PAPER Special Section on Foundations of Computer Science

Static Dependency Pair Method for Simply-Typed Term Rewriting and Related Techniques

Keiichirou KUSAKARI[†a)] *and* **Masahiko SAKAI**[†b)], *Members*

SUMMARY A static dependency pair method, proposed by us, can effectively prove termination of simply-typed term rewriting systems (STRSs). The theoretical basis is given by the notion of strong computability. This method analyzes a static recursive structure based on definition dependency. By solving suitable constraints generated by the analysis result, we can prove the termination. Since this method is not applicable to every system, we proposed a class, namely, plain function-passing, as a restriction. In this paper, we first propose the class of safe function-passing, which relaxes the restriction by plain function-passing. To solve constraints, we often use the notion of reduction pairs, which is designed from a reduction order by the argument filtering method. Next, we improve the argument filtering method for STRSs. Our argument filtering method does not destroy type structure unlike the existing method for STRSs. Hence, our method can effectively apply reduction orders which make use of type information. To reduce constraints, the notion of usable rules is proposed. Finally, we enhance the effectiveness of reducing constraints by incorporating argument filtering into usable rules for STRSs.

key words: *simply-typed term rewriting, termination, static dependency pair method, argument filtering, usable rule*

1. Introduction

A simply-typed term-rewriting system (STRS), proposed by Kusakari, is a computational model that provides operational semantics for functional programs and directly handles higher-order functions [18]. For example, the left-folding function `foldl`, a typical higher-order function, is represented as the following STRS $R_{\mathtt{foldl}}$:

$$\begin{cases} \mathtt{foldl}[f, y, \mathtt{nil}] & \to & y \\ \mathtt{foldl}[f, y, \mathtt{cons}[x, xs]] & \to & \mathtt{foldl}[f, f[y, x], xs] \end{cases}$$

Using the function `foldl`, the `sum` function, which calculates the total sum for an input list, can be represented as STRS $R_{\mathtt{sum}}$, which is the union of $R_{\mathtt{foldl}}$ and the following rules:

$$\begin{cases} \mathtt{add}[0, y] & \to & y \\ \mathtt{add}[\mathtt{s}[x], y] & \to & \mathtt{s}[\mathtt{add}[x, y]] \\ \mathtt{sum} & \to & \mathtt{foldl}[\mathtt{add}, 0] \end{cases}$$

A dependency pair method, proposed by Arts and Giesl, is a method for proving termination of first-order term rewriting systems (TRSs) based on recursive structure analysis [1]. In higher-order settings, there are two kinds of analysis for recursive structures. One is a dynamic analysis based on function-call dependency, and the other is a static analysis based on definition dependency. In other words, a dynamic dependency pair method considers a dependency through higher-order variables, but a static dependency pair method need not consider such a dependency. Hence, a static dependency pair method has more practical advantage than a dynamic method. Dynamic dependency pair methods were introduced in STRSs [18] and in HRSs [24], which are natural extensions of the dependency pair method in TRSs [1]. We also proposed a static dependency pair method in [22]. The key idea of the static dependency pair method is to analyze a recursive structure from the viewpoint of strong computability, which was introduced for proving termination in typed λ-calculus [12], [27]. For the STRS $R_{\mathtt{sum}}$, the static dependency pair method returns the following two static recursion components:

$$\{\mathtt{foldl}^\sharp[f, y, \mathtt{cons}[x, xs]] \to \mathtt{foldl}^\sharp[f, f[y, x], xs]\}$$
$$\{\mathtt{add}^\sharp[\mathtt{s}[x], y] \to \mathtt{add}^\sharp[x, y]\}$$

We can effectively and efficiently prove the termination of STRSs by showing the non-loopingness of these components as will hereinafter be described in detail.

Unfortunately static dependency pair methods are not applicable to every STRSs, that is, there exists a non-terminating STRS that has no static recursive structure. The STRS $\{\mathtt{foo}[\mathtt{bar}[f]] \to f[\mathtt{bar}[f]]\}$ is a such example. Hence, we need a suitable restriction under which static dependency pair methods work well. As such a restriction, we proposed the notion of plain function-passing [22]. Roughly speaking, plain function-passing means that every higher-order variable occurs in an argument position on the left-hand side. For example, the STRS $R_{\mathtt{app}_0}$

$$\begin{cases} \mathtt{app}_0[\mathtt{nilF}] & \to & \mathtt{nil} \\ \mathtt{app}_0[\mathtt{consF}[\underline{f}, fs]] & \to & \mathtt{cons}[f[0], \mathtt{app}_0[fs]] \end{cases}$$

is not plain function-passing because the underlined occurrence of the higher-order variable f is not an argument position. Hence, the static dependency pair method in [22] was not applicable to $R_{\mathtt{app}_0}$. In this paper, we introduce the notion of a peeling order, and by using this notion we introduce the notion of safe function-passing, which expands the application range of the static dependency pair method. Thus, we can apply the static dependency pair method to $R_{\mathtt{app}_0}$.

To show the non-loopingness of each static recursion

Manuscript received April 1, 2008.
Manuscript revised July 2, 2008.

[†]The authors are with the Graduate School of Information Science, Nagoya University, Nagoya-shi, 464–8603 Japan.
a) E-mail: kusakari@is.nagoya-u.ac.jp
b) E-mail: sakai@is.nagoya-u.ac.jp

DOI: 10.1587/transinf.E92.D.235

component, we often use reduction pairs or the subterm criterion. The argument filtering method generates a reduction pair from a given reduction order. This method was introduced in TRSs [1], and extended to STRSs [18]. However the method does not work well in general STRSs and may destroy the well-typedness of terms. In [18], we showed that the method works well in left-firmness STRSs, that is, any variable of the left-hand sides occurs at a leaf position. On the other hand, destroying the well-typedness remarkably complicates the application of the argument filtering method to reduction orders which make use of type information [19]. In this paper, we improve the argument filtering method. Although the improved method requires that target STRSs is left-firmness, this never destroys the well-typedness. In spite of the fact that the idea is simple, our improvement yields very substantial benefits when combined with reduction orders that make use of type information. In contrast to the discussion about the applications of the argument filtering method in [19], we need not individually discuss application to each reduction order, and we can comb out some applied conditions.

To reduce the number of constraints when proving the non-loopingness by reduction pairs, the notion of usable rules was introduced in TRSs [11], [15], [29]. We extended the notion onto STRSs [26]. In first-order TRSs, we know that usable rules can be strengthened by incorporating argument filtering into usable rules [11], [29]. In this paper, we also strengthen usable rules by incorporating argument filtering into usable rules for STRSs.

The remainder of this paper is organized as follows. The next section provides preliminaries required later in the paper. In Sect. 3, we introduce the notion of safe function-passing, and show that the static dependency pair method works well in safe function-passing STRSs. In Sect. 4, we introduce the argument filtering method which never destroys the well-typedness, unlike in existing method. In Sect. 5, we strengthen usable rules by incorporating argument filtering into usable rules for STRSs. Concluding remarks are presented in Sect. 6.

2. Preliminaries

Untyped term rewriting systems (UTRSs) were introduced by removing arity constraints from first-order term rewriting systems (TRSs), and simply-typed term rewriting systems (STRSs) were introduced as UTRSs with simple-type constraints [18].

In this section, we introduce the basic notations for simply-typed term rewriting systems, according to the literature [22]. We assume that the reader is familiar with notions of term rewriting systems [28].

2.1 Abstract Reduction System

An *abstract reduction system* (ARS) is a pair $\langle A, \to \rangle$ where A is a set and $\to$ is a binary relation on A. The transitive-reflexive closure and the transitive closure of a binary relation $\to$ are denoted by $\xrightarrow{*}$ and $\xrightarrow{+}$, respectively. An element $a \in A$ is said to be *terminating* or *strongly normalizing* in an ARS $R = \langle A, \to \rangle$, denoted by $SN(R, a)$, if every reduction sequence staring from a is finite. An ARS $R = \langle A, \to \rangle$ is said to be *terminating* or *strongly normalizing*, denoted by $SN(R)$, if $SN(R, a)$ holds for any $a \in A$.

2.2 Untyped Term Rewriting System

The set $\mathcal{T}(\Sigma, \mathcal{V})$ of *(untyped) terms* generated from a set Σ of *function symbols* and a set $\mathcal{V}$ of *variables* with $\Sigma \cap \mathcal{V} = \emptyset$ is the smallest set such that $a[t_1, \ldots, t_n] \in \mathcal{T}(\Sigma, \mathcal{V})$ whenever $a \in \Sigma \cup \mathcal{V}$ and $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$. If $n = 0$, we write a for $a[\,]$. The identity of terms is denoted by $\equiv$. We often write $s_0[s_1, \ldots, s_n]$ for $a[u_1, \ldots, u_k, s_1, \ldots, s_n]$, where $s_0 \equiv a[u_1, \ldots, u_k]$. $Var(t)$ is the set of variables in t, and $args(t)$ is the set of *arguments* in t, defined as $args(a[t_1, \ldots, t_n]) = \{t_1, \ldots, t_n\}$.

The set of *positions* of a term t is the set $Pos(t)$ of strings over positive integers, which is inductively defined as $Pos(a[t_1, \ldots, t_n]) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in Pos(t_i)\}$. The *prefix order* $\prec$ on positions is defined by $p \prec q$ iff $pw = q$ for some w ($\neq \varepsilon$). The position ε is said to be the *root*, and a position p such that $p \in Pos(t) \wedge p1 \notin Pos(t)$ is said to be a *leaf*. The symbol at position p in t is denoted by $(t)_p$. Sometimes the root symbol $(t)_\varepsilon$ in a term t is denoted by $root(t)$.

A *substitution* θ is a mapping from variables to terms. A substitution θ is extended to a mapping from terms to terms, denoted by $\hat{\theta}$, as $\hat{\theta}(f[t_1, \ldots, t_n]) = f[\hat{\theta}(t_1), \ldots, \hat{\theta}(t_n)]$ if $f \in \Sigma$; $\hat{\theta}(z[t_1, \ldots, t_n]) = a[u_1, \ldots, u_k, \hat{\theta}(t_1), \ldots, \hat{\theta}(t_n)]$ if $z \in \mathcal{V}$ with $\theta(z) = a[u_1, \ldots, u_k]$. For simplicity, we identify θ and $\hat{\theta}$, and write $t\theta$ instead of $\theta(t)$.

A *context* is a term with one occurrence of the special symbol $\Box$, called a hole. The notation $C[t]$ denotes the term obtained by substituting t into the hole of $C[\,]$, that is, $C[t] \equiv a[t_1, \ldots, t_n, u_1, \ldots, u_k]$ if $C[\,] \equiv \Box[u_1, \ldots, u_k]$ and $t \equiv a[t_1, \ldots, t_n]$, and $C[t] \equiv a[\ldots, C'[t], \ldots]$ if $C[\,] \equiv a[\ldots, C'[\,], \ldots]$. A context is said to be a *leaf-context* if the hole occurs at a leaf position, and to be a *root-context* if the hole occurs at the root position. For example, $\mathtt{s}[\Box]$ and $\mathtt{foldl}[f, \Box]$ are leaf-contexts, $\Box[0]$ and $\Box[f, \mathtt{nil}]$ are root-contexts, and $\Box$ is a leaf-context and a root-context.

A term u is said to be a *subterm* (resp. an *extended subterm*) of t, denoted by $t \geq_{sub} u$ (resp. $t \geq_{esub} u$), if there exists a leaf-context (resp. context) $C[\,]$ such that $t \equiv C[u]$. We also define $>_{sub} = \geq_{sub} \setminus \equiv$ and $>_{esub} = \geq_{esub} \setminus \equiv$. We denote all subterms (resp. extended subterms) of t by $Sub(t)$ (resp. $ESub(t)$). The subterm of t at position p is denoted by $t|_p$. For example, $Sub(a'[a[x, y]]) = \{a'[a[x, y]], a[x, y], x, y\}$ and $ESub(a'[a[x, y]]) = \{a'[\,], a[\,], a[x]\} \cup Sub(a'[a[x, y]])$. A term u is said to be a *prefix* of a term t, denoted by $u \sqsubseteq t$, if t has the form $u[u_1, \ldots, u_n]$.

A *rule* is a pair (l, r) of terms, denoted by $l \to r$, such that $root(l) \in \Sigma$ and $Var(l) \supseteq Var(r)$. The *reduction relation* $\xrightarrow[R]{}$ of a set R of rules is defined by $s \xrightarrow[R]{} t$ iff $s \equiv C[l\theta]$ and $t \equiv C[r\theta]$ for some rule $l \to r \in R$, context $C[\,]$ and substitution θ. We often omit the subscript R whenever no confusion

arises. An *untyped term rewriting system* (UTRS) is an abstract reduction system $\langle \mathcal{T}(\Sigma, \mathcal{V}), \underset{R}{\rightarrow} \rangle$. We often denote an UTRS $\langle \mathcal{T}(\Sigma, \mathcal{V}), \underset{R}{\rightarrow} \rangle$ by R.

2.3 Simply-Typed Term Rewriting System

A set of *basic types* is denoted by $\mathcal{B}$. The set $\mathcal{S}$ of *simple types* (with product types) is generated from $\mathcal{B}$ by type constructors $\rightarrow$ and $\times$, that is, $\mathcal{S} ::= \mathcal{B} \mid (\mathcal{S}_1 \rightarrow \mathcal{S}_2) \mid (\mathcal{S}_1 \times \cdots \times \mathcal{S}_n)$. To minimize the number of parentheses, we assume that $\rightarrow$ is right-associative and $\rightarrow$ has lower precedence than $\times$. A *product type* is a simple type of the form $\alpha_1 \times \cdots \times \alpha_n$. A *functional type* or a *higher-order type* is a simple type of the form $\alpha \rightarrow \beta$. We denote the set of functional types by $\mathcal{S}_{fun}$, and the set of non-functional types by $\mathcal{S}_{nfun}$. A simple type α is said to be a *suffix* of a simple type β, denoted by $\beta \sqsupseteq_{\mathcal{S}} \alpha$, if β has the form $\alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow \alpha$.

A *typing function* τ is a function from $\mathcal{V} \cup (\Sigma \setminus \{\mathtt{tp}\})$ to $\mathcal{S}$. We assume that for any $\alpha \in \mathcal{S}$ there exists a variable $x \in \mathcal{V}$ such that $\tau(x) = \alpha$. We also assume that Σ contains a special constructor $\mathtt{tp}$, called a *tuple*. We write $(t_1, \ldots, t_n)$ instead of $\mathtt{tp}[t_1, \ldots, t_n]$. Each typing function τ is naturally extended to terms as follows: for any $t \equiv a[t_1, \ldots, t_n] \in \mathcal{T}(\Sigma, \mathcal{V})$, if $\tau(t_i) = \alpha_i$ $(i = 1, \ldots, n)$ and either $\tau(a) = \alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow \alpha$ or $a = \mathtt{tp} \wedge \alpha = \alpha_1 \times \cdots \times \alpha_n$, then $\tau(t) = \alpha$. A term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is said to be *simply-typed* if t has a simple type, that is, $\tau(t)$ is defined. A term t, which has a simple type α, is often denoted by t^{α}. We denote the set of all simply-typed terms by $\mathcal{T}_{\tau}(\Sigma, \mathcal{V})$. We also denote the set of functional (resp. non-functional) typed terms by $\mathcal{T}_{fun}(\Sigma, \mathcal{V})$ (resp. $\mathcal{T}_{nfun}(\Sigma, \mathcal{V})$). We use $\mathcal{V}_{fun}$ to stand for the set of functionally typed variables (higher-order variables), and $\mathcal{V}_{nfun}$ to stand for the set $\mathcal{V} \setminus \mathcal{V}_{fun}$. Now we restrict substitutions to type preserving substitutions. We also index the hole $\Box_{\alpha}$ with every simple type α, and assume that $\tau(t) = \alpha$ whenever we denote $C[t]$ for each context $C[\,]$ with a hole $\Box_{\alpha}$. In the following, a simply-typed term is often shortly denoted by a term.

A *simply-typed rule* is a pair (l, r) of simply-typed terms, denoted by $l \rightarrow r$, such that $root(l) \in \Sigma \setminus \{\mathtt{tp}\}$, $Var(l) \supseteq Var(r)$ and $\tau(l) = \tau(r)$. A *simply-typed term rewriting system* (STRS) is an abstract reduction system $\langle \mathcal{T}_{\tau}(\Sigma, \mathcal{V}), \underset{R}{\rightarrow} \rangle$. We often denote an STRS $\langle \mathcal{T}_{\tau}(\Sigma, \mathcal{V}), \underset{R}{\rightarrow} \rangle$ by R. For each STRS R, we define $\mathcal{T}_{SN}(R) = \{t \mid SN(R, t)\}$, $\mathcal{T}_{\neg SN}(R) = \mathcal{T}_{\tau}(\Sigma, \mathcal{V}) \setminus \mathcal{T}_{SN}(R)$, and $\mathcal{T}_{SN}^{args}(R) = \{t \mid \forall u \in args(t).SN(R, u)\}$.

Let R be an STRS and $l \rightarrow r \in R$ such that $\tau(l) = \alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow \alpha$ and $\alpha \in \mathcal{S}_{nfun}$. The set $(l \rightarrow r)^{ex}$ of the *expansion forms* of a rule $l \rightarrow r$ is defined as $\{l \rightarrow r, l[z_1] \rightarrow r[z_1], \ldots, l[z_1, \ldots, z_n] \rightarrow r[z_1, \ldots, z_n]\}$, where $z_1^{\alpha_1}, \ldots, z_n^{\alpha_n}$ are fresh variables. We also define $R^{ex} = \bigcup_{l \rightarrow r \in R} (l \rightarrow r)^{ex}$. The rule $(l \rightarrow r)^{ex\uparrow}$ of the *full expansion form* of $l \rightarrow r$ is defined as $l[z_1, \ldots, z_n] \rightarrow r[z_1, \ldots, z_n]$, where $z_1^{\alpha_1}, \ldots, z_n^{\alpha_n}$ are fresh variables. We also define $R^{ex\uparrow} = \{(l \rightarrow r)^{ex\uparrow} \mid l \rightarrow r \in R\}$.

Proposition 2.1 Let R be an STRS. If $s \underset{R}{\rightarrow} t$ then there exist a rule $l \rightarrow r \in R^{ex}$, a leaf-context $C[\,]$, and a substitution θ such that $s \equiv C[l\theta]$ and $t \equiv C[r\theta]$.

A term t is said to be *finite branching* in an STRS R if $\{t' \mid t \underset{R}{\rightarrow} t'\}$ is finite. An STRS R is said to be *finite branching* if any term is finite branching in R.

A well-founded strict order $>$ on terms is said to be a *reduction order* (resp. *semi-reduction order*) if $>$ is closed under substitutions and contexts (resp. leaf-contexts). We note that STRS R is terminating iff $R \subseteq >$ for some reduction order $>$, and iff $R^{ex} \subseteq >$ for some semi-reduction order $>$.

All root symbols of the left-hand sides of rules in an STRS R, denoted by $\mathcal{D}_R$, are called *defined*, whereas all other function symbols, denoted by C_R, are called *constructors*.

3. Static Dependency Pair Method

We proposed the static dependency pair method, which can effectively prove termination of STRSs [22]. This method analyzes a static recursive structure based on definition dependency, in contrast to dynamic dependency pair methods that analyze a dynamic recursive structure based on function-call dependency through higher-order variables [18], [24]. Hence, static dependency pair methods have a more practical advantage than dynamic ones. The key idea of the static dependency pair method is that a static recursive structure can be formulated as a recursive structure from the viewpoint of strong computability, which was introduced for proving termination in typed λ-calculus [12], [27]. As described in the Introduction, static dependency pair methods are not applicable to every STRS. Hence, we proposed the notion of plain function-passing [22]. Roughly speaking, plain function-passing means that every higher-order variable occurs in an argument position on the left-hand side.

From a technical viewpoint, we have noticed that the unclosedness of strong computability with respect to the subterm relation is the reason why the static dependency pair method is not applicable to every STRS. Accordingly, we introduce the notion of a peeling order and reconstruct the strong computability by using this peeling order. Then we can peel a strongly computable term such that peeled subterms are strongly computable. As a result, we introduce the notion of safe function-passing which expands the application range of the static dependency pair method. Thus, we can apply the static dependency pair method to $R_{\mathtt{app}_{\emptyset}}$ displayed in the Introduction. Since we change the definition of strong computability, which gives a theoretical basis for the static dependency pair method, we prove the soundness of the static dependency pair method under this new framework.

3.1 Safe Function-Passing

We introduce the notion of a peeling order, and by using this notion we introduce the notion of safe function-passing

under which the static dependency pair method works well.

Definition 3.1 (Peeling Order) A well-founded quasi order $\gtrsim_{\mathcal{S}}$ on types is said to be a *peeling order* if $\alpha \to \beta \gtrsim_{\mathcal{S}} \alpha$ and $\alpha \to \beta \gtrsim_{\mathcal{S}} \beta$ hold.

For any peeling order $\gtrsim_{\mathcal{S}}$, term t and set A of types, we define $Sub_A^{\gtrsim_{\mathcal{S}}}(t)$ as the smallest set satisfying the following properties:

- $args(t) \subseteq Sub_A^{\gtrsim_{\mathcal{S}}}(t)$
- if $u \equiv a[u_1, \ldots, u_n] \in Sub_A^{\gtrsim_{\mathcal{S}}}(t)$, $a \in C_R$, $\tau(u) \in A$ and $u \gtrsim_{\mathcal{S}} u_i$ then $u_i \in Sub_A^{\gtrsim_{\mathcal{S}}}(t)$

Example 3.2 Let $R_{\mathtt{app}_0}$ be the STRS defined as follows:

$$\begin{cases} \mathtt{app}_0[\mathtt{nilF}] & \to & \mathtt{nil} \\ \mathtt{app}_0[\mathtt{consF}[f, fs]] & \to & \mathtt{cons}[f[0], \mathtt{app}_0[fs]] \end{cases}$$

where $\tau(\mathtt{app}_0) = L_{N\to N} \to L_N$, $\tau(\mathtt{nil}) = L_N$, $\tau(\mathtt{nilF}) = L_{N\to N}$, $\tau(\mathtt{cons}) = N \to L_N \to L_N$, $\tau(\mathtt{consF}) = (N \to N) \to L_{N\to N} \to L_{N\to N}$, and so on. Since simple types can be interpreted as first-order terms, we present an order $\gtrsim_{\mathcal{S}}$ on simple-types by the recursive path order with the precedence $L_{N\to N} \rhd \to$ and $L_{N\to N} \rhd N$ [5]. Then $\gtrsim_{\mathcal{S}}$ is a peeling order. For $A = \{L_{N\to N}\}$, we have $Sub_A^{\gtrsim_{\mathcal{S}}}(\mathtt{app}_0[\mathtt{consF}[f, fs]]) = \{\mathtt{consF}[f, fs], f, fs\}$.

Definition 3.3 (Safe Function-Passing) An STRS R is said to be *safe function-passing* with respect to a peeling order $\gtrsim_{\mathcal{S}}$ if there exists a set PT of non-functional types such that for any $l \to r \in R$ and $v \in Sub(r)$, the following properties hold:

- if $root(v) \in \mathcal{V}_{fun}$ then there exists $u \in Sub_{PT}^{\gtrsim_{\mathcal{S}}}(l)$ such that $u \sqsubseteq v$, and
- if $v \in \mathcal{V}_{nfun}$ and $\tau(v) \in PT$ then $v \in Sub_{PT}^{\gtrsim_{\mathcal{S}}}(l)$.

The set PT is said to be *peeling types*, and a safe function-passing STRS is often shortly denoted by SFP-STRS.

Example 3.4 Consider the STRS $R_{\mathtt{app}_0}$ given in Example 3.2. Take PT as the set A in Example 3.2. Then $R_{\mathtt{app}_0}$ is safe function-passing because we have $f, fs \in \{\mathtt{consF}[f, fs], f, fs\} = Sub_{PT}^{\gtrsim_{\mathcal{S}}}(\mathtt{app}_0[\mathtt{consF}[f, fs]])$.

We note that plain function-passing [22] corresponds to safe function-passing if $PT = \{\alpha \mid \alpha \text{ is a product type}, \alpha \neq \tau(z) \text{ for all } l \to r \in R \text{ and } z \in Var(r)\}$ and $\gtrsim_{\mathcal{S}}$ is defined as the subtype relation.

3.2 Strong Computability

In this subsection, we build peeling order/types into the strong computability, which gives a theoretical basis for the static dependency pair method.

Definition 3.5 (Strong Computability) Let R be an SFP-STRS with a peeling order $\gtrsim_{\mathcal{S}}$ and peeling types PT. A term t is said to be *strongly computable* in R, if $SC(R, t)$ holds, which is defined as follows:

- in case of $\tau(t) \in \mathcal{S}_{nfun} \setminus PT$, $SC(R, t)$ is defined as $SN(R, t)$,
- in case of $\tau(t) \in PT$, $SC(R, t)$ is defined as $SN(R, t)$ and $SC(R, u)$ for any $u \in \bigcup\{args(t') \mid t \xrightarrow[R]{*} t', root(t') \in C_R\}$ such that $\tau(t) \gtrsim_{\mathcal{S}} \tau(u)$.
- in case of $\tau(t) = \alpha \to \beta$, $SC(R, t)$ is defined as $SC(R, u) \Rightarrow SC(R, t[u])$ for any u^α.

For each SFP-STRS R, we define $\mathcal{T}_{SC}(R) = \{t \mid SC(R, t)\}$, $\mathcal{T}_{\neg SC}(R) = \mathcal{T}_\tau(\Sigma, \mathcal{V}) \setminus \mathcal{T}_{SC}(R)$, and $\mathcal{T}_{SC}^{args}(R) = \{t \mid \forall u \in args(t).SC(R, u)\}$.

Theorem 3.6 The predicate SC is well-defined for SFP-STRSs.

Proof. Let R be an SFP-STRS with $\gtrsim_{\mathcal{S}}$ and PT. Assume that SC is not well-defined.

Let t_0 be a minimal term with respect to $\gtrsim_{\mathcal{S}}$ such that $SC(R, t_0)$ is not well-defined, that is, $SC(R, t)$ is well-defined for any t with $\tau(t_0) \gtrsim_{\mathcal{S}} \tau(t)$. From the minimality of t_0, $\tau(t_0) \in PT$, $SN(R, t_0)$, and there exist t_0' and t_1 such that $t_0 \xrightarrow[R]{*} t_0'$, $t_1 \in args(t_0')$, $\tau(t_0) \sim_{\mathcal{S}} \tau(t_1)$, and $SC(R, t_1)$ is not well-defined, where $\sim_{\mathcal{S}}$ is the equivalence part of $\gtrsim_{\mathcal{S}}$.

Since $\tau(t_0) \sim_{\mathcal{S}} \tau(t_1)$, t_1 is also a minimal term with respect to $\gtrsim_{\mathcal{S}}$ such that $SC(R, t_1)$ is not well-defined. By applying the procedure above, we obtain t_1' and t_2 such that $t_1 \xrightarrow[R]{*} t_1'$, $t_2 \in args(t_1')$, $\tau(t_1) \sim_{\mathcal{S}} \tau(t_2)$, and $SC(R, t_2)$ is not well-defined.

By applying this procedure repeatedly, we obtain $t_2', t_3', \ldots$ and $t_3, t_4, \ldots$ such that $t_i \xrightarrow[R]{*} t_i'$ and $t_{i+1} \in args(t_i')$ for $i = 2, 3, \ldots$. Since $>_{sub} \cup \xrightarrow[R]{}$ is well-founded on terminating terms, this contradicts with $SN(R, t_0)$. □

We now present the basic properties of strong computability.

Lemma 3.7 For any SFP-STRS R, the following properties hold:

(1) For any strongly computable terms $t^{\alpha_1\to\cdots\to\alpha_n\to\alpha}$ and $u_i^{\alpha_i}$ $(i = 1, \ldots, n)$, we have $SC(R, t[u_1, \ldots, u_n])$.
(2) For any non-strongly computable term $t^{\alpha_1\to\cdots\to\alpha_n\to\alpha}$, there exist strongly computable terms $u_i^{\alpha_i}$ $(i = 1, \ldots, n)$ such that $\neg SC(R, t[u_1, \ldots, u_n])$.
(3) $SC(R, t) \wedge t \xrightarrow[R]{*} t' \Rightarrow SC(R, t')$ for all t and t'.
(4) Any variable z^α is strongly computable, for all $\alpha \in \mathcal{S}$.
(5) $SC(R, t^\alpha) \Rightarrow SN(R, t^\alpha)$, for all $\alpha \in \mathcal{S}$.

Proof. The properties (1) and (2) are easily shown by induction on n.

(3) We prove the claim by induction on $\tau(t)$. The case $\tau(t) \in \mathcal{S}_{nfun}$ is trivial. Suppose that $\tau(t) = \tau(t') = \alpha \to \beta$. Let u^α be an arbitrary strongly computable term. Then $SC(R, t[u])$ follows from $SC(R, t)$. Since $t[u] \xrightarrow[R]{*} t'[u]$

and $\tau(t[u]) = \beta$, $SC(R, t'[u])$ follows from the induction hypothesis. Hence, $SC(R, t')$ holds.

(4, 5) We prove claims by simultaneous induction on α. The case $\alpha \in \mathcal{S}_{nfun}$ is trivial. Suppose that $\alpha = \alpha_1 \to \cdots \to \alpha_n \to \beta$ and $\beta \in \mathcal{S}_{nfun}$.

(4): Assume that z is not strongly computable for some $z \in \mathcal{V}_\alpha$. From (2), there exist strongly computable terms $u_1^{\alpha_1}, \ldots, u_n^{\alpha_n}$ such that $z[u_1, \ldots, u_n]$ is not strongly computable. From the induction hypothesis (5), each u_i is terminating, hence so is $z[u_1, \ldots, u_n]$. Since $z[u_1, \ldots, u_n]$ is not strongly computable and $\beta \in \mathcal{S}_{nfun}$, we have $\beta \in PT$ and there exist terms u' and u such that $z[u_1, \ldots, u_n] \xrightarrow[R]{*} u'$, $u \in args(u')$, and u is not strongly computable. Since $root(l) \notin \mathcal{V}$ for all $l \to r \in R$, there exists i such that $u_i \xrightarrow[R]{*} u$. From (3), u_i is not strongly computable. This is a contradiction.

(5): From the induction hypothesis (4), an arbitrary variable $z_1^{\alpha_1}$ is strongly computable. Thus, $t[z_1]$ is strongly computable. From the induction hypothesis (5), $t[z_1]$ is terminating, hence so is t. □

We previously mentioned that we can peel a strongly computable term such that peeled subterms are strongly computable. In the proof of the soundness of the static dependency pair method, this mention is formulated as the following lemma.

Lemma 3.8 Let R be an SFP-STRS, $l \to r \in R$, and θ be a substitution such that $l\theta \in \mathcal{T}_{SC}^{args}(R)$. Then $SC(R, u\theta)$ holds for any $u \in Sub_{PT}^{\gtrsim_S}(l)$.

Proof. Since $u \in Sub_{PT}^{\gtrsim_S}(l)$, we have either $u \in args(l)$ or there exists $u' \equiv a[\ldots, u, \ldots] \in Sub_{PT}^{\gtrsim_S}(l)$ such that $a \in C_R$, $\tau(u') \in PT$ and $\tau(u') \gtrsim_S \tau(u)$. In the former case, we have $SC(R, u\theta)$ because of $l\theta \in \mathcal{T}_{SC}^{args}(R)$. In the latter case, it suffices to show that $SC(R, u\theta)$ whenever $SC(R, u'\theta)$, which is directly deduced from the definition of strong computability. □

3.3 Static Dependency Pair Method

We present a static dependency pair method for SFP-STRSs. Since we modified the definition of strong computability, which gives a theoretical basis for the static dependency pair method, we prove the soundness of the static dependency pair method under this new framework.

Definition 3.9 For each $f \in \mathcal{D}_R$, we provide a new function symbol $f^\sharp$, called the *marked-symbol* of f. For each $t \equiv a[t_1, \ldots, t_n]$, we define the *marked term* $t^\sharp$ by $a^\sharp[t_1, \ldots, t_n]$ if $a \in \mathcal{D}_R$; otherwise $t^\sharp \equiv t$.

Let R be an SFP-STRS. For each $l \to r \in R^{ex\uparrow}$ and $a[r_1, \ldots, r_m] \in Sub(r)$ such that

- $a \in \mathcal{D}_R$,
- there exists no $u \in Sub_{PT}^{\gtrsim_S}(l)$ such that $u \sqsubseteq a[r_1, \ldots, r_m]$, and
- there exists no $u \in Sub(l)\backslash\{l\}$ such that $u \equiv a[r_1, \ldots, r_m]$ and $\tau(u) \in \mathcal{S}_{nfun} \setminus PT$,

we define a *static dependency pair* of R as a pair $\langle l^\sharp, a^\sharp[r_1, \ldots, r_m, z_1, \ldots, z_n]\rangle$, denoted by

$$l^\sharp \to a^\sharp[r_1, \ldots, r_m, z_1, \ldots, z_n],$$

where $\tau(a^\sharp[r_1, \ldots, r_m, z_1, \ldots, z_n]) \in \mathcal{S}_{nfun}$ and $z_1, \ldots, z_n$ are fresh variables. We denote by $SDP(R)$ the set of all static dependency pairs of R.

Example 3.10 Let $R_{\mathtt{sumF}}$ be the following STRS:

$$R_{\mathtt{sumF}} = R_{\mathtt{sum}} \cup R_{\mathtt{app}_\mathtt{0}} \cup \{\mathtt{sumF}[fs] \to \mathtt{sum}[\mathtt{app}_\mathtt{0}[fs]]\}$$

where $R_{\mathtt{sum}}$ and $R_{\mathtt{app}_\mathtt{0}}$ are displayed in the Introduction. Since $R_{\mathtt{sum}} \cup R_{\mathtt{app}_\mathtt{0}}$ is safe function-passing (cf. Example 3.4) and $fs \in args(\mathtt{sumF}[fs])$, then STRS $R_{\mathtt{sumF}}$ is safe function-passing. Thus, the set $SDP(R_{\mathtt{sumF}})$ consists of the following seven static dependency pairs:

$$\begin{cases}
\mathtt{foldl}^\sharp[f, y, \mathtt{cons}[x, xs]] \to \mathtt{foldl}^\sharp[f, f[y, x], xs] \\
\mathtt{add}^\sharp[\mathtt{s}[x], y] \to \mathtt{add}^\sharp[x, y] \\
\mathtt{sum}^\sharp[z] \to \mathtt{foldl}^\sharp[\mathtt{add}, 0, z] \\
\mathtt{sum}^\sharp[z] \to \mathtt{add}^\sharp[z'] \\
\mathtt{app}_\mathtt{0}{}^\sharp[\mathtt{consF}[f, fs]] \to \mathtt{app}_\mathtt{0}{}^\sharp[fs] \\
\mathtt{sumF}^\sharp[fs] \to \mathtt{sum}^\sharp[\mathtt{app}_\mathtt{0}[fs]] \\
\mathtt{sumF}^\sharp[fs] \to \mathtt{app}_\mathtt{0}{}^\sharp[fs]
\end{cases}$$

Definition 3.11 Let R be an SFP-STRS. A (possibly infinite) sequence $u_1^\sharp \to v_1^\sharp, \ldots, u_n^\sharp \to v_n^\sharp$ of static dependency pairs of R is said to be a *static dependency chain* of R if there exist $\theta_1, \ldots, \theta_n$ such that $u_i^\sharp\theta_i, v_i^\sharp\theta_i \in \mathcal{T}_{SC}^{args}(R)$ and $v_i^\sharp\theta_i \xrightarrow[R]{*} u_{i+1}^\sharp\theta_{i+1}$ for any i. A *static dependency graph* of R is a directed graph, in which nodes are $SDP(R)$ and there exists an arc from $u^\sharp \to v^\sharp$ to $u'^\sharp \to v'^\sharp$ if $u^\sharp \to v^\sharp, u'^\sharp \to v'^\sharp$ is a static dependency chain.

Definition 3.12 A *(maximal) static recursion component* of R is a set of nodes in a (maximal) strongly connected subgraph of a static dependency graph. We denote by $SRC(R)$ the set of all static recursion components of R.

A static recursion component $C \in SRC(R)$ is said to be *non-looping* if there exists no infinite static dependency chain $u_0^\sharp \to v_0^\sharp, u_1^\sharp \to v_1^\sharp, \cdots$ such that $u_i^\sharp \to v_i^\sharp \in C$ for all i and every $u^\sharp \to v^\sharp \in C$ occurs infinitely many times.

Example 3.13 Referencing to Example 3.10. The set $SRC(R_{\mathtt{sumF}})$ consists of the following three static recursion components:

$$\{\mathtt{foldl}^\sharp[f, y, \mathtt{cons}[x, xs]] \to \mathtt{foldl}^\sharp[f, f[y, x], xs]\}$$
$$\{\mathtt{add}^\sharp[\mathtt{s}[x], y] \to \mathtt{add}^\sharp[x, y]\}$$
$$\{\mathtt{app}_\mathtt{0}{}^\sharp[\mathtt{consF}[f, fs]] \to \mathtt{app}_\mathtt{0}{}^\sharp[fs]\}$$

In the remainder of this subsection, we show the soundness of the static dependency pair method on SFP-STRSs. That is, we show that if any static recursion component of SFP-STRS R are non-looping, then R is terminating. We need prepare two key lemmas.

Lemma 3.14 If an SFP-STRS R is not terminating then $\mathcal{T}_{nfun}(\Sigma,\mathcal{V}) \cap \mathcal{T}_{\neg sc}(R) \cap \mathcal{T}_{sc}^{args}(R) \neq \emptyset$.

Proof. Since R is not terminating, $\mathcal{T}_{\neg sc}(R) \neq \emptyset$ follows from Lemma 3.7 (5).

Let s be a minimal term in $\mathcal{T}_{\neg sc}(R)$ with respect to term size. Then $s \in \mathcal{T}_{sc}^{args}(R)$ holds because the strong computability of each $s' \in args(s)$ follows from the minimality of s. Hence, we have $\mathcal{T}_{\neg sc}(R) \cap \mathcal{T}_{sc}^{args}(R) \neq \emptyset$.

Let t be a minimal term in $\mathcal{T}_{\neg sc}(R) \cap \mathcal{T}_{sc}^{args}(R)$ with respect to type size. It suffices to show that $t \in \mathcal{T}_{nfun}(\Sigma,\mathcal{V})$. Assume that $t \notin \mathcal{T}_{nfun}(\Sigma,\mathcal{V})$. Let $\tau(t) = \alpha \to \beta$ and u^{α} be an arbitrary strongly computable term. Since $t \in \mathcal{T}_{sc}^{args}(R)$ and $u \in \mathcal{T}_{sc}(R)$, we have $t[u] \in \mathcal{T}_{sc}^{args}(R)$. From $\tau(t[u]) = \beta$ and the minimality of $\tau(t) = \alpha \to \beta$, we have $t[u] \notin \mathcal{T}_{\neg sc}(R) \cap \mathcal{T}_{sc}^{args}(R)$. Hence, $t[u] \in \mathcal{T}_{sc}(R)$. Then we have $t \in \mathcal{T}_{sc}(R)$, which is a contradiction. □

Lemma 3.15 Let R be an SFP-STRS. For any $t \in \mathcal{T}_{nfun}(\Sigma,\mathcal{V}) \cap \mathcal{T}_{\neg sc}(R) \cap \mathcal{T}_{sc}^{args}(R)$, there exist $l^{\sharp} \to v^{\sharp} \in SDP(R)$ and θ such that $t^{\sharp} \xrightarrow[R]{*} l^{\sharp}\theta$ and $l\theta, v\theta \in \mathcal{T}_{nfun}(\Sigma,\mathcal{V}) \cap \mathcal{T}_{\neg sc}(R) \cap \mathcal{T}_{sc}^{args}(R)$.

Proof. Let $t \in \mathcal{T}_{nfun}(\Sigma,\mathcal{V}) \cap \mathcal{T}_{\neg sc}(R) \cap \mathcal{T}_{sc}^{args}(R)$. Then $t \in \mathcal{T}_{sn}^{args}(R)$ follows from $t \in \mathcal{T}_{sc}^{args}(R)$ and Lemma 3.7 (5).

- Consider the case that $t \notin \mathcal{T}_{sn}(R)$. Since $t \in \mathcal{T}_{sn}^{args}(R) \cap \mathcal{T}_{nfun}(R)$, there exist $l \to r \in R^{ex\uparrow}$ and θ' such that $t^{\sharp} \xrightarrow{*} l^{\sharp}\theta'$, $\neg SN(R, l\theta')$ and $\neg SN(R, r\theta')$. Hence, $\neg SC(R, l\theta')$ and $\neg SC(R, r\theta')$ follow from Lemma 3.7 (5).
- Consider the case that $t \in \mathcal{T}_{sn}(R)$. Since $t \in \mathcal{T}_{\neg sc}(R) \cap \mathcal{T}_{nfun}(R)$, we have $\tau(t) \in PT$ and there exist terms t' and $t'' \in args(t')$ such that $t \xrightarrow{*} t'$, $root(t') \in C_R$, $\tau(t) \gtrsim_S \tau(t'')$, and $t'' \in \mathcal{T}_{\neg sc}(R)$. Assume that $root(t) \in C_R$. Then $SC(R,t)$ follows from $t \in \mathcal{T}_{sn}(R) \cap \mathcal{T}_{sc}^{args}(R)$, $root(t) \in C_R$, and Lemma 3.7 (3). This is a contradiction. Hence, $root(t) \notin C_R$. Thus, there exist $l \to r \in R^{ex\uparrow}$ and θ' such that $t^{\sharp} \xrightarrow{*} l^{\sharp}\theta'$ and $l\theta' \to r\theta' \xrightarrow{*} t'$. Since t'' is not strongly computable, so is t'. From Lemma 3.7 (3), we have $\neg SC(R, l\theta')$ and $\neg SC(R, r\theta')$.

In both cases above, we have $\{v' \in Sub(r) \mid \neg SC(R, v'\theta')\} \neq \emptyset$ because $r \in Sub(r)$ and $\neg SC(R, r\theta')$. Let $v' \equiv a[r_1, \ldots, r_m]$ be a minimal size term in this set. Then $SC(R, r_i\theta')$ holds for every i. From Lemma 3.7 (2), there exist $v_1, \ldots, v_k \in \mathcal{T}_{sc}(R)$ such that $\tau(v'\theta'[v_1, \ldots, v_k]) \in \mathcal{S}_{nfun}$ and $\neg SC(v'\theta'[v_1, \ldots, v_k])$. Here $v'\theta'[v_1, \ldots, v_k] \in \mathcal{T}_{sc}^{args}(R)$.

Now take v by $a[r_1, \ldots, r_m, z_1, \ldots, z_k]$ where $z_1, \ldots, z_k$ are fresh variables, and $\theta(x)$ is defined by v_i if $x = z_i$ $(i = 1, \ldots, k)$; otherwise by $\theta'(x)$. Then we have $l\theta = l\theta'$ and $v\theta = v'\theta'[v_1, \ldots, v_k]$. Since $l\theta \in \mathcal{T}_{sc}^{args}(R)$ follows from $t \in \mathcal{T}_{sc}^{args}(R)$ and Lemma 3.7 (3), we have $l\theta \in \mathcal{T}_{nfun}(\Sigma,\mathcal{V}) \cap \mathcal{T}_{\neg sc}(R) \cap \mathcal{T}_{sc}^{args}(R)$. Because $v\theta \in \mathcal{T}_{nfun}(\Sigma,\mathcal{V}) \cap \mathcal{T}_{\neg sc}(R) \cap \mathcal{T}_{sc}^{args}(R)$ also holds, it suffices to show that $l^{\sharp} \to v^{\sharp} \in SDP(R)$. We prove this by contradiction. Assume that $l^{\sharp} \to v^{\sharp} \notin SDP(R)$. Let $l \equiv l'[z'_1, \ldots, z'_p]$ and $r \equiv r'[z'_1, \ldots, z'_p]$ such that $l' \to r' \in R$ and $z'_1, \ldots, z'_p$ are fresh variables.

- Assume that $a \in \mathcal{V}_{nfun}$ and $\tau(v) \notin PT$. Since $v\theta \equiv a\theta \in Sub(l\theta)$ and $l\theta \in \mathcal{T}_{sn}^{args}(R)$, $SN(R, v\theta)$ holds, and hence $SC(R, v\theta)$ also holds. This is a contradiction.
- Assume that either $a \in \mathcal{V}_{fun}$ or $a \in \mathcal{V}_{nfun}$ and $\tau(v) \in PT$. Since R is safe function-passing, $SC(R, v\theta)$ follows from Lemma 3.8, $v\theta \in \mathcal{T}_{sc}^{args}(R)$, and Lemma 3.7 (1). This is a contradiction.
- Assume that $a \in C_R$. Since $v\theta \in \mathcal{T}_{sn}^{args}(R)$ from Lemma 3.7 (5), $v\theta$ is terminating. Since $v\theta \in \mathcal{T}_{nfun}(\Sigma,\mathcal{V}) \cap \mathcal{T}_{\neg sc}(R)$, we have $\tau(v\theta) \in PT$ and there exist terms u' and $u'' \in args(u')$ such that $v\theta \xrightarrow{*} u'$, $root(u') \in C_R$, $\tau(v\theta) \gtrsim_S \tau(u'')$ and $u'' \in \mathcal{T}_{\neg sc}(R)$. Since $root(v\theta) = a \in C_R$ and $v\theta \in \mathcal{T}_{sc}^{args}(R)$, $u' \in \mathcal{T}_{sc}^{args}(R)$ follows from Lemma 3.7 (3). This is a contradiction.
- Assume that $a \in \mathcal{D}_R$ and there exists $u \in Sub_{PT}^{\gtrsim_S}(l)$ such that $u \sqsubseteq a[r_1, \ldots, r_m]$. From Lemma 3.8, we have $SC(R, u\theta)$. From $v\theta \in \mathcal{T}_{sc}^{args}(R)$ and Lemma 3.7 (1), we have $SC(R, v\theta)$. This is a contradiction.
- Assume that $a \in \mathcal{D}_R$ and there exists $u \in Sub(l) \setminus \{l\}$ such that $u \equiv a[r_1, \ldots, r_m]$ and $\tau(u) \in \mathcal{S}_{nfun} \setminus PT$. Then $u \equiv v$ follows from $\tau(u) \in \mathcal{S}_{nfun}$. Since $l\theta \in \mathcal{T}_{sc}^{args}(R)$, $l\theta \in \mathcal{T}_{sn}^{args}(R)$ follows from Lemma 3.7 (5), and hence $v\theta$ is terminating. Since $\tau(v\theta) \in \mathcal{S}_{nfun} \setminus PT$, $v\theta$ is strongly computable. This is a contradiction. □

We obtain the fundamental theorem of the static dependency pair method.

Theorem 3.16 Let R be an SFP-STRS. If there exists no infinite static dependency chain then R is terminating.

Proof. Assume that $\neg SN(R)$. From Lemma 3.14, there exists $t \in \mathcal{T}_{nfun} \cap \mathcal{T}_{\neg sc}(R) \cap \mathcal{T}_{sc}^{args}(R)$. By applying Lemma 3.15 repeatedly, we have an infinite static dependency chain, which leads to a contradiction. □

Note that the inverse of the theorem does not hold. For example, let $R_{\mathtt{fix}}$ be the SFP-STRS $\{\mathtt{fix}[f,x] \to f[\mathtt{fix}[f], x]\}$. Although $R_{\mathtt{fix}}$ is terminating, the infinite sequence composed of the static dependency pair $fix^{\sharp}[f,x] \to fix^{\sharp}[f,z]$ is an infinite static dependency chain. Hence, the static dependency pair method has a theoretical limitation for the completeness.

Corollary 3.17 Let R be an SFP-STRS such that there exists no infinite path[†] in the static dependency graph. If all recursion components in $SRC(R)$ are non-looping then R is terminating.

[†]Each node cannot appear twice in a path.

3.4 Non-loopingness of Recursion Components

In this subsection, we present a powerful and efficient method for proving termination by using notions of (semi-)reduction pairs and the subterm criterion, which prove that recursion components do not loop.

First, we introduce the notion of (semi-)reduction pairs according to the literature [22]. The notion of reduction pairs was introduced in [17], which is a slight abstraction of weak-reduction order [1]. The notion of semi-reduction pairs was introduced in [18].

Definition 3.18 For a predicate P, a relation Υ is P-closed under substitutions if $s\theta\Upsilon t\theta$ for any substitution θ and terms s, t such that $P(s,t)$ holds.

A pair $(\gtrsim, >)$ of a quasi-order $\gtrsim$ and a well-founded strict order $>$ is said to be a *semi-reduction pair* w.r.t. a predicate P if $\gtrsim$ is closed under leaf-contexts, $\gtrsim$ and $>$ are P-closed under substitutions, and either $\gtrsim \cdot > \subseteq >$ or $> \cdot \gtrsim \subseteq >$. A semi-reduction pair $(\gtrsim, >)$ w.r.t. a predicate P is said to be a *reduction pair* w.r.t. P if $\gtrsim$ is closed under contexts.

Proposition 3.19 Let R be an STRS and C be a static recursion component. If there exists a reduction pair (resp. semi-reduction pair) $(\gtrsim, >)$ w.r.t. a predicate P satisfying the following conditions, then C is non-looping.

- $P(s,t)$ holds for any $(s,t) \in R \cup C$ (resp. $(s,t) \in R^{ex} \cup C$),
- $R \subseteq \gtrsim$ (resp. $R^{ex} \subseteq \gtrsim$), and
- $C \subseteq \gtrsim \cup >$ and $C \cap > \neq \emptyset$.

The argument filtering method, which generates a reduction pair from a given reduction order, was introduced in first-order TRSs [1]. The method was extended to STRSs [18] and will be improved in the next section. In both the methods in STRSs, as a predicate P in the definition above, we need to use left-firmness (cf. Definition 4.3).

Although the path order based on strong computability in [19] generates reduction pairs, the path order based on the simplification order in [18] does not generate reduction pairs and only generates semi-reduction pairs.

We next introduce the subterm criterion [22] and the strictly subterm criterion, which are slight improvements of the criterion in [15]. Although the original definition of the codomain of π (see the following definition) in [15] allows only positive integers, the improved definition allows sequences of positive integers [22].

Definition 3.20 ((Strictly) Subterm Criterion) Let R be an SFP-STRS and $C \in SRC(R)$. We say that C satisfies the *subterm criterion* if there exists a function π from $\mathcal{D}_R$ to non-empty sequences of positive integers such that

- $u|_{\pi(root(u))} >_{esub} v|_{\pi(root(v))}$ for some $u^\sharp \to v^\sharp \in C$, and
- the following conditions hold for any $u^\sharp \to v^\sharp \in C$:
  - $u|_{\pi(root(u))} \geq_{esub} v|_{\pi(root(v))}$,
  - $(u)_p \notin \mathcal{V}$ for all $p \prec \pi(root(u))$, and
  - $q \neq \varepsilon \Rightarrow (v)_q \in C_R$ for all $q \prec \pi(root(v))$.

Specially, we say that C satisfies the *strictly subterm criterion* if any $u^\sharp \to v^\sharp \in C$ satisfies the following condition:

- $u|_{\pi(root(u))} >_{esub} v|_{\pi(root(v))}$,
- $(u)_p \notin \mathcal{V}$ for all $p \prec \pi(root(u))$, and
- $q \neq \varepsilon \Rightarrow (v)_q \in C_R$ for all $q \prec \pi(root(v))$.

We can easily see that if C satisfies the strictly subterm criterion, then any subset of C satisfies the subterm criterion.

Proposition 3.21 Let R be an STRS and C be a static recursion component. If C satisfies the subterm criterion, then C is non-looping.

From Corollary 3.17, and Proposition 3.19 and 3.21, we obtain the following method for proving termination of SFP-STRSs.

Theorem 3.22 Let R be an SFP-STRS such that there exists no infinite path in the static dependency graph. If each $C \in SRC(R)$ satisfies one of the following properties, then R is terminating.

(1) C satisfies the subterm criterion.
(2) There exists a reduction pair (resp. semi-reduction pair) $(\gtrsim, >)$ w.r.t. a predicate P such that $P(s,t)$ holds for any $(s,t) \in R \cup C$ (resp. $(s,t) \in R^{ex} \cup C$), $R \subseteq \gtrsim$ (resp. $R^{ex} \subseteq \gtrsim$), $C \subseteq \gtrsim \cup >$, and $C \cap > \neq \emptyset$.
(3) There exists a maximal static recursion component C' such that $C \subseteq C'$ and C' satisfies one of the following properties:
   (i) C' satisfies the strictly subterm criterion.
   (ii) There exists a reduction pair (resp. semi-reduction pair) $(\gtrsim, >)$ w.r.t. a predicate P such that $P(s,t)$ holds for any $(s,t) \in R \cup C'$ (resp. $(s,t) \in R^{ex} \cup C'$), $R \subseteq \gtrsim$ (resp. $R^{ex} \subseteq \gtrsim$), and $C' \subseteq >$.

In case of $|SDP(R)| = n$, there exist $2^n - 1$ static recursion components in the worst case, but the number of maximal static recursion components is at most n. Hence, by checking (3) before checking (1) and (2), we can prove the termination more efficiently. This idea has already been formulated in [14], and used in early implementations in TRSs [2], [6].

Example 3.23 Consider the SFP-STRS $R_{\mathtt{sumF}}$ shown in Example 3.10. All $C \in SRC(R_{\mathtt{sumF}})$ shown in Example 3.13 satisfy the subterm criterion by setting π to the underlined parts below ($\pi(\mathtt{foldl}) = 3$ and $\pi(\mathtt{add}) = \pi(\mathtt{app}_\mathbb{0}) = 1$):

$$\{\mathtt{foldl}^\sharp[f, y, \underline{\mathtt{cons}[x, xs]}] \to \mathtt{foldl}^\sharp[f, f[y, x], \underline{xs}]\}$$
$$\{\mathtt{add}^\sharp[\underline{\mathtt{s}[x]}, y] \to \mathtt{add}^\sharp[\underline{x}, y]\}$$
$$\{\mathtt{app}_\mathbb{0}{}^\sharp[\underline{\mathtt{consF}[f, xs]}] \to \mathtt{app}_\mathbb{0}{}^\sharp[\underline{xs}]\}$$

Hence, the termination is shown by Theorem 3.22.

4. Argument Filtering Method

The argument filtering method, designed by eliminating unnecessary subterms, generates a reduction pair from a given reduction order. Arts and Giesl first introduced the method on first-order TRSs [1], Kusakari then extended the method to STRSs [18].

In the argument filtering method in [18], the term $\mathtt{sub}[x, y]$ is transformed into $\mathtt{sub}[x]$ after argument filtering. Thus, the type of sub should be interpreted as $\tau(\mathtt{sub}) = N \to N$ after argument filtering. However, when $\mathtt{add}[x, y]$ does not change by argument filtering, the type of add should not change, that is, $\tau(\mathtt{add}) = N \to N \to N$. Hence, for a higher-order variable $f^{N\to N\to N}$ we cannot decide the type of f after argument filtering, because the type should correspond with both substitutions $\{f := \mathtt{add}\}$ and $\{f := \mathtt{sub}\}$. As a consequence, the argument filtering method in [18] may destroy the well-typedness of terms. When the method applies to a reduction order which makes use of type information, this fact remarkably complicates the application, and some redundant condition may be required (cf. [19]).

In this section we improve the argument filtering method. In the new argument filtering method, the term $\mathtt{sub}[x, y]$ is transformed into $\mathtt{sub}[x, \bot]$ instead of $\mathtt{sub}[x]$. The method, then, never destroys the well-typedness. Although the idea is surely simple, our improvement yields very substantial benefits when combined with reduction orders that make use of type information. Indeed, in contrast to the method in [19], we need not individually discuss application to each reduction order, and we can comb out some applied conditions as described later.

Definition 4.1 We prepare the fresh function symbol $\bot_\alpha$ with $\tau(\bot_\alpha) = \alpha$, for each $\alpha \in \mathcal{S}$.

An *argument filtering function* is a function π such that for any $f \in \Sigma$, $\pi(f)$ is a list of positive integers $[i_1, \ldots, i_k]$ with $i_1 < \cdots < i_k \le n$, where $\tau(f) = \alpha_1 \to \cdots \to \alpha_n \to \beta$ and $\beta \in \mathcal{S}_{nfun}$. We extend π over terms as $\pi(a[t_1, \ldots, t_n]) = a[t'_1, \ldots, t'_n]$, where $t'_i \equiv \bot_{\alpha_i}$ if $a \in \Sigma$ and $i \notin \pi(a)$; otherwise $t'_i \equiv \pi(t_i)$. We also define θ_π by $\theta_\pi(x) = \pi(\theta(x))$.

For given argument filtering function π and binary relation $>$, we define $s \gtrsim^\pi t$ by $\pi(s) \ge \pi(t)$, and $s >^\pi t$ by $\pi(s) > \pi(t)$.

We often omit the index α in $\bot_\alpha$ whenever no confusion arises. We hereafter assume that if $\pi(f)$ is not defined explicitly then it is intended to be $[1, \ldots, n]$, where $\tau(f) = \alpha_1 \to \cdots \alpha_n \to \beta$ and $\beta \in \mathcal{S}_{nfun}$.

In the definition above, it is easily seen that if t has a type α then so does $\pi(t)$.

Example 4.2 Let $R_{\mathtt{div}}$ be the following STRS.

$$\begin{cases} \mathtt{sub}[x, 0] & \to & x \\ \mathtt{sub}[0, y] & \to & 0 \\ \mathtt{sub}[\mathtt{s}[x], \mathtt{s}[y]] & \to & \mathtt{sub}[x, y] \\ \mathtt{div}[0, \mathtt{s}[y]] & \to & 0 \\ \mathtt{div}[\mathtt{s}[x], \mathtt{s}[y]] & \to & \mathtt{s}[\mathtt{div}[\mathtt{sub}[x, y], \mathtt{s}[y]]] \end{cases}$$

Let $\pi(\mathtt{sub}) = [1]$ for a function symbol sub with $\tau(\mathtt{sub}) = N \to N \to N$. Then $\pi(\mathtt{sub}[x, y]) = \mathtt{sub}[x, \bot_N]$.

Unfortunately, as indicated in [18], $\gtrsim^\pi$ is not closed under substitutions. Our improved method cannot solve this problem. For example, let $\theta(f) = \mathtt{foo}$, $\pi(\mathtt{foo}) = [2]$ and $>_{rpo}$ be a recursive path order in [19] (cf. Definition 4.6) with the precedence $2 \rhd 1 \rhd 0$. Then we have $\pi(f[2, 0]) \equiv f[2, 0]$, $\pi(f[1, 1]) \equiv f[1, 1]$, $\pi(f[2, 0]\theta) \equiv \pi(\mathtt{foo}[2, 0]) \equiv \mathtt{foo}[\bot, 0]$, and $\pi(f[1, 1]\theta) \equiv \pi(\mathtt{foo}[1, 1]) \equiv \mathtt{foo}[\bot, 1]$. Thus, we obtain the following counterexample:

$$f[2, 0] >_{rpo} f[1, 1], \text{ but } \mathtt{foo}[\bot, 0] <_{rpo} \mathtt{foo}[\bot, 1].$$

Hence, the notion of left-firmness was introduced [18].

Definition 4.3 A term t is said to be *firmness* if any variable occurs at a leaf position. A pair (s, t) of terms is said to be *left-firmness*, denoted by $LF(s, t)$, if s is firmness.

Definition 4.4 A (semi-)reduction order $>$ satisfies the $\bot$-*condition* if $t \ge \bot_\alpha$ for any t^α.

Theorem 4.5 For any (semi-)reduction order $>$ with $\bot$-condition, the pair $(\gtrsim^\pi, >^\pi)$ is a (semi-)reduction pair w.r.t. the predicate LF.

Proof. It suffices to show that $\pi(t)\theta_\pi \ge \pi(t\theta)$ for any $t \equiv a[t_1, \ldots, t_n]$. Note that $\pi(s)\theta_\pi \equiv \pi(s\theta)$ for any firmness term s can be proved as similar to the proof. These properties show the LF-closedness of $\gtrsim^\pi, >^\pi$ under substitutions: $\pi(s) \ge \pi(t) \Rightarrow \pi(s\theta) \equiv \pi(s)\theta_\pi \ge \pi(t)\theta_\pi \ge \pi(t\theta)$ and $\pi(s) > \pi(t) \Rightarrow \pi(s\theta) \equiv \pi(s)\theta_\pi > \pi(t)\theta_\pi \ge \pi(t\theta)$. Moreover, the remainder of conditions can be proved similar to the proof of the early argument filtering method in [18].

We prove the claim by induction on $|t|$. From the induction hypothesis, $\pi(t_i)\theta_\pi \ge \pi(t_i\theta)$ for any i.

In case of $a \in \Sigma$, we suppose that $t'_i \equiv \bot$ if $i \notin \pi(a)$; otherwise $t'_i \equiv \pi(t_i)$, and $t''_i \equiv \pi(t_i\theta)$ if $i \in \pi(a)$; otherwise $t''_i \equiv \bot$. Then we have $t'_i\theta_\pi \ge t''_i$, and hence $\pi(t)\theta_\pi \equiv a[t'_1\theta_\pi, \ldots, t'_n\theta_\pi] \ge a[t''_1, \ldots, t''_n] \equiv \pi(t\theta)$.

In case of $a \in \mathcal{V}$ and $root(\theta(a)) \in \mathcal{V}$, we have $\pi(t)\theta_\pi \equiv \theta_\pi(a)[\pi(t_1)\theta_\pi, \ldots, \pi(t_n)\theta_\pi] \ge \theta_\pi(a)[\pi(t_1\theta), \ldots, \pi(t_n\theta)] \equiv \pi(\theta(a)[t_1\theta, \ldots, t_n\theta]) \equiv \pi(t\theta)$.

In case of $a \in \mathcal{V}$ and $root(\theta(a)) \in \Sigma$, we suppose that $\theta(a) = a'[u_1, \ldots, u_k]$ and $t''_i \equiv \pi(t_i\theta)$ if $i + k \in \pi(a')$; otherwise $t''_i \equiv \bot$. Then we have $\pi(t_i)\theta \ge t''_i$, and hence $\pi(t)\theta_\pi \equiv \theta_\pi(a)[\pi(t_1)\theta_\pi, \ldots, \pi(t_n)\theta_\pi] \ge \theta_\pi(a)[t''_1, \ldots, t''_n] \equiv \pi(\theta(a)[t_1\theta, \ldots, t_n\theta]) \equiv \pi(t\theta)$. □

The argument filtering method improved in this paper never destroys the well-typedness. Our improvement yields

very substantial benefits when combined with reduction orders that make use of type information as follows:

Definition 4.6 [19] A *precedence* $\rhd$ is a strict partial order on Σ. For any $s \equiv a[s_1, \ldots, s_n]$ and $t \equiv a'[t_1, \ldots, t_m]$, we define $s >_{rpo} t$ if $\tau(s)$ and $\tau(t)$ have the same type under identifying all basic types, and one of the following properties holds:

- $\tau(s) \in \mathcal{B}$, $a \rhd a'$ and for all j either $s >_{rpo} t_j$ or $\exists i.\ s_i \geq_{rpo} t_j$,
- $a = a'$ and $\{s_1, \ldots, s_n\} >_{rpo}^{mul} \{t_1, \ldots, t_m\}$, where $>_{rpo}^{mul}$ is the multiset extension of $>_{rpo}$, or
- there exists k such that $\exists i.\ s_i \geq_{rpo} a'[t_1, \ldots, t_k]$ and $\forall j > k.\ \exists i_j.\ s_{i_j} \geq_{rpo} t_j$.

Proposition 4.7 [19] $>_{rpo}^{+}$ is a reduction order.

Note that $>_{rpo}$ is not transitive, however this is not a problem for proving termination.

Since the argument filtering method in [18] may destroy the well-typedness of terms, the method with $>_{rpo}$ requires the following strong restriction:

- For any $l \to r \in R$ and $x \in Var(\pi(l))$, if $\tau(x) \in \mathcal{S}_{fun}$ then for each $i \in \pi(root(l))$ we have either $x \equiv l_i$ or $x \notin Var(\pi(l_i))$.

On the other hand, the new argument filtering method in this paper does not require such restrictions.

Example 4.8 Consider the left-firmness SFP-STRS $R_{\mathtt{div}}$ given in Example 4.2. Then the set $SRC(R_{\mathtt{div}})$ consists of the following two static recursion components:

$$\{\mathtt{sub}^\sharp[\mathtt{s}[x], \mathtt{s}[y]] \to \mathtt{sub}^\sharp[x, y]\}$$
$$\{\mathtt{div}^\sharp[\mathtt{s}[x], \mathtt{s}[y]] \to \mathtt{div}^\sharp[\mathtt{sub}[x, y], \mathtt{s}[y]]\}$$

The first component satisfies the subterm criterion. For the second component, we have

$$\begin{array}{rcl}
\mathtt{div}^\sharp[\mathtt{s}[x], \mathtt{s}[y]] & >_{rpo}^{\pi} & \mathtt{div}^\sharp[\mathtt{sub}[x, y], \mathtt{s}[y]] \\
\mathtt{sub}[x, 0] & \gtrsim_{rpo}^{\pi} & x \\
\mathtt{sub}[0, y] & \gtrsim_{rpo}^{\pi} & 0 \\
\mathtt{sub}[\mathtt{s}[x], \mathtt{s}[y]] & \gtrsim_{rpo}^{\pi} & \mathtt{sub}[x, y] \\
\mathtt{div}[0, \mathtt{s}[y]] & \gtrsim_{rpo}^{\pi} & 0 \\
\mathtt{div}[\mathtt{s}[x], \mathtt{s}[y]] & \gtrsim_{rpo}^{\pi} & \mathtt{s}[\mathtt{div}[\mathtt{sub}[x, y], \mathtt{s}[y]]]
\end{array}$$

with $\pi(\mathtt{sub}) = [1]$ and $\mathtt{div} \rhd \mathtt{s} \rhd \mathtt{sub}$. Hence, the termination of $R_{\mathtt{div}}$ can be shown by Theorem 3.22 and 4.5.

Example 4.9 Let $R_{\mathtt{ave}}$ be the left-firmness SFP-STRS, which is the union of $R_{\mathtt{sum}}$, $R_{\mathtt{div}}$ and the following rules:

$$\left\{\begin{array}{rcl}
\mathtt{s}'[x, y] & \to & \mathtt{s}[x] \\
\mathtt{len} & \to & \mathtt{foldl}[\mathtt{s}', 0] \\
\mathtt{ave}[xs] & \to & \mathtt{div}[\mathtt{sum}[xs], \mathtt{len}[xs]]
\end{array}\right.$$

Here $R_{\mathtt{sum}}$ and $R_{\mathtt{div}}$ are displayed in the Introduction and Example 4.8, respectively. Then the function `ave` calculates the average $\frac{x_1 + \cdots + x_n}{n}$ for an input list $[x_1, \ldots, x_n]$. The set $SRC(R_{\mathtt{ave}})$ consists of the following four static recursion components:

$$\{\mathtt{add}^\sharp[\mathtt{s}[x], y] \to \mathtt{add}^\sharp[x, y]\}$$
$$\{\mathtt{foldl}^\sharp[f, y, \mathtt{cons}[x, xs]] \to \mathtt{foldl}^\sharp[f, f[y, x], xs]\}$$
$$\{\mathtt{sub}^\sharp[\mathtt{s}[x], \mathtt{s}[y]] \to \mathtt{sub}^\sharp[x, y]\}$$
$$\{\mathtt{div}^\sharp[\mathtt{s}[x], \mathtt{s}[y]] \to \mathtt{div}^\sharp[\mathtt{sub}[x, y], \mathtt{s}[y]]\}$$

Any static recursion component except for the last component satisfies the subterm criterion. However, different than Example 4.8, the non-loopingness of the last component cannot be shown, because the constraint $R_{\mathtt{foldl}} \subseteq \gtrsim_{rpo}^{\pi}$ cannot be solved.

To show the termination of $R_{\mathtt{ave}}$ we need the notion of usable rules that will be introduced in the next section.

5. Usable Rules with Argument Filtering

First, we consider why the non-loopingness of the static recursion component

$$\{\mathtt{div}^\sharp[\mathtt{s}[x], \mathtt{s}[y]] \to \mathtt{div}^\sharp[\mathtt{sub}[x, y], \mathtt{s}[y]]\}$$

can be shown in Example 4.8, but cannot be shown in Example 4.9. The reason is that we should solve the constraint $R_{\mathtt{foldl}} \subseteq \gtrsim_{rpo}^{\pi}$ in Example 4.9, but not in Example 4.8. Many programmers may query why we should orient rules for `foldl` in order to show the non-loopingness for `div`. The notion of usable rules solves this problem.

The notion of usable rules was introduced in TRSs [11], [15], [29], which is based on the technique of interpretation and the notion of C_e-termination [13], [30]. Afterward we extended the method to STRSs [26]. By using the usable rules for STRSs, we can show the non-loopingness for `div`, because we can solve the following constraint:

$$\begin{array}{rcll}
\mathtt{div}^\sharp[\mathtt{s}[x], \mathtt{s}[y]] & >_{rpo}^{\pi} & \mathtt{div}^\sharp[\mathtt{sub}[x, y], \mathtt{s}[y]] & \\
\mathtt{sub}[x, 0] & \gtrsim_{rpo}^{\pi} & x & \\
\mathtt{sub}[0, y] & \gtrsim_{rpo}^{\pi} & 0 & \\
\mathtt{sub}[\mathtt{s}[x], \mathtt{s}[y]] & \gtrsim_{rpo}^{\pi} & \mathtt{sub}[x, y] & \\
\mathtt{c}_\alpha[x, y] & \gtrsim_{rpo}^{\pi} & x & \text{for any } \alpha \in \mathcal{S} \\
\mathtt{c}_\alpha[x, y] & \gtrsim_{rpo}^{\pi} & y & \text{for any } \alpha \in \mathcal{S}
\end{array}$$

We can see that the constraint above does not include $R_{\mathtt{foldl}} \subseteq \gtrsim_{rpo}^{\pi}$, which prevents us from showing the termination of the STRS $R_{\mathtt{ave}}$.

Next, we consider the STRS $R_{\mathtt{sum\_n}}$ of the union of $R_{\mathtt{sum}}$ and the following rules:

$$\left\{\begin{array}{rcl}
\mathtt{drop}[0, yss] & \to & yss \\
\mathtt{drop}[x, \mathtt{nilL}] & \to & \mathtt{nilL} \\
\mathtt{drop}[\mathtt{s}[x], \mathtt{consL}[y, yss]] & \to & \mathtt{drop}[x, yss] \\
\mathtt{sum_n}[v, x, \mathtt{nilL}] & \to & v \\
\mathtt{sum_n}[v, \mathtt{s}[x], \mathtt{consL}[xs, xss]] & & \\
\quad \to \mathtt{sum_n}[\mathtt{add}[v, \mathtt{sum}[xs]], \mathtt{s}[x], \mathtt{drop}[x, xss]] & &
\end{array}\right.$$

Then, the function $\mathtt{sum\_n}[0, n, xss]$ calculates the total sum of the total sums of $xs_0, xs_n, xs_{2n}, \ldots$ for an input list of lists $xss = [xs_0, xs_1, \ldots, xs_m]$. The set $SRC(R_{\mathtt{sum\_n}})$ consists of the following four static recursion components:

$$\{\mathtt{foldl}^\sharp[f, y, \mathtt{cons}[x, xs]] \to \mathtt{foldl}^\sharp[f, f[y, x], xs]\}$$
$$\{\mathtt{add}^\sharp[\mathtt{s}[x], y] \to \mathtt{add}^\sharp[x, y]\}$$
$$\{\mathtt{drop}^\sharp[\mathtt{s}[x], \mathtt{consL}[y, yss]] \to \mathtt{drop}^\sharp[x, yss]\}$$
$$\{\mathtt{sum\_n}^\sharp[v, \mathtt{s}[x], \mathtt{consL}[xs, xss]] \to \mathtt{sum\_n}^\sharp[\mathtt{add}[v, \mathtt{sum}[xs]], \mathtt{s}[x], \mathtt{drop}[x, xss]]\}$$

Any static recursion component except for the last component satisfies the subterm criterion. However, as in Example 4.9, the non-loopingness of the last component cannot be shown, because the constraint $R_{\mathtt{foldl}} \subseteq \gtrsim^{\pi}_{rpo}$ cannot be solved. This problem cannot be solved by usable rules for STRSs [18].

In first-order TRSs, we know that usable rules can be strengthened by incorporating argument filtering into usable rules [11], [29]. In this section, we also strengthen usable rules for STRSs [26] by incorporating argument filtering into usable rules. Then we can reduce $R_{\mathtt{foldl}} \subseteq \gtrsim^{\pi}_{rpo}$ from the constraint that we should solve, and hence we can prove the termination of the STRS $R_{\mathtt{sum\_n}}$.

Definition 5.1 For any $t \equiv a[t_1, \ldots, t_n]$, we define $Sub_\pi(t)$ as $\{t\} \cup \bigcup_{i \in I} Sub_\pi(t_i)$, where $I = \pi(a)$ if $a \in \Sigma$; otherwise $I = \{1, \ldots, n\}$, and $Sub^{int}_{\mathcal{V},\pi}(t)$ as $\{t' \in Sub_\pi(t) \mid root(t') \in \mathcal{V}, args(t') \neq \emptyset\}$.

Definition 5.2 For each pair $\langle u, v\rangle$ of terms, the subset $\mathcal{U}'(\langle u, v\rangle, \pi)$ of STRS R is defined by $l \to r \in \mathcal{U}'(\langle u, v\rangle, \pi)$ iff $l \to r$ satisfies one of the following conditions:

(1) $root(l) = root(v')$ and $\tau(l) \sqsupseteq_{\mathcal{S}} \tau(v')$ for some $v' \in Sub_\pi(v)$,
(2) $\tau(root(l)) \sqsupseteq_{\mathcal{S}} \tau(root(v'))$ and $\tau(l) \sqsupseteq_{\mathcal{S}} \tau(v')$ for some $v' \in Sub^{int}_{\mathcal{V},\pi}(v)$, or
(3) $\tau(l) \sqsupseteq_{\mathcal{S}} \tau(root(u'))$ for some $u' \in Sub^{int}_{\mathcal{V},\pi}(u)$ with $root(u') \in Var(v)$.

We define the set $\mathcal{U}(\langle u, v\rangle, \pi)$ by the smallest set satisfying $\mathcal{U}'(\langle u, v\rangle^{ex\uparrow}, \pi) \subseteq \mathcal{U}(\langle u, v\rangle, \pi)$, and $\mathcal{U}(\langle l, r\rangle^{ex\uparrow}, \pi) \subseteq \mathcal{U}(\langle u, v\rangle, \pi)$ whenever $l \to r \in \mathcal{U}(\langle u, v\rangle, \pi)$. For each set C of pairs of terms, we define *usable rules with argument filtering* π by $\mathcal{U}(C, \pi) = \bigcup_{\langle u,v\rangle \in C} \mathcal{U}(\langle u, v\rangle, \pi)$.

Notice that $\mathcal{U}(C, \pi)$ is the same as the usable rules $\mathcal{U}(C)$ without argument filtering in [26] whenever $\pi(f) = [1, \ldots, n]$ for any $f^{\alpha_1 \to \cdots \alpha_n \to \beta} \in \Sigma$ with $\beta \in \mathcal{S}_{nfun}$.

Example 5.3 We suppose that C is the static recursion component

$$\{\mathtt{sum\_n}^\sharp[v, \mathtt{s}[x], \mathtt{consL}[xs, xss]] \to \mathtt{sum\_n}^\sharp[\mathtt{add}[v, \mathtt{sum}[xs]], \mathtt{s}[x], \mathtt{drop}[x, xss]]\}$$

of STRS $R_{\mathtt{sum\_n}}$, which is the second example in the beginning of this section. Let $\pi(\mathtt{sum\_n}^\sharp) = [3]$. Then the set $\mathcal{U}(C, \pi)$ consists of only three rules for $\mathtt{drop}$.

Note that the usable rules $\mathcal{U}(C)$ without argument filtering in [26] consist of eight rules for $\mathtt{drop}$, $\mathtt{add}$, $\mathtt{sum}$, and $\mathtt{foldl}$.

In the following, we assume that R is a finitely branching STRS, C is a static recursion component, and $t \in \Delta$ iff $root(t) = root(l)$ and $\tau(l) \sqsupseteq_{\mathcal{S}} \tau(t)$ for some $l \to r \in R \setminus \mathcal{U}(C, \pi)$.

Notice that any redex for $(R \setminus \mathcal{U}(C, \pi))^{ex}$ is in Δ. That is, if $t \equiv l\theta$ for some $l \to r \in (R \setminus \mathcal{U}(C, \pi))^{ex}$ and θ, then $t \in \Delta$.

By eliminating rules in $R \setminus \mathcal{U}(C, \pi)$, the notion of usable rules reduces the constraints for non-loopingness. In this elimination, we must carefully analyze a dependency between rules. In the definition of $\mathcal{U}(C, \pi)$, condition (1) is for analysis of a dependency through defined symbols, which is the same analysis as first-order settings. Conditions (2) and (3) are for analysis of a dependency through higher-order variables in right- and left-hand sides, respectively. Condition (3) seems to be unnatural because it is for left-hand sides. However, condition (3) is necessary for technical reasons (cf. Lemma 5.6).

Lemma 5.4 For each $l \to r \in C \cup \mathcal{U}(C, \pi)^{ex}$ and θ, the following properties hold:

(1) $v\theta \notin \Delta$ for all $v \in Sub_\pi(r)$ with $root(v) \in \Sigma$,
(2) $v\theta \notin \Delta$ for all $v \in Sub^{int}_{\mathcal{V},\pi}(r)$, and
(3) $root(u)\theta \notin \Delta$ for all $u \in Sub^{int}_{\mathcal{V},\pi}(l)$ with $root(u) \in Var(r)$.

Proof. (1) Assume that $v\theta \in \Delta$. Then there exists $l' \to r' \in R \setminus \mathcal{U}(C, \pi)$ such that $root(v\theta) = root(l')$ and $\tau(l') \sqsupseteq_{\mathcal{S}} \tau(v\theta)$. Since $root(v) \in \Sigma$, we have $root(v) = root(l')$. Since $\tau(l') \sqsupseteq_{\mathcal{S}} \tau(v\theta) = \tau(v)$, we have $l' \to r' \in \mathcal{U}'(\langle l, r\rangle^{ex\uparrow}, \pi)$. Hence, $l' \to r' \in \mathcal{U}(C, \pi)$, which is a contradiction.
(2) Assume that $v\theta \in \Delta$. Then there exists $l' \to r' \in R \setminus \mathcal{U}(C, \pi)$ such that $root(v\theta) = root(l')$ and $\tau(l') \sqsupseteq_{\mathcal{S}} \tau(v\theta)$. Since $root(v) \in \mathcal{V}$ and $root(v\theta) = root(l')$, we have $\tau(root(l')) \sqsupseteq_{\mathcal{S}} \tau(root(v))$. Since $\tau(l') \sqsupseteq_{\mathcal{S}} \tau(v\theta) = \tau(v)$, we have $l' \to r' \in \mathcal{U}'(\langle l, r\rangle^{ex\uparrow}, \pi)$. Hence, $l' \to r' \in \mathcal{U}(C, \pi)$, which is a contradiction.
(3) Assume that $root(u)\theta \in \Delta$. Then there exists $l' \to r' \in R \setminus \mathcal{U}(C, \pi)$ such that $root(root(u)\theta) = root(l')$ and $\tau(l') \sqsupseteq_{\mathcal{S}} \tau(root(u)\theta)$. Thus, we have $l' \to r' \in \mathcal{U}'(\langle l, r\rangle^{ex\uparrow}, \pi)$. Hence, $l' \to r' \in \mathcal{U}(C, \pi)$, which is a contradiction. □

Definition 5.5 For each $\alpha \in \mathcal{S}$, we prepare the fresh function symbol $\bot_\alpha$ and $\mathtt{c}_\alpha$ with $\tau(\bot_\alpha) = \alpha$ and $\tau(\mathtt{c}_\alpha) = \alpha \to \alpha \to \alpha$. We define the STRS C_e by $\{\mathtt{c}_\alpha[x, y] \to x \mid \alpha \in \mathcal{S}\} \cup \{\mathtt{c}_\alpha[x, y] \to y \mid \alpha \in \mathcal{S}\}$.

The interpretation I_π is a mapping from terminating terms in $\mathcal{T}_\tau(\Sigma, \mathcal{V})$ to terms in $\mathcal{T}_\tau(\Sigma \cup \bigcup_{\alpha \in \mathcal{S}}\{\bot_\alpha, \mathtt{c}_\alpha\}, \mathcal{V})$; for each $t^\alpha \equiv a[t_1^{\alpha_1}, \ldots, t_n^{\alpha_n}]$, $I_\pi(t)$ is defined as follows:

$$\begin{cases} a[t'_1,\dots,t'_n] & \text{if } t \notin \Delta \\ \mathsf{c}_\alpha[a[t'_1,\dots,t'_n], Red_\alpha(\{I_\pi(t') \mid t \xrightarrow[R]{} t'\})] & \text{if } t \in \Delta \end{cases}$$

where $t'_i \equiv I_\pi(t_i)$ if either $a \in \mathcal{V}$ or $a \in \Sigma$ and $i \in \pi(a)$; otherwise $t'_i \equiv \bot_{\alpha_i}$, and

$$Red_\alpha(T) = \begin{cases} \bot_\alpha & \text{if } T = \emptyset \\ \mathsf{c}_\alpha[least(T), Red(T \setminus \{t\})] & \text{if } T \neq \emptyset \end{cases}$$

Thanks to the well-ordering theorem, we assume an arbitrary but fixed well-order on $\mathcal{T}_\tau(\Sigma,\mathcal{V})$. We denote by $least(T)$ the least element in T with respect to the well-order. For each terminating substitution θ, we define θ^{I_π} by $\theta^{I_\pi}(x) = I_\pi(\theta(x))$ for each $x \in \mathcal{V}$.

The interpretation I_π is inductively defined on terminating terms with respect to $>_{sub} \cup \xrightarrow[R]{}$, which is well-founded on terminating terms. Moreover, the set $\{I_\pi(t') \mid t \xrightarrow[R]{} t'\}$ is finite because R is finitely branching. Hence, the above definition of I_π is well-defined.

Lemma 5.6 Let $l \to r \in C \cup \mathcal{U}(C,\pi)^{ex}$ and θ be a substitution such that $l\theta$ is terminating. We define σ as $\sigma(x) = u$ if $x \notin Var(r)$ and $\theta^{I_\pi}(x)$ has the form $\mathsf{c}[u,T]$; otherwise $\sigma(x) = \theta^{I_\pi}(x)$. Then we have $I_\pi(l\theta) \xrightarrow[C_e]{*} \pi(l)\sigma$,

Proof. We prove $\forall t \in Sub_\pi(l).I_\pi(t\theta) \xrightarrow[C_e]{*} \pi(t)\sigma$ by induction on t. Let $t \equiv a[t_1,\dots,t_n]$.

- In case of $a \in \Sigma$: We suppose that $t'_i \equiv \pi(t_i)$ and $t''_i \equiv I_\pi(t_i\theta)$ if $i \in \pi(a)$; otherwise $t'_i \equiv t''_i \equiv \bot$. Then we have $I_\pi(t\theta) \equiv I_\pi(a[t_1\theta,\dots,t_n\theta])$ $(\equiv \cup \xrightarrow[C_e]{})$ $a[t''_1,\dots,t''_n] \xrightarrow[C_e]{*} a[t'_1\sigma,\dots,t'_n\sigma] \equiv \pi(t)\sigma$.
- In case that $a \in \mathcal{V}$ and $\sigma(a)$ does not have the form $\mathsf{c}[u,T]$: We suppose that $\theta(a) \equiv a'[u_1,\dots,u_k]$, and $t'_i \equiv \pi(t_i)$ and $t''_i \equiv I_\pi(t_i\theta)$ if either $a' \in \mathcal{V}$ or $a' \in \Sigma$ and $i+k \in \pi(a')$; otherwise $t'_i \equiv t''_i \equiv \bot$. Then we have $I_\pi(t\theta) \equiv I_\pi(\theta(a)[t_1\theta,\dots,t_n\theta])$ $(\equiv \cup \xrightarrow[C_e]{})$ $\sigma(a)[t''_1,\dots,t''_n] \xrightarrow[C_e]{*} \sigma(a)[t'_1\sigma,\dots,t'_n\sigma] \equiv \pi(t)\sigma$.
- In case that $a \in \mathcal{V}$ and $\sigma(a)$ has the form $\mathsf{c}[u,T]$: From the definition of σ, $\theta^{I_\pi}(a)$ has the form $\mathsf{c}[u,T]$ and $a \in Var(r)$. Since $\theta^{I_\pi}(a)$ has the form $\mathsf{c}[u,T]$, we have $\theta(a) \in \Delta$.
  Assume that $n > 0$. Since $t \in Sub_\pi(l)$ and $a \in \mathcal{V}$, we have $t \in Sub^{int}_{\mathcal{V},\pi}(l)$. From Lemma 5.4 (3), we have $\theta(a) \notin \Delta$, which leads to a contradiction. Hence we have $n = 0$, that is, $t \equiv a[\,]$. Therefore we have $I_\pi(t\theta) \equiv I_\pi(a\theta) \equiv a\theta^{I_\pi} \equiv t\sigma$. □

Lemma 5.7 Let $l \to r \in C \cup \mathcal{U}(C,\pi)^{ex}$ and θ be a substitution such that $r\theta$ is terminating. Then we have $I_\pi(r\theta) \equiv \pi(r)\theta^{I_\pi}$.

Proof. We prove $\forall t \in Sub_\pi(r).I_\pi(t\theta) \equiv \pi(t)\theta^{I_\pi}$ by induction on t. Let $t \equiv a[t_1,\dots,t_n]$.

- In case of $a \in \Sigma$, we suppose that $t'_i \equiv \pi(t_i)$ and $t''_i \equiv I_\pi(t_i\theta)$ if $i \in \pi(a)$; otherwise $t'_i \equiv t''_i \equiv \bot$. From Lemma 5.4 (1), we have $I_\pi(t\theta) \equiv I_\pi(a[t_1\theta,\dots,t_n\theta]) \equiv a[t''_1,\dots,t''_n] \equiv a[t'_1\theta^{I_\pi},\dots,t'_n\theta^{I_\pi}] \equiv \pi(t)\theta^{I_\pi}$.
- In case of $a \in \mathcal{V}$ and $t\theta \notin \Delta$, we suppose that $\theta(a) \equiv a'[u_1,\dots,u_k]$, and $t'_i \equiv \pi(t_i)$ and $t''_i \equiv I_\pi(t_i\theta)$ if either $a' \in \mathcal{V}$ or $a' \in \Sigma$ and $i+k \in \pi(a')$; otherwise $t'_i \equiv t''_i \equiv \bot$.
  Assume that $a\theta \in \Delta$. Then there exists $l' \to r' \in R \setminus \mathcal{U}(C,\pi)$ such that $root(a\theta) = root(l')$ and $\tau(l') \sqsupseteq_{\mathcal{S}} \tau(a\theta)$. Since $root(t\theta) = root(a\theta) = root(l')$ and $\tau(l') \sqsupseteq_{\mathcal{S}} \tau(a\theta) \sqsupseteq_{\mathcal{S}} \tau(t\theta)$, we have $t\theta \in \Delta$, which leads to a contradiction. Thus, we have $a\theta \notin \Delta$. Hence we have $I_\pi(t\theta) \equiv I_\pi(\theta(a)[t_1\theta,\dots,t_n\theta]) \equiv \theta^{I_\pi}(a)[t''_1,\dots,t''_n] \equiv \theta^{I_\pi}(a)[t'_1\theta^{I_\pi},\dots,t'_n\theta^{I_\pi}] \equiv \pi(t)\theta^{I_\pi}$.
- In case of $a \in \mathcal{V}$ and $t\theta \in \Delta$, we have $t \equiv a[\,]$ from Lemma 5.4 (2). Hence $I_\pi(t\theta) \equiv I_\pi(a\theta) \equiv a\theta^{I_\pi} \equiv t\theta^{I_\pi}$. □

Lemma 5.8 If $s \xrightarrow[R]{} t$ and s is terminating then $I_\pi(s) \xrightarrow[\pi(\mathcal{U}(C,\pi))\cup C_e]{*} I_\pi(t)$.

Proof. From Proposition 2.1, there exist a rule $l \to r \in R^{ex}$, a leaf-context $C[\,]$ and substitution θ such that $s \equiv C[l\theta]$ and $t \equiv C[r\theta]$. We prove the claim by induction on $C[\,]$. Because $C[\,]$ is a leaf-context, it suffices to show the following cases.

- Suppose that $C[\,] \equiv \Box$ and $s \notin \Delta$. Then $l \to r \in \mathcal{U}(C,\pi)^{ex}$. We define the substitution σ as similar to Lemma 5.6. From Lemma 5.6 and 5.7, we have $I_\pi(l\theta) \xrightarrow[C_e]{*} \pi(l)\sigma \xrightarrow[\pi(\mathcal{U}(C,\pi))]{} \pi(r)\sigma \equiv \pi(r)\theta^{I_\pi} \equiv I_\pi(r\theta)$.
- Suppose that $C[\,] \equiv a[\dots,u_{i-1},C'[\,],u_{i+1},\dots]$, $s \notin \Delta$, $a \in \Sigma$ and $i \notin \pi(a)$. Then $t \notin \Delta$ and hence $I_\pi(C[l\theta]) \equiv a[\dots,\bot,\dots] \equiv I_\pi(C[r\theta])$.
- Suppose that $C[\,] \equiv a[\dots,u_{i-1},C'[\,],u_{i+1},\dots]$, $s \notin \Delta$, and either $a \in \mathcal{V}$ or $a \in \Sigma$ and $i \in \pi(a)$. Then $t \notin \Delta$, and hence $I_\pi(C[l\theta]) \equiv a[\dots,I_\pi(C'[l\theta]),\dots] \xrightarrow[\pi(\mathcal{U}(C,\pi))\cup C_e]{*} a[\dots,I_\pi(C'[r\theta]),\dots] \equiv I_\pi(C[r\theta])$.
- Suppose that $s \in \Delta$. Then $I_\pi(C[l\theta]) \xrightarrow[C_e]{} Red(\{I_\pi(v) \mid C[l\theta] \xrightarrow[R]{} v\}) \xrightarrow[C_e]{+} I_\pi(C[r\theta])$. □

Theorem 5.9 Let R be a finitely branching SFP-STRS, C be a static recursion component, and π be an argument filtering function such that $C \cup \mathcal{U}(C,\pi)$ is left-firmness. If there exists a reduction order (resp. semi-reduction order) $>$ satisfying the $\bot$-condition and the following conditions, then C is non-looping.

(i) $C_e \subseteq >$ (resp. $C_e^{ex} \subseteq >$),
(ii) $\mathcal{U}(C,\pi) \subseteq \gtrsim^\pi$ (resp. $\mathcal{U}(C,\pi)^{ex} \subseteq \gtrsim^\pi$), and
(iii) $C \subseteq \gtrsim^\pi$ and $C \cap >^\pi \neq \emptyset$.

Proof. We show only the case that $>$ is a reduction order.

Assume that pairs in C generate an infinite chain $u^\sharp_0 \to v^\sharp_0, u^\sharp_1 \to v^\sharp_1, u^\sharp_2 \to v^\sharp_2, \cdots$ in which every $u^\sharp \to v^\sharp \in C$ occurs infinitely many times, and let $\theta_0,\theta_1,\dots$ be substitutions such that $v^\sharp_i\theta_i \xrightarrow[R]{*} u^\sharp_{i+1}\theta_{i+1}$ and $u_i\theta_i, v_i\theta_i \in \mathcal{T}^{args}_{SN}(R)$ for each i.

Let i be an arbitrary number. From Lemma 5.7, we have $\pi(v^\sharp_i)\theta^{I_\pi}_i \equiv I_\pi(v^\sharp_i\theta_i)$. From Lemma 5.8, we have $I_\pi(v^\sharp_i\theta_i) \xrightarrow[\pi(\mathcal{U}(C,\pi))\cup C_e]{*} I(u^\sharp_{i+1}\theta_{i+1})$. From Lemma 5.6, we have $I_\pi(u^\sharp_{i+1}\theta_{i+1}) \xrightarrow[C_e]{*} \pi(u^\sharp_{i+1})\sigma_{i+1}$, where the substitution σ_{i+1} is

generated from $\theta^{I_\pi}_{i+1}$ as similar to Lemma 5.6. From the construction of σ_{i+1}, we have $\pi(v^\sharp_{i+1})\sigma_{i+1} \equiv \pi(v^\sharp_{i+1})\theta^{I_\pi}_{i+1}$. Hence we have $\pi(v^\sharp_i)\theta^{I_\pi}_i \equiv I_\pi(v^\sharp_i\theta_i) \gtrsim I(u^\sharp_{i+1}\theta_{i+1}) \gtrsim \pi(u^\sharp_{i+1})\sigma_{i+1} \gtrsim \pi(v^\sharp_{i+1})\sigma_{i+1} \equiv \pi(v^\sharp_{i+1})\theta^{I_\pi}_{i+1}$ for any i. Moreover, from $C \cap >^\pi \neq \emptyset$, we have $\pi(v^\sharp_j)\theta^{I_\pi}_j > \pi(v^\sharp_{j+1})\theta^{I_\pi}_{j+1}$ for infinitely many j. This contradicts the well-foundedness of $>$. □

Example 5.10 Consider the finitely branching and left-firmness SFP-STRS $R_{\mathtt{sum\_n}}$. As previously mentioned, any static recursion component except for the following component satisfies the subterm criterion:

$$\{\mathtt{sum\_n}^\sharp[v, \mathtt{s}[x], \mathtt{consL}[xs, xss]] \to \mathtt{sum\_n}^\sharp[\mathtt{add}[v, \mathtt{sum}[xs]], \mathtt{s}[x], \mathtt{drop}[x, xss]]\}$$

We suppose that C is this static recursion component as in Example 5.3. Suppose that $\pi(\mathtt{sum\_n}^\sharp) = [3]$ and $\pi(\mathtt{drop}) = [2]$. Then the set $\mathcal{U}(C, \pi)$ consists of only three rules for drop described in Example 5.3. Hence it suffices to show that the following constraint can be solved:

$$\begin{aligned}
&\mathtt{sum_n}^\sharp[v, \mathtt{s}[x], \mathtt{consL}[xs, xss]] \\
&\quad > \mathtt{sum_n}^\sharp[\mathtt{add}[v, \mathtt{sum}[xs]], \mathtt{s}[x], \mathtt{drop}[x, xss]] \\
&\qquad \mathtt{drop}[0, yss] \gtrsim yss \\
&\qquad \mathtt{drop}[x, \mathtt{nilL}] \gtrsim \mathtt{nilL} \\
&\mathtt{drop}[\mathtt{s}[x], \mathtt{consL}[y, yss]] \gtrsim \mathtt{drop}[x, yss] \\
&\qquad \mathtt{c}_\alpha[x, y] \gtrsim x \quad \text{for any } \alpha \in \mathcal{S} \\
&\qquad \mathtt{c}_\alpha[x, y] \gtrsim y \quad \text{for any } \alpha \in \mathcal{S}
\end{aligned}$$

Let $\triangleright$ be the precedence $\mathtt{consL} \triangleright \mathtt{drop}$. Then $(\gtrsim^\pi_{rpo}, >^\pi_{rpo})$ can solve the constraint above. Hence the non-loopingness of C follows from Theorem 5.9. Therefore the termination of STRS $R_{\mathtt{sum\_n}}$ follows from Corollary 3.17.

6. Concluding Remarks

In this paper, we presented powerful methods for proving termination of STRSs. We summarize these methods by incorporating Theorem 5.9 into Theorem 3.22.

Corollary 6.1 Let R be an SFP-STRS such that there exists no infinite path in the static dependency graph. For any $C \in SRC(R)$,

- C satisfies one of the properties of (1), (2), or (3) in Theorem 3.22, or
- R is finitely branching, and there exist a reduction order (resp. semi-reduction order) $>$ and an argument filtering function π such that $>$ satisfies the $\perp$-condition, $C \cup \mathcal{U}(C, \pi)$ is left-firmness, and properties (i), (ii), and (iii) in Theorem 5.9 hold.

Then R is terminating.

A difficulty of studying static dependency pair methods arises, because strong computability is not closed under the subterm relation. Hence, to strengthen static dependency pair methods, guaranteeing the strong computability of subterms as far as possible is necessary. In this paper, we introduced the notion of safe function-passing, which expands the application range of static dependency pair methods, more than the notion of plain function-passing [22]. To extend the applicable scope to static dependency pair methods other than safe function-passing, using the notion of pattern computable closure [4] might be interesting. This is a topic for future study.

The argument filtering method improved in this paper never destroys the well-typedness, although the argument filtering method in [18] may destroy the well-typedness of terms. Our improvement eliminates a strong restriction (see the discussion below Proposition 4.7). Moreover, although the method in [18] can only combine with reduction orders on a superset of simply-typed terms [19], the method in this paper can combine with any reduction orders on simply-typed terms. Since reduction orders for simply-typed settings are usually designed on simply-typed terms, our improvement yields very substantial benefits.

The notion of usable rules reduces the constraints for proving non-loopingness. In this paper, we strengthen the notion by incorporating argument filtering into usable rules. Usable rules with argument filtering decrease the constraints more effectively than usable rules without argument filtering [26]. Using usable rules with argument filtering, reduction pairs must be designed by the argument filtering method, which requires a left-firmness restriction. Usable rules without argument filtering can use any reduction pair. Although all existing reduction pairs in STRSs have been designed by the argument filtering method, if other methods design reduction pairs without the left-firmness restriction, then usable rules without argument filtering may revive.

In first-order TRSs, many termination provers have recently has developed [23]. These systems efficiently solve constraints by using an SAT solver. Developing a termination prover for STRSs based on our results will also be future work. We also hope to see the results of this research applied to inductive reasoning [20] and the Knuth-Bendix procedure [21] on STRSs.

Acknowledgments

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

This research was partially supported by MEXT KAKENHI #20500008, #18500011, #20300010, and by the Kayamori Foundation of Informational Science Advancement.

References

[1] T. Arts and J. Giesl, "Termination of term rewriting using dependency pairs," Theor. Comput. Sci., vol.236, pp.133–178, 2000.
[2] T. Arts, "System description: The dependency pair method," Proc.

11th Int. Conf. on Rewriting Techniques and Applications, LNCS 1833 (RTA2000), pp.261–264, 2000.
[3] F. Blanqui, "Termination and confluence of higher-order rewrite systems," Proc. 11th Int. Conf. on Rewriting Techniques and Applications, LNCS 1833 (RTA2000), pp.47–61, 2000.
[4] F. Blanqui, "Higher-order dependency pairs," Proc. 8th Int. Workshop on Termination (WST06), pp.22–26, 2006.
[5] N. Dershowitz, "Orderings for term-rewriting systems," Theor. Comput. Sci., vol.17, no.3, pp.279–301, 1982.
[6] C. Contejean, C. Marché, B. Monate, and X. Urbain, CiME version 2, 2000. Available at http://cime.lri.fr/
[7] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke, "Improving dependency pairs," Proc. 10th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning, LNAI 2850 (LPAR2003), pp.165–179, 2003.
[8] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke, "Automated termination proofs with AProVE," Proc. 15th Int. Conf. on Rewriting Techniques and Applications, LNCS 3091 (RTA2004), pp.210–220, 2004.
[9] J. Giesl, R. Thiemann, and P. Schneider-Kamp, "The dependency pair framework: Combining techniques for automated termination proofs," Proc. 11th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LNCS 3452 (LPAR2004), pp.301–331, 2005.
[10] J. Giesl, R. Thiemann, and P. Schneider-Kamp, "Proving and disproving termination of higher-order functions," Proc. 5th Int. Workshop on Frontiers of Combining Systems, LNAI 3717 (FroCoS'05), pp.216–231, 2005.
[11] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke, "Mechanizing and improving dependency pairs," J. Automated Reasoning, vol.37, no.3, pp.155–203, 2006.
[12] J.-Y. Girard, Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur, Ph.D. Thesis, University of Paris VII, 1972.
[13] G. Gramlich, "Generalized sufficient conditions for modular termination of rewriting," Applicable Algebra in Engineering, Communication and Computing, vol.5, pp.131–158, 1994.
[14] N. Hirokawa and A. Middeldorp, "Automating the dependency pair method," Proc. 19th Int. Conf. on Automated Deduction, LNAI 2741 (CADE03), pp.32–46, 2003.
[15] N. Hirokawa and A. Middeldorp, "Dependency pairs revisited," Proc. 15th Int. Conf. on Rewriting Techniques and Applications, LNCS 3091 (RTA2004), pp.249–268, 2004.
[16] J.-P. Jouannaud and A. Rubio, "The higher-order recursive path ordering," Proc. 14th Annual IEEE Symposium on Logic in Computer Science, pp.402–411, 1999.
[17] K. Kusakari, M. Nakamura, and Y. Toyama, "Argument filtering transformation," Proc. Int. Conf. on Principles and Practice of Declarative Programming, LNCS 1702 (PPDP'99), pp.47–61, 1999.
[18] K. Kusakari, "On proving termination of term rewriting systems with higher-order variables," IPSJ Trans. Programming, vol.42, no.SIG 7 (PRO 11), pp.35–45, 2001.
[19] K. Kusakari, "Higher-order path orders based on computability," IEICE Trans. Inf. & Syst., vol.E87-D, no.2, pp.352–359, Feb. 2004.
[20] K. Kusakari, M. Sakai, and T. Sakabe, "Primitive inductive theorems bridge implicit induction methods and inductive theorems in higher-order rewriting," IEICE Trans. Inf. & Syst., vol.E88-D, no.12, pp.2715–2726, Dec. 2005.
[21] K. Kusakari and Y. Chiba, "A higher-order Knuth-Bendix procedure and its applications," IEICE Trans. Inf. & Syst., vol.E90-D, no.4, pp.707–715, April 2007.
[22] K. Kusakari and M. Sakai, "Enhancing dependency pair method using strong computability in simply-typed term rewriting systems," Applicable Algebra in Engineering, Communication and Computing, vol.18, no.5, pp.407–431, 2007.
[23] C. Marché and M. Zantema, "The termination competition," Proc. 18th Int. Conf. on Rewriting Techniques and Applications, LNCS 4533 (RTA2007), pp.303–313, 2007.
[24] M. Sakai, Y. Watanabe, and T. Sakabe, "Dependency pair method for proving termination of higher-order rewrite systems," IEICE Trans. Inf. & Syst., vol.E84-D, no.8, pp.1025–1032, Aug. 2001.
[25] M. Sakai and K. Kusakari, "On dependency pair method for proving termination of higher-order rewrite systems," IEICE Trans. Inf. & Syst., vol.E88-D, no.3, pp.583–593, March 2005.
[26] T. Sakurai, K. Kusakari, M. Sakai, T. Sakabe, and N. Nishida, "Usable rules and labeling product-typed terms for dependency pair method in simply-typed term rewriting systems," IEICE Trans. Inf. & Syst. (Japanese Edition), vol.J90-D, no.4, pp.978–989, April 2007.
[27] T.T. Tait, "Intensional interpretation of functionals of finite type," Journal of Symbolic Logic, vol.32, pp.198–212, 1967.
[28] Terese, Term Rewriting Systems, Cambridge Tracts in Theoretical Computer Science, vol.55, Cambridge University Press, 2003.
[29] R. Thiemann, J. Giesl, and P. Schneider-Kamp, "Improved modular termination proofs using dependency pairs," Proc. 2nd Int. Joint Conf. on Automated Reasoning, LNAI 3097 (IJCAR2004), pp.75–90, 2004.
[30] X. Urbain, "Modular & incremental automated termination proofs," J. Automated Reasoning, vol.32, no.4, pp.315–355, 2004.

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 assistant 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.

Masahiko Sakai completed graduate course of Nagoya University in 1989 and became Assistant Professor, where he obtained a D.E. degree in 1992. From April 1993 to March 1997, he was Associate Professor in JAIST. In 1996 he stayed at SUNY at Stony Brook for six months as Visiting Research Professor. From April 1997, he was Associate Professor in Nagoya University. Since December 2002, he has been Professor. He is interested in term rewriting system, verification of specification and software generation. He received the Best Paper Award from IEICE in 1992. He is a member of JSSST.