settings. However this approach erases practicality of the static dependency pair method. Hence we give polymorphism to the static dependency pair. In order to keep the soundness of the static dependency pair method. Hence we give polymorphism to the static dependency pair. In order to keep the soundness of the static dependency pair method. Hence we give polymorphism to the static dependency pair. In order to keep the soundness of the static dependency pair method. Hence we give polymorphism to the static dependency pair method. Hence we give polymorphism to the static dependency pair method.

As an example showing the effectiveness of the static dependency pair method, there exists polymorphic-typed combinatory logic, which is represented as the following RFP

$$R_{CL} = \begin{cases} S f g x & \rightarrow f x (g x) \\ K x y & \rightarrow x \end{cases}$$

The static dependency pair method can prove its termination from the following two easily checked reasons:

- Each rule is not explicitly recursively defined, that is, S and K do not occur on the right-hand sides.
- Any variable occurs in an argument position on the left-hand sides.

Although several proofs of the termination of polymorphic-typed combinatory logic are known [8], we believe that our proof is very elegant.

The remainder of this paper is organized as follows. The next section provides rewriting systems for functional programs (RFPs) with product, algebraic data, and ML-polymorphic types. In Sect. 3, we provide the notion of strong computability, which gives a theoretical basis for the static dependency pair method. We also give the class of safe function-passing in which the static dependency pair is sound. In Sect. 4, we give the static dependency pair method on RFPs. In Sect. 5, we give the notion of the subterm criterion and reduction pairs that prove the non-loopingness of the static recursion component. Concluding remarks are presented in Sect. 6.

2. Rewriting Systems for Functional Programs

In this section, we introduce rewriting systems for functional programs (RFPs) with product, algebraic data, and ML-polymorphic types. Intuitively, algebraic data types allow type constructors, and ML-polymorphic types allow type variables. RFPs are extensions of term rewriting systems.

The set $S$ of product, ML-polymorphic and algebraic data types (types for short) is generated from the set $TV$ of type variables by the type constructors $(\rightarrow, \times) \cup TC$, in which each symbol $c \in TC$ is associated with a natural number $n$, denoted by $arity(c) = n$. Formally, the set $S$ is defined as the least set satisfying the following properties:

- If $a \in TV$ then $a \in S$.
- If $\sigma_1, \sigma_2 \in S$ then $(\sigma_1 \rightarrow \sigma_2) \in S$.
- If $\sigma_1, \ldots, \sigma_n \in S$ then $(\sigma_1 \times \cdots \times \sigma_n) \in S$.
- If $\sigma_1, \ldots, \sigma_n \in S$ and $c \in TC$ with $arity(c) = n$ then $c(\sigma_1, \ldots, \sigma_n) \in S$.

A functional type or higher-order type is a type of the form $(\sigma_1 \rightarrow \sigma_2)$. A product type is a type of the form $(\sigma_1 \times \cdots \times \sigma_n)$ for $n \geq 2$. A data type is either a product type or a type of the form $c(\sigma_1, \ldots, \sigma_n)$. We denote by $S_{fun}$ the set of non-functional types. To minimize the number of parentheses, we assume that $\rightarrow$ is right-associative and $\times$ has lower precedence than $\rightarrow$. We shortly denote $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma_0$ by $\overline{\sigma_0}$. Under these conventions, any type $\sigma$ is uniquely denoted by the form $\overline{\sigma_0} \rightarrow \sigma_0$ with $\sigma_0 \in S_{fun}$, which we call the canonical form. A type $\sigma$ is said to be closed if no type variable occurs in $\sigma$. A type $\sigma$ is said to be an instance of a type $\sigma'$, denoted by $\sigma' \geq \sigma$, if there is a type substitution $\xi$ such that $\sigma = \xi(\sigma')$.

The set $T_{raw}$ of raw terms generated from the set $F$ of function symbols and the set $V$ of variables without name collision is the smallest set such that $(a \ t_1 \cdots \ t_n)$, $(t_1,\ldots,t_n) \in T_{raw}$ whenever $a \in V \cup F$ and $t_1,\ldots,t_n \in T_{raw}$.

A type environment is a pair $(\Sigma, \Gamma)$ of mappings $\Sigma : F \rightarrow S$ and $\Gamma : V \rightarrow S$. Under an environment $(\Sigma, \Gamma)$,

- if $\Sigma(x) = \overline{\sigma_0} \rightarrow \sigma_0$ then $(x \ t_1^1 \cdots t_n^i)^{\overline{\sigma_0}}$ is a typed term,
- if $\Sigma(f) \geq \overline{\sigma_0} \rightarrow \sigma_0$ then $(f \ t_1^1 \cdots t_n^i)^{\overline{\sigma_0}}$ is a typed term,
- $(t_1^{\overline{\sigma_1}} \cdots t_n^{\overline{\sigma_n}})^{\overline{\sigma_0}}$ is a typed term,

whenever $t_1^1, \ldots, t_n^i$ are typed terms. The identity of typed terms is denoted by $\equiv$. We shortly denote $(a)^\overline{\sigma}$ by $a^\overline{\sigma}$ for $a \in F \cup V$. For a term $t^\overline{\sigma} \equiv (t_1^{\overline{\sigma_1}} \cdots t_n^{\overline{\sigma_n}})^{\overline{\sigma_0}}$, we identify $t^\overline{\sigma} \equiv (t_1^{\overline{\sigma_1}})^{\overline{\sigma_0}} \equiv t_1^i$ if $n = 1$, and $t^\overline{\sigma} \equiv (\unit)^{\overline{\sigma_0}}$ if $n = 0$, where $\unit$ is the special type constructor with $arity(\unit) = 0$. No confusion arises about type environments for the discussions in this paper, because the current version of our
rewriting systems does not allow functional abstraction (λ-abstraction) and let-expressions. Hence we omit a type environment for typed terms, and shortly denote by \( \mathcal{T} \) the set of typed terms (terms for short). We often denote \( t^\sigma \) by \( t : \sigma \), or shortly \( t \) whenever no confusion arises. We abbreviate \((a_1 \ldots a_n)^\sigma\) by \((a_1 \ldots a_n)_{\sigma}^\gamma\), or shortly \( a_n^\gamma\). We often write \( u \) for \((a_1 \ldots a_n)^{\gamma_1 \ldots \gamma_n}\) where \( \gamma = (\gamma_1, \ldots, \gamma_n) \).

**Example 2.1:** Let \( \Sigma(\text{map}) = (\alpha \to \beta) \to \text{list}(\alpha) \to \text{list}(\beta) \). Then we have

\[
\text{map}(\alpha \to \beta) \to \text{list}(\alpha) \to \text{list}(\beta) \in \mathcal{T}
\]

because of \( \Sigma(\text{map}) \geq (\alpha \to \beta) \to \text{list}(\alpha) \to \text{list}(\beta) \). Hence we have

\[
(\text{map map}(\alpha \to \beta) \to \text{list}(\alpha) \to \text{list}(\beta)) \in \mathcal{T}
\]

because of \( \Sigma(\text{map}) \geq \sigma = (\alpha \to \beta) \to \text{list}(\alpha) \to \text{list}(\beta) \to \text{list}(\alpha) \to \text{list}(\beta) \).

The set of positions of a term \( t \) is the set \( \text{Pos}(t) \) of strings over positive integers, which is inductively defined as follows:

\( \text{Pos}(a_n^\gamma) = \{0\} \cup \bigcup_{\gamma} \{p | p \in \text{Pos}(t_i)\} \). The prefix order \( \prec \) on positions is defined by \( p \prec q \) if \( pw = q \) for some \( w \neq e \). The position \( e \) is said to be the root, and a position \( p \) such that \( p \in \text{Pos}(t) \) and \( \exists t \in \text{Pos}(t) \) is said to be a leaf.

The subterm at position \( p \) in \( t \), denoted by \( t_p \), is defined as \( t_p \equiv e \), (a \( t_n^\gamma \) is defined as \( \{p \mid t_{p}^\gamma \in \text{Pos}(t_n) \}) \), and \( \{p \mid t_p \} \in [t] \).

The symbol \( \sigma \) at position \( p \) in \( t \), denoted by \( (a \sigma)^{\gamma_1 \ldots \gamma_n} \equiv (a_1 \ldots a_n \gamma_1 \ldots \gamma_n) \), (a \( t_p^\gamma \) is defined as \( \{p \mid t_{p}^\gamma \in \text{Pos}(t_n) \}) \), and \( \{p \mid t_p \} \in [t] \).

Here it is defined as \( a_n^\gamma \), and \( \{p \mid t_p \} \in [t] \).

**Example 2.2:** Let \( R_{\text{list}} \) be the following RFP:

\[
R_{\text{list}} = \begin{cases} 
\text{hd} \ (\text{cons}(x, xs)) &\rightarrow x \\
\text{tl} \ (\text{cons}(x, xs)) &\rightarrow xs
\end{cases}
\]

Here we suppose that \( \text{nil} : \text{list}(\alpha) \), \( \text{cons} : \alpha \times \text{list}(\alpha) \to \text{list}(\alpha) \), \( \text{hd} : \text{list}(\alpha) \to \alpha \), and \( \text{tl} : \text{list}(\alpha) \to \text{list}(\alpha) \).

Then \( \text{Act}((\text{hd} \ (\text{cons}(x, xs)) \equiv \text{list}(\alpha)) \rightarrow x) \) consists of the rules that have the following form:

\[
(\text{hd} \ (\text{cons}(x, xs) \equiv \text{list}(\alpha)) \rightarrow x) \rightarrow (\text{hd} \ (\text{cons}(x, xs) \equiv \text{list}(\alpha)) \rightarrow x)
\]

where \( \sigma \) is an arbitrary type.

**Example 2.3:** Let \( R_{\text{map}} \) be the following RFP:

\[
\begin{cases} 
\text{map} \ (\text{nil}) &\rightarrow \text{nil} \\
\text{map} \ (\text{cons}(x, xs)) &\rightarrow \text{cons} \ (f \ x, \text{map} \ (f \ x, \text{cons}(x, xs)))
\end{cases}
\]

Here we suppose that \( \text{map} : \ (\alpha \to \beta) \to \text{list}(\alpha) \to \text{list}(\beta) \).

Then we have the following reduction for \( R_{\text{list}} \cup R_{\text{map}} \):

\[
\text{hd} \ (\text{map map} \ (\text{cons} \ (\text{succ}, \text{nil}) \))) \ (\text{cons} \ (0, \text{nil})) \)
\]

\[
\text{hd} \ (\text{map} \ (\text{cons} \ (\text{succ}, \text{nil}) \))) \ (\text{cons} \ (0, \text{nil})) \)
\]

\[
\text{map} \ (\text{succ} \ (0, \text{nil})) \)
\]

\[
\text{succ} \ (0, \text{map point} \ (\text{succ} \ (0, \text{nil}))) \)
\]

A term \( t \) is terminating or strongly normalizing if there exists no infinite reduction sequence starting from \( t \). Then we denote SN(t). An RFP \( R \) is said to be terminating or strongly normalizing if every term is so. We denote by \( T_{\text{SN}} \) the set of strongly normalizing terms. We also define sets \( \bigcup T_{\text{SN}} = T_{\text{SN}} \cup T_{\text{SN}} \) and \( \bigcup T_{\text{SN}}^\text{arity} = \{\text{arity} \mid \text{arity} \in T_{\text{SN}}^\text{arity} \} \).

Since actual rewrite rules are closed under type substitution, we obtain the following proposition.
Proposition 2.4: Let \( R \) be an RFP. If \( s \xrightarrow{\xi} t \) then \( s\xi \xrightarrow{\xi} t\xi \) for any type substitution \( \xi \). Hence if any closed term is terminating then \( R \) is terminating.

3. Strong Computability, Safety Function and Safe Function-Passing

The theoretical basis of the static dependency pair method is given by the notion of strong computability, which is introduced for proving the termination of typed \( \lambda \)-calculi [7], [23]. Unfortunately the static dependency pair method is not applicable to every RFP, that is, there exists a non-terminating RFP that has no static recursive structure. The following one rule RFP is such an example.

\[
(\text{foo} \, \text{bar} \, f^{a\rightarrow b} \, y^{b}) \rightarrow (f (\text{bar} \, f^{a\rightarrow b} \, y^{b})
\
\]

From a technical viewpoint, this problem arises from the reason that strong computability is not closed under the subterm relation. For the example, some terms that are not strongly computable are accidentally passed through the higher-order variable \( f \) from the left-hand side to the right-hand side, because even if an actual argument \( \text{bar} \, t \) of \text{foo} is strongly computable, its subterm \( t \) may not be strongly computable.

From this observation, we proposed notions of plain function-passing [16] and of safe function-passing [17], under which the static dependency pair method works well. In this section, we extend the notion of safe function-passing to RFPs, with the notions of a strong computability predicate and a safety function.

To increase reusability, we divide an abstract framework from these constructions. Note that any proof in the following sections will not refer to any discussion in the constructing section (Sect. 3.2). Any proof in the following sections will refer only to the abstract framework (Sect. 3.1).

3.1 Abstract Framework

Definition 3.1: Let \( R \) be an RFP. A predicate \( SC \) over closed terms is said to be a strong computability predicate if the following properties hold:

(1) For any \( t \in T^{\text{cls}} \), if \( SC(t) \) then \( SN(t) \).
(2) For any \( f^{\alpha \rightarrow \beta} \in T^{\text{cls}} \), if \( SC(t) \) and \( SC(u) \) then \( SC(f\, t \, u) \).
(3) For any \( f^{\alpha \rightarrow \beta} \in T^{\text{cls}} \), if \( SC(t\, u) \) for all \( u^{\alpha} \in T^{\text{cls}} \) such that \( SC(u) \) then \( SC(t) \).
(4) For any \( t, u \in T^{\text{cls}} \), if \( SC(t) \) and \( t \xrightarrow{u} u \) then \( SC(u) \).
(5) For any \( t \in T^{\text{cls}} \), if \( SC(u) \) for all \( u \in T^{\text{cls}} \) then \( SC(t) \).

\( T^{SC} = \{ t \mid SC(t) \}, T_{\neg SC} = \{ t \mid \neg SC(t) \}, \) and \( T_{\neg SC}^{\text{args}} = \{ t \mid \text{args}(t) \subseteq T_{\neg SC} \}. \)

Definition 3.2: For a strong computability predicate \( SC \), a function \( \text{Safe} \) is said to be a safety function if it satisfies the following properties:

(S1) If \( u \in \text{Safe}(t) \) and \( t \in T^{\text{args}}_{\neg SC} \) then \( SC(u) \), for any \( t, u \in T^{\text{cls}} \).
(S2) If \( u \in \text{Safe}(t) \) then \( u\theta \in \text{Safe}(t\theta) \) for any \( t, u \in T^{\text{cls}} \) and term substitution \( \theta \).
(S3) If \( u \in \text{Safe}(t) \) then \( u\xi \in \text{Safe}(t\xi) \) for any closed type substitution \( \xi \).

Definition 3.3: An RFP \( R \) is said to be safe function-passing if there exists a safety function \( \text{Safe} \) for a strong computability predicate such that for any \( l \rightarrow r \in R \) and a \( \text{Sup}(r) \) with \( a \in V \), there exists \( k (k \leq n) \) such that a \( \text{Sup}(l) \) is safe(l). A safe function-passing RFP is often denoted by \( \text{SFP-RFP} \).

3.2 Constructing a Strong Computability Predicate and a Safety Function

To formulate the notion of safe function-passing in a simple type setting, we introduced notions of peeling types and peeling orders[17]. We extend these notions to RFPs, and construct a strong computability predicate and a safety function.

Definition 3.4: A set \( PT \) of peeling types is a set of data types. We define \( PT_{\geq} = \{ \sigma \mid \sigma' \geq \sigma \) for some \( \sigma' \in PT \) \}. A well-founded quasi order \( \geq_S \) on types is said to be a peeling order if the following properties hold:

• \( \sigma_1 \rightarrow \sigma_2 \geq_S \sigma_i \) for any closed types \( \sigma_1 \) and \( \sigma_2 \).
• If \( \sigma' \geq_S \sigma \) and then \( \xi(\sigma') \geq_S \xi(\sigma) \) for any closed type substitution \( \xi \).

We define the set \( \text{Sub}^{\geq_{PT}}(t) \) of peeled subterms as the smallest set satisfying the following properties:

• \( \text{args}(t) \subseteq \text{Sub}^{\geq_{PT}}(t), \)
• if \( u \equiv (a \, u^{\alpha}_{<}) \in \text{Sub}^{\geq_{PT}}(t), \sigma \in PT_{\geq}, \) and \( \sigma \geq_S \sigma_t \) then \( u_1 \in \text{Sub}^{\geq_{PT}}(t), \) and
• if \( u \equiv (a_{i_1}, \ldots, u^n_{<}) \in \text{Sub}^{\geq_{PT}}(t), \sigma \in PT_{\geq}, \) and \( \sigma \geq_S \sigma_t \) then \( u_i \in \text{Sub}^{\geq_{PT}}(t). \)

For a set \( PT \) of peeling types and a peeling order \( \geq_S \), we define the function \( \text{Safe} \) as \( \text{Safe}(t) = \text{Sub}^{\geq_{PT}}(t) \cup \{ u \mid t \xrightarrow{\text{sub}} u', \sigma \) is a data type such that \( \sigma \not\in PT_{\geq} \} \).

Definition 3.5: For a set \( PT \) of peeling types and peeling order \( \geq_S \), we define \( SC(t) \) as follows:

• In case of \( r \in T_{\neg \text{func}}^{\text{cls}} \) and \( \sigma \not\in PT_{\geq} \), \( SC(t) \) is defined as \( SN(t) \).
• In case of \( r \in T_{\neg \text{func}}^{\text{cls}} \) and \( \sigma \in PT_{\geq} \), \( SC(t) \) is defined as \( SN(t) \) and \( SC(u) \) for any \( u^{\alpha} \in T^{\text{cls}} \cap \{ \text{args}(t') \mid t \xrightarrow{t'} \} \) such that \( \sigma \geq_S \sigma' \).
• In case of \( r^{\alpha \rightarrow \beta} \in T_{\neg \text{func}}^{\text{cls}} \), \( SC(t) \) is defined as \( SC(t\, u) \) for all \( u^{\alpha} \in T^{\text{cls}} \) with \( SC(u) \).

Theorem 3.6: The predicate \( SC \) given in Definition 3.5 is a strong computability predicate.
**Proof:** We first prove the well-definedness of $SC$, that is, $SC(t)$ is defined for any $t \in T^{cls}$. Assume that $SC$ is not well-defined.

Let $t'^0_0$ be a minimal term with respect to $\succeq_S$ such that $SC(t_0)$ is undefined. From the minimality of $t_0$, we have $\sigma_0 \in PT_z, SN(t_0)$, and there exist $t'_0$ and $t_1$ such that $t_0 \rightarrow t'_0, t'_0 \rightarrow t_1 \in argS(t'_0), \sigma_0 \sim_S \sigma_1$, and $SC(t_1)$ is undefined, where $\sim_S$ is the equivalence part of $\succeq_S$.

Since $\sigma_0 \sim_S \sigma_1$, $t_1$ is also a minimal term with respect to $\succeq_S$ such that $SC(t_1)$ is undefined. By applying the procedure above, we obtain $t'_1$ and $t_2$ such that $t_1 \rightarrow t'_1, t'_2 \rightarrow argS(t'), \sigma_1 \sim_S \sigma_2$, and $SC(t_2)$ is undefined.

By applying this procedure repeatedly, we obtain $t'_2, t'_3, \ldots$ and $t_1, t_2, \ldots$ such that $t_i \rightarrow t'_i$ and $t_{i+1} \in argS(t'_i)$ for $i = 2, 3, \ldots$. Since $\rightarrow \cup \rightarrow$ is well-founded on terminating terms, this contradicts with $SN(t_0)$.

Next we will prove that the predicate $SC$ satisfies the conditions in Definition 3.1. The conditions (SC1), (SC3), and (SC5) are trivial.

**SC4** Let $SC(t', \sigma) \rightarrow t \rightarrow t'$. We prove $SC(t')$ by induction on $\sigma$. The case $\sigma \in SNf_{n}$ is trivial. Suppose that $\sigma = \sigma_1 \rightarrow \sigma_2$. Let $t^* \in \sigma_1$ be an arbitrary term such that $SC(u)$ holds. Then $SC(t \ u)$ follows from $SC(t)$ and (SC2). Since $(t \ u)^* \rightarrow t' \ u$, we have $SC(t' \ u)$ by the induction hypothesis. Hence, $SC(t')$ follows from (SC3).

**SC1** We prove the following claims by simultaneous induction on $\sigma$.

(i) If $SC(t')$ then $SN(t)$.

(ii) $SC(C(x')^t)$ for all $x \in V$.

Let $\sigma_n \rightarrow \sigma_0$ be the canonical form of $\sigma$. The case $n = 0$ is trivial. Suppose that $n > 0$.

(i): From the induction hypothesis (ii), an arbitrary variable $z_1$ is strongly computable. From (SC2), we have $SC(t \ z_1)$. From the induction hypothesis (i), $t \ z_1$ is terminating, hence so is $t$.

(ii): Assume that $\neg SC(z)$ for some variable $z$. From (SC3), there exist strongly computable terms $t_i^{r_1}, \ldots, t_i^{r_n}$ such that $z \ u_0$ is not strongly computable. From the induction hypothesis (i), each $u_i$ is terminating, hence so is $z \ u_0$. Since $(z \ u_0)^{r_0}$ is not strongly computable and $\sigma_0 \in SN_{fun}$, we have $\sigma_0 \in PT_z$ and there exist terms $v'$ and $v$ such that $z \ u_0 \rightarrow v', v \in argS(v')$, and $\sigma$ is not strongly computable. Since $root(l) \notin V$ for all $l \rightarrow r \in R$, there exists $i$ such that $u_i \rightarrow v$. From (SC4), $u_i$ is not strongly computable. This is a contradiction.

**Theorem 3.7:** The function $Safe$ given in Definition 3.4 is a safety function.

**Proof:** (S1) Let $t, u \in T^{cls}, u \in Safe(t)$ and $t \in T^{args}_S$. We prove $SC(u)$.

If $u \in [u \rightarrow \rightarrow t_{sub}^u \ u', \sigma$ is a data type such that $\sigma \notin PT_z]$, then $SC(u)$ follows from $u \rightarrow \rightarrow t_{sub} \rightarrow argS(t)$ and (SC1).

Suppose that $u'' \in Sub^{\succeq_S}(t)$. Then we have either $u \in argS(t)$ or there exists $u'' \in Sub^{\succeq_S}(t)$ such that $u \in argS(v)$, $\sigma' \in PT_z$, and $\sigma' \succeq_S \sigma$. In the former case, we have $SC(u)$ because of $t_0 \in T^{args}_{\succeq_S}$. In the latter case, it suffices to show that $SC(u)$ whenever $SC(v)$, which is directly deduced from the definition of $SC$.

(S2) It is obvious because $\rightarrow_{sub}$ is closed under term substitutions, and any term substitution does not change type information.

(S3) It is obvious because $\succeq_S$ and $PT_z$ are closed under closed type substitutions.

**Example 3.8:** We show that $Res$ as discussed in the Introduction is safe function-passing.

Since types can be interpreted as first-order terms (we interpret a product type $\sigma_1 \times \cdots \times \sigma_n$ as a first-order term $\sigma_1, \ldots, \sigma_n$), we can construct the peeling order $\succeq_S$ by using the recursive path order $\succ$ [5] with the argument filtering method [1] over first-order term rewriting systems. We take the argument filtering function by $\pi(t_\alpha) = n, \pi(\rightarrow) = [1, 2]$, and $\pi(c) = [1, \ldots, arity(c)]$ for any $c \in TC$. Then the order $\succ_{\succ}$, defined as $\sigma_1 \succ_{\succ} \sigma_2$ if $\pi(\sigma_1) \succ_{\succ} \pi(\sigma_2)$, becomes a peeling order. We take $PT$ as the set of all data types.

The first rule of $Res$ trivially satisfies the desired property. Suppose that $t \equiv foldl \ f \ e \ (cons \ (x, x))$. Then:

- we have $f, e, cons \ (x, x) \in Sub^{\succeq_S}(t)$ because of $argS(t) \subset Sub^{\succeq_S}(t)$,
- we have $(x, x) \in Sub^{\succeq_S}(t)$ because of $cons \ (x, x) \in Sub^{\succeq_S}(t), list(\alpha) \in PT$, and $\pi(list(\alpha)) = list(\alpha) \succeq_S list(\alpha) \succeq_S list(\alpha) = \pi(\alpha \times list(\alpha))$, and
- we have $x, x \in Sub^{\succeq_S}(t)$ because of $(x, x) \in Sub^{\succeq_S}(t), cons \ (x, x) \in Sub^{\succeq_S}(t), \alpha \times list(\alpha) \in PT, \pi(\alpha \times list(\alpha)) = list(\alpha) \succeq_S list(\alpha) = \pi(\alpha \times list(\alpha)) = list(\alpha) \succeq_S \alpha = \pi(\alpha)$.

Hence we have $Safe(t) = \{f, e, cons \ (x, x), \alpha \times list(\alpha), \alpha, list(\alpha)\}$, and then the second rule of $Res$ also satisfies the desired property. Therefore $Res$ is safe function-passing.

**4. Static Dependency Pair Method**

The static dependency pair method is a powerful method to prove termination, which was introduced on STRSs [16], [17], and extended to HRSSs [18], [22]. In this section, we extend the method to RFPs.

**Definition 4.1:** Let $R$ be an SFP-RFP. All root symbols of the left-hand sides of rewrite rules, denoted by $\mathcal{D}_R$, are called defined symbols, whereas all other function symbols, denoted by $\mathcal{C}_R$, are constructors.

For each $f \in \mathcal{D}_R$, we provide a new function symbol $f^3$, called the marked-symbol of $f$. For each $t \equiv a \in \mathcal{D}_R$, we define the marked term $t^3$ by $a^3_t$.

A pair $(\bar{f}, a^3_t)$, denoted by $\bar{f} \rightarrow a^3_t$, is said to be an outer static dependency pair in $R$ if there exists a rule $l \rightarrow a \in \mathcal{D}_R$ satisfying the following conditions:
Definition 4.4: Let $\sigma \in \Sigma_{\text{sf}}$. Then $\sigma$ is said to be a static dependency chain in $R$ if there exist a non-empty leaf-context $C[]$ and $l \in C[a_{\tau_0}] \in R$ satisfying the two conditions above.

A static dependency pair in $R$ is an outer or inner static dependency pair. We denote by $SDP(R)$ the set of static dependency pairs in $R$.

Example 4.2: We consider the SFP-RFP $R_{\text{sf}}$, that is, the union of $R_{\text{foldl}}$, $R_{\text{map}}$ and the following rules:

\[
\begin{align*}
\text{add}((0,), y) & \rightarrow y \\
\text{add}(\text{succ}(x,), y) & \rightarrow \text{succ}(\text{add}(x, y)) \\
\text{sum} & \rightarrow \text{foldl} \text{ add} 0 \\
\text{sigma} f \text{ xs} & \rightarrow \text{sum}(\text{map} f \text{ xs})
\end{align*}
\]

where $R_{\text{foldl}}$ and $R_{\text{map}}$ are displayed in the Introduction and Example 2.3, respectively. Here we suppose that add : Nat $\times$ Nat $\rightarrow$ Nat, sum : list(Nat) $\rightarrow$ Nat, and sigma : (Nat $\rightarrow$ Nat $\rightarrow$ list(a a)) $\rightarrow$ Nat. The function calculates the total sum for an input list, and the function sigma f xs calculates $\Sigma_{\text{sf}}(f(i))$. Note that similar to Example 3.8 we can prove that $R_{\text{sf}}$ is safe function-passing. Then there are three outer static dependency pairs:

\[
\begin{align*}
\text{foldl} \text{ f e (cons (x, y))} & \rightarrow \text{foldl} \text{ f (f (e, x)) y} \\
\text{sum} & \rightarrow \text{foldl} \text{ add} 0 \\
\text{sigma} f \text{ xs} & \rightarrow \text{sum}(\text{map} f \text{ xs})
\end{align*}
\]

and there are four inner static dependency pairs:

\[
\begin{align*}
\text{map} \text{ f (cons (x, y))} & \rightarrow \text{map} \text{ f xs} \\
\text{add} \text{ (succ (x, y))} & \rightarrow \text{add} \text{ (x, y)} \\
\text{sum} & \rightarrow \text{add} \\
\text{sigma} f \text{ xs} & \rightarrow \text{map} f \text{ xs}
\end{align*}
\]

Definition 4.3: Let $R$ be an SFP-RFP. For any outer static dependency pair $u \rightarrow v$, we define the set Act$(u \rightarrow v)$ of actual outer static dependency pairs as: $\rightarrow \in \text{Act}(u \rightarrow v)$ : $\sigma$ iff $\rightarrow$ and $\rightarrow'$ are closed terms, and there is a type substitution $\xi$ such that $\rightarrow' \equiv \xi(\rightarrow)$. Then $\rightarrow$ and $\rightarrow'$ are fresh variables.

For any inner static dependency pair $u \rightarrow v$, we define the set Act$(u \rightarrow v)$ of actual inner static dependency pairs as: $\rightarrow \in \text{Act}(u \rightarrow v)$ : $\sigma$ iff $\rightarrow$ and $\rightarrow'$ are closed terms, and there is a type substitution $\xi$ such that $\rightarrow' \equiv \xi(\rightarrow)$. Then $\rightarrow$ and $\rightarrow'$ are fresh variables.

An actual static dependency pair in $R$ is an actual outer or inner static dependency pair. We denote by Act$(SDP(R))$ the set of actual static dependency pairs in $R$.

Definition 4.4: Let $R$ be an SFP-RFP. A sequence $u_1 \rightarrow v_1, u_2 \rightarrow v_2, \ldots$ of static dependency pairs in $R$ is said to be a static dependency chain in $R$ if there exist $s_1 \rightarrow r_1 \in \text{Act}(u_1 \rightarrow v_1), s_2 \rightarrow r_2 \in \text{Act}(u_2 \rightarrow v_2), \ldots$, and term substitutions $\sigma_1, \sigma_2, \ldots$ such that for any $i, r_i \sigma_i \rightarrow_r s_{i+1} \sigma_{i+1}$, and $s_i, t_i \in T_{\text{args}}$.

Lemma 4.5: If an SFP-RFP $R$ is not terminating then $T_{\text{args}} \cap T_{\text{act}} \cap T_{\text{args}} \neq \emptyset$.

Proof: From Proposition 2.4, we have $T_{\text{cl}} \cap T_{\text{act}} \neq \emptyset$. From (SC1), we have $T_{\text{cl}} \cap T_{\text{act}} \cap T_{\text{args}} \neq \emptyset$. Let $\tau$ be a minimal term in $T_{\text{cl}} \cap T_{\text{act}}$ with respect to term size. From the minimality, we have $\tau \in T_{\text{args}}$. Hence, we have $T_{\text{cl}} \cap T_{\text{act}} \cap T_{\text{args}} \neq \emptyset$.

Lemma 4.6: Let $R$ be an SFP-RFP. If $\tau' \in T_{\text{cl}} \cap T_{\text{act}} \cap T_{\text{args}}$ then root$(\tau) \in D_R$.

Proof: Assume that $\text{root}(\tau) \notin D_R$. Let $\tau_0 \rightarrow \sigma_0$ be the canonical form of $\sigma$. From (SC3), there are $u_1, \ldots, u_n$, such that $\forall l \in \text{SC}(u_i)$ and $\neg l \in \text{ST}(\tau_0)$. Then the termination of $\tau_0$ follows from root$(\tau) \notin D_R$, $\tau_0 \in T_{\text{cl}}$ and (SC1).

From (SC5) there exists $t_1 \in T_{\text{cl}}$ such that $t_1 \rightarrow t_0$ with $\neg l \in \text{ST}(t_1)$, because $t_1 \in T_{\text{cl}} \cap T_{\text{act}} \cap T_{\text{args}}$. From root$(\tau) \notin D_R$ and (SC4), we have $t_1 \in T_{\text{cl}} \cap T_{\text{act}} \cap T_{\text{args}}$. From this in a similar way, there is $t_2 \in T_{\text{cl}} \cap T_{\text{act}} \cap T_{\text{args}}$ such that $t_1 \rightarrow t_2$ and root$(t_2) \notin D_R$. By applying this procedure repeatedly, we construct an infinite sequence $\tau_0 \rightarrow t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots$, which leads to a contradiction with the termination of $\tau_0$.

Lemma 4.7: Let $R$ be an SFP-RFP. If $\tau \in T_{\text{cl}} \cap T_{\text{act}} \cap T_{\text{args}}$ then there exist $t \rightarrow r \in \text{Act}(R)$ and $\theta'$ such that $\theta' \in T_{\text{act}} \cap T_{\text{args}}$ and root$(\tau) \notin D_R$.

Proof: We proceed by induction on $t$ ordered by $\rightarrow$. From $\tau \in T_{\text{cl}}$ and (SC5), there exist $t \rightarrow \tau$ such that $\tau \rightarrow \tau'$. In the case where a redex is rewritten in $t \rightarrow \tau'$, there exist $l \rightarrow r \in \text{Act}(R)$ and $\theta'$ such that $\theta' \in T_{\text{cl}} \cap T_{\text{act}} \cap T_{\text{args}}$ and root$(\tau) \notin D_R$. Then the desired property holds. In other cases, since $\tau \in T_{\text{cl}}$ follows from (SC4), the desired property follows from the induction hypothesis.

Lemma 4.8: Let $R$ be an SFP-RFP. For any $\tau \in T_{\text{cl}} \cap T_{\text{act}} \cap T_{\text{args}}$ there exist $u \rightarrow v \in \text{Act}(SDP(R))$ and term substitution $\theta$ such that $\theta \in T_{\text{cl}} \cap T_{\text{act}} \cap T_{\text{args}}$.

Proof: Let $\tau' \in T_{\text{cl}} \cap T_{\text{act}} \cap T_{\text{args}}$. Then $\tau \in T_{\text{args}}$ follows from $\tau \in T_{\text{cl}} \cap T_{\text{act}} \cap T_{\text{args}}$.

Consider the case of $\tau \notin T_{\text{cl}}$. Since $\tau \in T_{\text{cl}}$, there exist closed rewrite rules $l \rightarrow r \in \text{Act}(R)$ and closed term substitution $\theta'$ such that $\theta' \in T_{\text{cl}} \cap T_{\text{act}} \cap T_{\text{args}}$. From (SC1) and (SC4), we have $\theta' \in T_{\text{cl}} \cap T_{\text{act}} \cap T_{\text{args}}$.
Consider the case of \( t \in \mathcal{T}_{sc} \). From Lemma 4.7, there exist closed rewrite rule \( l \to r \in \text{Act}(R) \) and closed term substitution \( \sigma' \) such that \( l^\dagger \xrightarrow{\sigma'} l^\dagger \sigma' \), \( l^\dagger \in \mathcal{T}_{nfun} \cap \mathcal{T}_{arg} \cap \mathcal{T}_{sc} \), and \( r^\dagger \in \mathcal{T}_{sc} \).

In both cases above, we have \( l^\dagger \in \mathcal{T}_{nfun} \cap \mathcal{T}_{sc} \cap \mathcal{T}_{arg} \) and \( \{ v'' \in \text{Sub}(r) \mid v'' \sigma' \in \mathcal{T}_{sc} \} \neq \emptyset \) because \( r \in \text{Sub}(r) \) and \( \neg \text{SC}(r^\dagger) \). Let \( v' : \sigma \) be a minimal size term in \( \{ v'' \in \text{Sub}(r) \mid v'' \sigma' \in \mathcal{T}_{sc} \} \) and \( \overline{v'} = \sigma_0 \) be the canonical form of \( \sigma \). From (SC3), there exist strongly computable closed terms \( \overline{v''} \) such that \( v'' \sigma' \overline{v''} \sigma' \in \mathcal{T}_{sc} \). From the minimality of \( v'' \), we have \( v'' \sigma' \overline{v''} \sigma' \in \mathcal{T}_{arg} \). Hence we have \( v'' \sigma' \overline{v''} \sigma' \in \mathcal{T}_{nfun} \cap \mathcal{T}_{sc} \cap \mathcal{T}_{arg} \). From Lemma 4.6, \( v' \) has the form of \( a \overline{r} \) with \( a \in \mathcal{D}_R \).

Now take \( v' = a \overline{r} \), where each \( \overline{r}_i \) is a fresh variable, and \( \theta(x) \) is defined by \( \theta \) if \( x = z_i (i = 1, \ldots, n) \); otherwise by \( \theta(x) \). Then we have \( l^\dagger \equiv l^\dagger \sigma' \) and \( v\theta = v'' \sigma' \overline{v''} \sigma' \). Hence we have \( l^\dagger \sigma' \overline{v''} \sigma' \). From (SC1), we have \( l^\dagger \sigma' \overline{v''} \sigma' \). It suffices to show that \( l^\dagger \sigma' \overline{v''} \sigma' \) is a directed graph, in which nodes are \( l \) and \( v \).

Consider the case of \( v'' \equiv v_i \). Then \( v'' \equiv v_i \). The form of \( a \overline{r}_{n-k} \) such that \( r_i = r_{i+k} \) for any \( i = 1, \ldots, n-k \), and \( r_{k+i} \) for any \( i = n-k+1, \ldots, k \). Assume \( a \overline{r}_i \in \text{Safe}(l') \) for some \( k \). From (S3), we have \( a \overline{r}_i \in \text{Safe}(l) \).

We now introduce the notions of static dependency graph, static recursion component and non-loopingness. As usual, the termination of SFP-RFPs can be proved by proving the non-loopingness of each static recursion component. These proofs are similar to other dependency pair methods.

**Definition 4.10:** Let \( R \) be an SFP-RFP. The static dependency graph of \( R \) is a directed graph, in which nodes are \( \text{SDP}(R) \) and there exists an arc from \( u^\dagger \to v^\dagger \) if \( u^\dagger \to v^\dagger \) is a static dependency chain.

**Example 4.11:** The static dependency graph of the SFP-RFP \( R_{\text{sigma}} \) (cf. Example 4.2) is displayed in Fig. 1.

**Definition 4.12:** Let \( R \) be an SFP-RFP. A static recursion component in \( R \) is a set of nodes in a strongly connected subgraph of the static dependency graph of \( R \). Using \( \text{SRC}(R) \) we denote the set of static recursion components in \( R \).

**Example 13:** The static dependency graph of \( R_{\text{sigma}} \) (cf. Example 4.11) has three strongly connected subgraphs. Thus, the set \( \text{SRC}(R_{\text{sigma}}) \) consists of the following three components:

\[
\{ \text{add}(\text{succ } x, y) \to \text{add}(x, y) \},
\{ \text{map}(\text{cons } (x, x)) \to \text{map } f x \},
\{ \text{fold}(\text{cons } (x, x)) \to \text{fold}(\text{cons } (e, x)) \}
\]

**Definition 4.14:** Let \( R \) be an SFP-RFP. A static recursion component \( C \in \text{SRC}(R) \) is said to be non-looping if there exists no infinite static dependency chain in which only pairs in \( C \) occur and every \( u^\dagger \to v^\dagger \). From Theorem 4.9, we obtain the following corollary.

**Corollary 4.15:** Let \( R \) be an SFP-RFP. If all static recursion components are non-looping then \( R \) is terminating.

**5. Proving Non-loopingness**

When proving termination by dependency pair methods,
non-loopingness should be shown for each recursion component (cf. Corollary 4.15). To prove the non-loopingness of components, the notions of subterm criterion and reduction pair have been proposed. The subterm criterion was introduced on TRSSs [9], and slightly improved by extending the subterms permitted by the criterion on STRSSs [16], and extended on HRSs [18]. Reduction pairs [15] are an abstraction of the notion of the weak-reduction orders [1]. In this section, we extend the notions to RFPs.

**Definition 5.1:** A pair $(\succ, \succ)$ of relations on terms is a reduction pair if $\succ$ and $\succ$ satisfy the following properties:

- $\succ$ is well-founded and closed under term substitutions,
- $\succ$ is closed under contexts, type substitutions and term substitutions,
- and $\succ \cdot \succ \subseteq \succ$ or $\succ \cdot \succ \subseteq \succ$.

In particular, $\succ$ is said to be a weak reduction order if $(\succ, \succ)$ is a reduction pair.

**Definition 5.2:** Let $R$ be an RFP and $C$ be a set of static dependency pairs. We say that $C$ satisfies the subterm criterion if there exists a function $\pi$ from $D_R$ to non-empty sequences of positive integers such that:

1. $u_0 \succ u_1 \succ u_2 \succ \cdots$ for some $u_i \rightarrow v_i \in C$, and
2. the following conditions hold for any $u_i \rightarrow v_i \in C$:
   - $u_i \succ u_{i+1}$,
   - $(u_i) \notin V$ for all $p \prec \pi(u_i)$, and
   - $q \neq e \Rightarrow (\pi) \in C'$ for all $q \prec \pi(v_i)$.

**Theorem 5.3:** Let $R$ be an SFP-RFP. Then, $C \in S R C(R)$ is non-looping if $C$ satisfies one of the following properties:

- There is a reduction pair $(\succ, \succ)$ such that $R \subseteq \succ$, $\text{Act}(C) \subseteq \succ \cup \succ$, and $\text{Act}(u_i \rightarrow v_i) \subseteq \succ$ for some $u_i \rightarrow v_i \in C$.
- $C$ satisfies the subterm criterion.

**Proof:** Assume that there exists an infinite static dependency chain $u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow \cdots$ in which only pairs in $C$ occur and every $u_i \rightarrow v_i \in C$ occurs infinitely many times. From the definition of the static dependency chain, there are $s_i \rightarrow t_i \in \text{Act}(u_i \rightarrow v_i)$, $s_i \rightarrow t_i \in \text{Act}(u_i \rightarrow v_i)$, and term substitutions $\theta_1, \theta_2, \ldots$, such that for any $i$, $s_i \rightarrow t_i \rightarrow s_i \rightarrow t_i \rightarrow \cdots$ in $C$, and $s_i \rightarrow t_i \notin \mathcal{T}_S$.

- Suppose that there is a reduction pair $(\succ, \succ)$ such that $R \subseteq \succ$, $\text{Act}(C) \subseteq \succ \cup \succ$, and $\text{Act}(u_i \rightarrow v_i) \subseteq \succ$ for some $u_i \rightarrow v_i \in C$. Since $\succ$ is closed under contexts, type substitutions and term substitutions, $\succ \subseteq \succ$ follows from $R \subseteq \succ$. Since $\succ$ is closed term substitutions, $s_i \rightarrow t_i \in \text{Act}(C) \subseteq \succ \cup \succ$. Hence we have $s_i \rightarrow t_i \in \text{Act}(C) \subseteq \succ \cup \succ$. From $\text{Act}(u_i \rightarrow v_i) \subseteq \succ$ for some $u_i \rightarrow v_i \in C$, this sequence contains infinitely many $\succ$. This is a contradiction with the well-foundedness of $\succ$ and $\succ \cdot \succ \subseteq \succ$ or $\succ \cdot \succ \subseteq \succ$.

- Suppose that $C$ satisfies the subterm criterion. We note that $u_i \rightarrow v_i$ (resp. $u_i \rightarrow v_i$) guarantees $s_i \rightarrow t_i$ (resp. $s_i \rightarrow t_i$) for any static dependency pair $u_i \rightarrow v_i$ and $s_i \rightarrow t_i \in \text{Act}(u_i \rightarrow v_i)$.

We denote $\pi(\text{root}(u_i))$ by $p_i$ for each $i$. Since $t_i \rightarrow s_i \rightarrow s_i \rightarrow t_i \rightarrow \cdots$, we have $\text{root}(t_i) \rightarrow \text{root}(s_i \rightarrow t_i)$. From the last two conditions in (ii) of the subterm criterion, we have $t_i \rightarrow s_i \rightarrow t_i \rightarrow \cdots$ for each $i$. Hence, from the first condition in (ii) of the subterm criterion, we have $s_0 \rightarrow t_0 \rightarrow s_1 \rightarrow t_1 \rightarrow \cdots s_2 t_2 \rightarrow t_2 \rightarrow \cdots$. From the condition (i) of the subterm criterion, this sequence contains infinitely many $\succ \cdot \succ$. Since $\succ \cdot \succ$ is well-founded and $\succ \cdot \succ \subseteq \succ$, there exists an infinite rewriting relation starting from $s_0 t_0$, that is, $s_0 t_0$ is not terminating. Since $p_0$ is non-empty, we have $s_0 t_0 \notin \mathcal{T}_S$. From the definition of the static dependency chain, we have $s_0 t_0 \in \mathcal{T}_S$.

From (SC1), we have $s_0 t_0 \in \mathcal{T}_S$, which leads to a contradiction.

**Example 5.4:** Let $\pi(\text{add}) = 1.1$, $\pi(\text{map}) = 2$, and $\pi(\text{foldl}) = 3$. Then, every static recursion component $C$ (cf. Example 4.13) satisfies the subterm criterion in the underlined positions below:

- $\text{add}(\text{succ} x, y) \rightarrow \text{add}(\text{succ} x, y)$,
- $\text{map}(f \cdot \text{cons}(x, x)) \rightarrow \text{map}(f \cdot x)$,
- $\text{foldl}(f \cdot \text{cons}(x, x)) \rightarrow \text{foldl}(f \cdot (f (x, x)) \cdot x)$.

Hence, from Theorem 5.3 these static recursion components are non-looping. Therefore the termination of $R_{\text{Sigma}}$ follows from Corollary 4.15.

In the Introduction, we said that the polymorphic-typed combinatory logic is an example that shows the strong efficacy of the static dependency pair method. Finally together with other well-known combinators [13], we give an elegant termination proof by the static dependency pair method.

**Example 5.5:** Let $R$ be the following RFP:

- $(S \cdot f g x y) \rightarrow f x (g x)$
- $(K \cdot x y) \rightarrow x$
- $(I \cdot x y) \rightarrow x$
- $(B \cdot x y z) \rightarrow x (y z)$
- $(B' \cdot x y z) \rightarrow y (x z)$
- $(C \cdot x y z) \rightarrow x z y$
- $(J \cdot x y z) \rightarrow y (x w z)$

Since any variable occurs in an argument position on the left-hand sides, $R$ is trivially safe function-passing. Since $S D P(R) = \emptyset$ and hence $S R C(R) = \emptyset$, the termination of $R$ follows from Corollary 4.15.

**6. Concluding Remarks**

In this paper, we present the static dependency pair method, which proves the termination of SFP-RFPs.
To prove termination effectively, the argument filtering method and the notion of usable rules are indispensable. The argument filtering method generates reduction pairs from reduction orders. The method was introduced for TRSs [1], and extended to STRSs [14], [17], and to HRSs [22]. In future research we will extend the method to RFPs. The notion of usable rules optimize a constraint generated by the dependency pair method. This analysis was first conducted for TRSs [6], [9], and has been extended to STRSs [17], [21], and to HRSs [22]. In the future we will extend the notion on RFPs.

To generate reduction pairs by the argument filtering method, it is also indispensable to construct reduction orders. Recently, an effective and practicable reduction order, namely higher-order recursive path orderings, was introduced [3], [4], [11]. We will import the orderings to RFPs in the future.

Since the static dependency pair method cannot apply to every RFP, it is important to expand its applicable class. To design the notion “General Scheme” for proving termination, Blanqui, Jouannaud, and Okada introduced the notion of accessibility [2]. Several extensions of the accessibility was introduced [3], [4]. We think that the accessibility has the similar motivation as our safety function. Hence, by importing the notion to our static dependency pair method, we can expect to expand the applicable class. This will also be future work. We note that the abstract framework for the strong computability and the safety function in Sect. 3 has the purpose of this future work.

Developing a termination prover for RFPs based on our results will also be future work.

Acknowledgments

We would like to thank the anonymous referees for their helpful comments.

This work was supported by KAKENHI #24500012.

References


Keiichirou Kusakari received B.E. from Tokyo Institute of Technology in 1994, received M.E. and the Ph.D. degree from Japan Advanced Institute of Science and Technology in 1996 and 2000. From 2000, he was a research associate at Tohoku University. He transferred to Nagoya University’s Graduate School of Information Science in 2003 as an associate professor and became an associate professor in 2006. His research interests include term rewriting systems, program theory, and automated theorem proving. He is a member of IPSJ and JSSST.