IEICE TRANS. INF. & SYST., VOL.E96–D, NO.3 MARCH 2013

PAPER Special Section on Foundations of Computer Science

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 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 $R_{\texttt{foldl}}$:

$$\begin{cases} \texttt{foldl}\ f\ e\ \texttt{nil} & \to & e \\ \texttt{foldl}\ f\ e\ (\texttt{cons}\ (x, xs)) & \to & \texttt{foldl}\ f\ (f\ (e, x))\ xs \end{cases}$$

Here we suppose that the function *foldl* has the type:

$$\texttt{foldl} : (\alpha \times \beta \to \alpha) \to \alpha \to \texttt{list}(\beta) \to \alpha$$

in which α and β 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 $R_{\texttt{foldl}}$, the static dependency pair method yields a single static recursion component:

$$\texttt{foldl}^\sharp\ f\ e\ (\texttt{cons}\ (x, xs)) \to \texttt{foldl}^\sharp\ f\ (f\ (e, x))\ xs$$

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:

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

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 $\texttt{foo} : \alpha \times \alpha \to \alpha$, and let R be the following RFP:

$$R_{\texttt{foldl}} \cup \{\texttt{foo}\ (x, y) \to \texttt{foldl}\ \texttt{foo}\ y\ (\texttt{cons}\ (x, \texttt{nil}))\}$$

Then the RFP R is not terminating because there exists the loop: $\texttt{foo}\ (0,0) \underset{R}{\to} \texttt{foldl}\ \texttt{foo}\ 0\ (\texttt{cons}\ (0, \texttt{nil})) \underset{R}{\to} \texttt{foldl}\ \texttt{foo}\ (\texttt{foo}\ (0,0))\ \texttt{nil} \underset{R}{\to} \texttt{foo}\ (0,0)$. As seen above, for the non-termination of R, the infinite sequence through

Manuscript received March 23, 2012.
Manuscript revised June 16, 2012.

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

DOI: 10.1587/transinf.E96.D.472

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 for safe function-passing RFPs, we split static dependency pairs into outer ones and inner ones (cf. Definition 4.1), moreover, we introduce the notion of outer/inner actual static dependency pairs (cf. Definition 4.3). Then we can prove the soundness by a similar story line in [17], although minor adjustments are needed in almost all parts.

As an example showing the effectivity of the static dependency pair method, there exists polymorphic-typed combinatory logic, which is represented as the following RFP R_{CL} with $\mathtt{S} : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$ and $\mathtt{K} : \alpha \to \beta \to \alpha$:

$$R_{\mathrm{CL}} = \left\{ \begin{array}{rcl} \mathtt{S}\ f\ g\ x & \to & f\ x\ (g\ x) \\ \mathtt{K}\ x\ y & \to & x \end{array} \right.$$

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

- If $\alpha \in TV$ then $\alpha \in \mathcal{S}$.
- If $\sigma_1, \sigma_2 \in \mathcal{S}$ then $(\sigma_1 \to \sigma_2) \in \mathcal{S}$.
- If $\sigma_1, \ldots, \sigma_n \in \mathcal{S}$ then $(\sigma_1 \times \cdots \times \sigma_n) \in \mathcal{S}$.
- If $\sigma_1, \ldots, \sigma_n \in \mathcal{S}$ and $c \in TC$ with $\mathit{arity}(c) = n$ then $c(\sigma_1, \ldots, \sigma_n) \in \mathcal{S}$.

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

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

A *type environment* is a pair (Σ, Γ) of mappings $\Sigma : \mathcal{F} \to \mathcal{S}$ and $\Gamma : \mathcal{V} \to \mathcal{S}$. Under an environment (Σ, Γ),

- if $\Gamma(x) = \overline{\sigma_n} \to \sigma_0$ then $(x\ t_1^{\sigma_1}\ \cdots\ t_n^{\sigma_n})^{\sigma_0}$ is a typed term,
- if $\Sigma(f) \geq \overline{\sigma_n} \to \sigma_0$ then $(f\ t_1^{\sigma_1}\ \cdots\ t_n^{\sigma_n})^{\sigma_0}$ is a typed term, and
- $(t_1^{\sigma_1}, \ldots, t_n^{\sigma_n})^{\sigma_1 \times \cdots \times \sigma_n}$ is a typed term,

whenever $t_1^{\sigma_1}, \ldots, t_n^{\sigma_n}$ are typed terms. The identity of typed terms is denoted by $\equiv$. We shortly denote $(a)^{\sigma}$ by a^{σ} for $a \in \mathcal{F} \cup \mathcal{V}$. For a term $t^{\sigma} \equiv (t_1^{\sigma_1}, \ldots, t_n^{\sigma_n})^{\sigma_1 \times \cdots \times \sigma_n}$, we identify $t^{\sigma} \equiv (t_1^{\sigma_1})^{\sigma_1} \equiv t_1^{\sigma_1}$ if $n = 1$, and $t^{\sigma} \equiv ()^{\mathtt{unit}}$ if $n = 0$, where `unit` is the special type constructor with $\mathit{arity}(\mathtt{unit}) = 0$. 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^σ by $t : \sigma$, or shortly t whenever no confusion arises. We abbreviate $(a\ t_1^{\sigma_1}\ \cdots\ t_n^{\sigma_n})^{\sigma_0}$ by $(a\ \overline{t_n^{\sigma_n}})^{\sigma_0}$, or shortly $a\ \overline{t_n}$. We often write $t\ u$ for $(a\ \overline{t_n}\ \overline{u_m^{\sigma_m}})^\sigma$ where $t \equiv (a\ \overline{t_n})^{\overline{\sigma_m}\to\sigma}$.

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

$$\mathtt{map}^{(\alpha\to\beta)\to\mathtt{list}(\alpha)\to\mathtt{list}(\beta)} \in \mathcal{T}$$

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

$$(\mathtt{map}\ \mathtt{map}^{(\alpha\to\beta)\to\mathtt{list}(\alpha)\to\mathtt{list}(\beta)})^\sigma \in \mathcal{T}$$

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

The set of *positions* of a term t is the set $Pos(t)$ of strings over positive integers, which is inductively defined as follows: $Pos(a\ \overline{t_n}) = Pos((t_1, \ldots, t_n)) = \{\varepsilon\} \cup \bigcup_{i=1}^n \{ip \mid p \in Pos(t_i)\}$. 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 subterm at position p in t, denoted by $t|_p$, is defined as $t|_\varepsilon \equiv t$, $(a\ \overline{t_n})|_{ip} \equiv t_i|_p$, and $(t_1, \ldots, t_n)|_{ip} \equiv t_i|_p$. The symbol at position p in t, denoted by $(t)_p$, is defined as $(a\ \overline{t_n})_\varepsilon = a$, $(t_1, \ldots, t_n)_\varepsilon = \mathtt{tp}$, $(a\ \overline{t_n})_{ip} = (t_i)_p$, and $(t_1, \ldots, t_n)_{ip} = (t_i)_p$. Here $\mathtt{tp}$ represents the tuple symbol. To improve readability, we often omit the type information. Sometimes the root symbol $(t)_\varepsilon$ in a term t is denoted by $root(t)$. We also define $args(t)$ by $\{t_1, \ldots, t_n\}$ if t has the form of $a\ \overline{t_n}$ or $(t_1, \ldots, t_n)$. We denote by $Sub(t)$ the set of subterms of t, and by $Var(t)$ the set of variables occurring in t. We write $t \trianglerighteq_{sub} u$ (resp. $t \triangleright_{sub} u$) if $u \in Sub(t)$ (resp. $u \in Sub(t) \setminus \{t\}$). We note that $\overline{\sigma_m} \to \sigma_0 = \overline{\sigma'_n} \to \sigma'_0$ holds whenever $(x\ \overline{u_m^{\sigma_m}})^{\sigma_0}, (x\ \overline{v_n^{\sigma'_n}})^{\sigma'_0} \in Sub(t)$ with $x \in \mathcal{V}$. A term t is said to be *closed* if σ is a closed type for any $u^\sigma \in Sub(t)$. We denote by $\mathcal{T}_{nfun}$ the set of non-functional typed terms, by $\mathcal{T}^{cls}$ the set of closed terms, and by $\mathcal{T}^{cls}_{nfun}$ the set of non-functional and closed terms.

A *context* is a term with one occurrence of the special symbol $\Box^\sigma$, called a hole. A *leaf context* is a context where the occurrence of $\Box$ is at a leaf position. The notation $C[t^\sigma]$ denotes the term obtained by substituting t into the hole of $C[\Box^\sigma]$.

A type substitution ξ is naturally extended over terms as $(a\ \overline{t_n})^\sigma\xi \equiv (a\ \overline{t_n\xi})^{\xi(\sigma)}$ and $(t_1^{\sigma_1}, \ldots, t_n^{\sigma_n})^\sigma\xi \equiv (t_1^{\sigma_1}\xi, \ldots, t_n^{\sigma_n}\xi)^{\xi(\sigma)}$.

A *term substitution* is a mapping with finite domain, denoted by $\theta = \{x_1^{\sigma_1} := t_1^{\sigma_1}, \ldots, x_n^{\sigma_n} := t_n^{\sigma_n}\}$. Each substitution θ is naturally extended over terms as $(a\ \overline{t_n^{\sigma_n}})^\sigma\theta = a'\ \overline{u_k}\ \overline{t_n\theta}$ if $a \in \mathcal{V}$ and $\theta(a^{\overline{\sigma_n}\to\sigma}) = a'\ \overline{u_k}$; $(a\ \overline{t_n})\theta = a\ \overline{t_n\theta}$ if $a \in \mathcal{F}$; $(t_1, \ldots, t_n)\theta = (t_1\theta, \ldots, t_n\theta)$.

A pair (l^σ, r^σ) of terms with the same type under the same type environment is said to be a *rewrite rule*, denoted by $l^\sigma \to r^\sigma$, if $root(l) \in \mathcal{F}$ and $Var(l) \supseteq Var(r)$ hold. We note that the condition $root(l) \in \mathcal{F}$ guarantees that l has the form of $a\ \overline{l_n}$, but not the form of $(l_1, \ldots, l_n)$. A *rewriting system for functional programs* (RFP) is a finite set of rewrite rules. As a matter of course, we note that the rules of an RFP R share a type environment Σ. For any rewrite rule $l^\sigma \to r^\sigma$, we define the set $Act(l \to r)$ of *actual rewrite rules* as: $u^{\sigma'} \to v^{\sigma'} \in Act(l^\sigma \to r^\sigma)$ iff there is a type substitution ξ such that $u \equiv l\xi\ \overline{z_k}$, $v \equiv r\xi\ \overline{z_k}$, and $\xi(\sigma) = \overline{\sigma_n} \to \sigma_0$ with $k \leq n$, and $\sigma' = \sigma_{k+1} \to \cdots \to \sigma_n \to \sigma_0$, where each $z_i^{\sigma_i}$ is a fresh variable. The *reduction relation* $\underset{R}{\to}$ of an RFP R is defined by $s \underset{R}{\to} t$ iff $s \equiv C[l\theta]$ and $t \equiv C[r\theta]$ for some actual rewrite rule $l \to r \in Act(R)$, leaf context $C[\ ]$, and term substitution θ.

Example 2.2: Let $R_{\mathtt{list}}$ be the following RFP:

$$R_{\mathtt{list}} = \left\{ \begin{array}{lcl} \mathtt{hd}\,(\mathtt{cons}\,(x, xs)) & \to & x \\ \mathtt{tl}\,(\mathtt{cons}\,(x, xs)) & \to & xs \end{array} \right.$$

Here we suppose that $\mathtt{nil} : \mathtt{list}(\alpha)$, $\mathtt{cons} : \alpha \times \mathtt{list}(\alpha) \to \mathtt{list}(\alpha)$, $\mathtt{hd} : \mathtt{list}(\alpha) \to \alpha$, and $\mathtt{tl} : \mathtt{list}(\alpha) \to \mathtt{list}(\alpha)$. Then $Act((\mathtt{hd}\ (\mathtt{cons}\ (x^\alpha, xs^{\mathtt{list}(\alpha)})^{\alpha\times\mathtt{list}(\alpha)})^{\mathtt{list}(\alpha)})^\alpha \to x^\alpha)$ consists of the rules that have the following form:

$$(\mathtt{hd}\,(\mathtt{cons}\,(x^\sigma, xs^{\mathtt{list}(\sigma)})^{\sigma\times\mathtt{list}(\sigma)})^{\mathtt{list}(\sigma)}\overline{z_n^{\sigma_n}})^{\sigma_0} \to (x\overline{z_n^{\sigma_n}})^{\sigma_0}$$

where $\sigma = \overline{\sigma_n} \to \sigma_0$ and each σ_i is an arbitrary type. Also, $Act((\mathtt{tl}\ (\mathtt{cons}\ (x^\alpha, xs^{\mathtt{list}(\alpha)})^{\alpha\times\mathtt{list}(\alpha)})^{\mathtt{list}(\alpha)})^{\mathtt{list}(\alpha)} \to xs^{\mathtt{list}(\alpha)})$ consists of the rules that have the following form:

$$(\mathtt{tl}\,(\mathtt{cons}\,(x^\sigma, xs^{\mathtt{list}(\sigma)})^{\sigma\times\mathtt{list}(\sigma)})^{\mathtt{list}(\sigma)})^{\mathtt{list}(\sigma)} \to xs^{\mathtt{list}(\sigma)}$$

where σ is an arbitrary type.

Example 2.3: Let $R_{\mathtt{map}}$ be the following RFP:

$$\left\{ \begin{array}{rcl} \mathtt{map}\ f\ \mathtt{nil} & \to & \mathtt{nil} \\ \mathtt{map}\ f\ (\mathtt{cons}\ (x, xs)) & \to & \mathtt{cons}\ (f\ x,\ \mathtt{map}\ f\ xs) \end{array} \right.$$

We suppose that $\mathtt{map} : (\alpha \to \beta) \to \mathtt{list}(\alpha) \to \mathtt{list}(\beta)$. Here $\mathtt{Nat}$ is a type of natural numbers, which are represented in the usual way by $0 : \mathtt{Nat}$ and $\mathtt{succ} : \mathtt{Nat} \to \mathtt{Nat}$, Then we have the following reduction for $R = R_{\mathtt{list}} \cup R_{\mathtt{map}}$:

$$\begin{aligned} &\mathtt{hd}\,(\mathtt{map}\,\mathtt{map}\,(\mathtt{cons}\,(\mathtt{succ},\ \mathtt{nil})))\,(\mathtt{cons}\,(0, \mathtt{nil})) \\ &\underset{R}{\to} \mathtt{hd}\,(\mathtt{cons}\,(\mathtt{map}\,\mathtt{succ}, \mathtt{map}\,\mathtt{map}\,\mathtt{nil}))\,(\mathtt{cons}\,(0, \mathtt{nil})) \\ &\underset{R}{\to} \mathtt{hd}\,(\mathtt{cons}\,(\mathtt{map}\,\mathtt{succ}, \mathtt{nil}))\,(\mathtt{cons}\,(0, \mathtt{nil})) \\ &\underset{R}{\to} \mathtt{map}\,\mathtt{succ}\,(\mathtt{cons}\,(0, \mathtt{nil})) \\ &\underset{R}{\to} \mathtt{cons}\,(\mathtt{succ}\,0, \mathtt{map}\,\mathtt{succ}\,\mathtt{nil}) \\ &\underset{R}{\to} \mathtt{cons}\,(\mathtt{succ}\,0, \mathtt{nil}) \end{aligned}$$

A term t is *terminating* or *strongly normalizing* if there exists no infinite reduction sequence starting from t. Then we denote $\mathcal{SN}(t)$. An RFP R is said to be *terminating* or *strongly normalizing* if every term is so. We denote by $\mathcal{T}_{SN}$ the set of strongly normalizing terms. We also define sets $\mathcal{T}_{\neg SN} = \mathcal{T} \setminus \mathcal{T}_{SN}$ and $\mathcal{T}^{args}_{SN} = \{t \mid args(t) \subseteq \mathcal{T}_{SN}\}$.

Since actual rewrite rules are closed under type substitution, we obtain the following proposition.

Proposition 2.4: Let R be an RFP. If $s \xrightarrow[R]{} t$ then $s\xi \xrightarrow[R]{} t\xi$ for any type substitution ξ. 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 λ-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.

$$(\mathtt{foo}\ (\mathtt{bar}\ f^{\alpha\to\beta})^{\alpha})^{\beta} \to (f\ (\mathtt{bar}\ f^{\alpha\to\beta})^{\alpha})^{\beta}$$

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 ($\mathtt{bar}\ t$) of $\mathtt{foo}$ 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 $\mathcal{SC}$ over closed terms is said to be a *strong computability predicate* if the following properties hold:

(SC1) For any $t \in \mathcal{T}^{cls}$, if $\mathcal{SC}(t)$ then $\mathcal{SN}(t)$.
(SC2) For any $t^{\sigma_1\to\sigma_2}, u^{\sigma_1} \in \mathcal{T}^{cls}$, if $\mathcal{SC}(t)$ and $\mathcal{SC}(u)$ then $\mathcal{SC}(t\ u)$.
(SC3) For any $t^{\sigma_1\to\sigma_2} \in \mathcal{T}^{cls}$, if $\mathcal{SC}(t\ u)$ for all $u^{\sigma_1} \in \mathcal{T}^{cls}$ such that $\mathcal{SC}(u)$ then $\mathcal{SC}(t)$.
(SC4) For any $t, u \in \mathcal{T}^{cls}$, if $\mathcal{SC}(t)$ and $t \xrightarrow[R]{} u$ then $\mathcal{SC}(u)$.
(SC5) For any $t \in \mathcal{T}^{cls}_{nfun}$, if $\mathcal{SC}(u)$ for all $u \in \mathcal{T}^{cls} \cap (args(t) \cup \{t' \mid t \xrightarrow[R]{} t'\})$ then $\mathcal{SC}(t)$.

We denote $\mathcal{T}_{SC} = \{t \mid \mathcal{SC}(t)\}$, $\mathcal{T}_{\neg SC} = \{t \mid \neg\mathcal{SC}(t)\}$, and $\mathcal{T}^{args}_{SC} = \{t \mid args(t) \subseteq \mathcal{T}_{SC}\}$.

Definition 3.2: For a strong computability predicate $\mathcal{SC}$, a function $\mathcal{S}afe$ is said to be a *safety function* if it satisfies the following properties:

(S1) If $u \in \mathcal{S}afe(t)$ and $t \in \mathcal{T}^{args}_{SC}$ then $\mathcal{SC}(u)$, for any $t, u \in \mathcal{T}^{cls}$.
(S2) If $u \in \mathcal{S}afe(t)$ then $u\theta \in \mathcal{S}afe(t\theta)$ for any $t, u \in \mathcal{T}^{cls}$ and term substitution θ.
(S3) If $u \in \mathcal{S}afe(t)$ then $u\xi \in \mathcal{S}afe(t\xi)$ for any closed type substitution ξ.

Definition 3.3: An RFP R is said to be *safe function-passing* if there exists a safety function $\mathcal{S}afe$ for a strong computability predicate such that for any $l \to r \in R$ and $a\ \overline{r_n} \in Sub(r)$ with $a \in \mathcal{V}$, there exists k ($k \le n$) such that $a\ \overline{r_k} \in \mathcal{S}afe(l)$. A safe function-passing RFP is often shortly denoted by 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 $PT_{\ge} = \{\sigma \mid \sigma' \ge \sigma \text{ for some } \sigma' \in PT\}$. A well-founded quasi order $\gtrsim_{\mathcal{S}}$ on types is said to be a *peeling order* if the following properties hold:

- $\sigma_1 \to \sigma_2 \gtrsim_{\mathcal{S}} \sigma_i$ $(i = 1, 2)$ for any closed types σ_1 and σ_2.
- If $\sigma' \gtrsim_{\mathcal{S}} \sigma$ then $\xi(\sigma') \gtrsim_{\mathcal{S}} \xi(\sigma)$ for any closed type substitution ξ.

We define the set $Sub^{\gtrsim_{\mathcal{S}}}_{PT}(t)$ of *peeled subterms* as the smallest set satisfying the following properties:

- $args(t) \subseteq Sub^{\gtrsim_{\mathcal{S}}}_{PT}(t)$,
- if $u \equiv (a\ \overline{u_n^{\sigma_n}})^{\sigma} \in Sub^{\gtrsim_{\mathcal{S}}}_{PT}(t)$, $\sigma \in PT_{\ge}$, and $\sigma \gtrsim_{\mathcal{S}} \sigma_i$ then $u_i \in Sub^{\gtrsim_{\mathcal{S}}}_{PT}(t)$, and
- if $u \equiv (u_1^{\sigma_1}, \ldots, u_n^{\sigma_n})^{\sigma} \in Sub^{\gtrsim_{\mathcal{S}}}_{PT}(t)$, $\sigma \in PT_{\ge}$, and $\sigma \gtrsim_{\mathcal{S}} \sigma_i$ then $u_i \in Sub^{\gtrsim_{\mathcal{S}}}_{PT}(t)$.

For a set PT of peeling types and a peeling order $\gtrsim_{\mathcal{S}}$, we define the function $\mathcal{S}afe$ as $\mathcal{S}afe(t) = Sub^{\gtrsim_{\mathcal{S}}}_{PT}(t) \cup \{u \mid t \rhd_{sub} u^{\sigma},\ \sigma \text{ is a data type such that } \sigma \notin PT_{\ge}\}$.

Definition 3.5: For a set PT of peeling types and peeling order $\gtrsim_{\mathcal{S}}$, we define $SC(t^{\sigma})$ as follows:

- In case of $t^{\sigma} \in \mathcal{T}^{cls}_{nfun}$ and $\sigma \notin PT_{\ge}$, $SC(t)$ is defined as $\mathcal{SN}(t)$.
- In case of $t^{\sigma} \in \mathcal{T}^{cls}_{nfun}$ and $\sigma \in PT_{\ge}$, $SC(t)$ is defined as $\mathcal{SN}(t)$ and $SC(u)$ for any $u^{\sigma'} \in \mathcal{T}^{cls} \cap \bigcup\{args(t') \mid t \xrightarrow[R]{*} t'\}$ such that $\sigma \gtrsim_{\mathcal{S}} \sigma'$.
- In case of $t^{\sigma_1\to\sigma_2} \in \mathcal{T}^{cls}$, $SC(t)$ is defined as $SC(t\ u)$ for all $u^{\sigma_1} \in \mathcal{T}^{cls}$ 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 $t \in \mathcal{T}^{cls}$. Assume that SC is not well-defined.

Let $t_0^{\sigma_0}$ be a minimal term with respect to $\gtrsim_{\mathcal{S}}$ such that $SC(t_0)$ is undefined. From the minimality of t_0, we have $\sigma_0 \in PT_{\geq}, \mathcal{SN}(t_0)$, and there exist t_0' and t_1 such that $t_0 \xrightarrow[R]{*} t_0'$, $t_1^{\sigma_1} \in args(t_0')$, $\sigma_0 \sim_{\mathcal{S}} \sigma_1$, and $SC(t_1)$ is undefined, where $\sim_{\mathcal{S}}$ is the equivalence part of $\gtrsim_{\mathcal{S}}$.

Since $\sigma_0 \sim_{\mathcal{S}} \sigma_1$, t_1 is also a minimal term with respect to $\gtrsim_{\mathcal{S}}$ such that $SC(t_1)$ is undefined. By applying the procedure above, we obtain t_1' and t_2 such that $t_1 \xrightarrow[R]{*} t_1'$, $t_2^{\sigma_2} \in args(t_1')$, $\sigma_1 \sim_{\mathcal{S}} \sigma_2$, and $SC(t_2)$ is undefined.

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 $\rhd_{sub} \cup \xrightarrow[R]{}$ is well-founded on terminating terms, this contradicts with $\mathcal{SN}(t_0)$.

Next we will prove that the predicate SC satisfies the conditions in Definition 3.1. The conditions (SC2), (SC3), and (SC5) are trivial.

(SC4) Let $SC(t^{\sigma})$ and $t \xrightarrow[R]{} t'$. We prove $SC(t')$ by induction on σ. The case $\sigma \in \mathcal{S}_{nfun}$ is trivial. Suppose that $\sigma = \sigma_1 \to \sigma_2$. Let u^{σ_1} be an arbitrary term such that $SC(u)$ holds. Then $SC(t\ u)$ follows from $SC(t)$ and (SC2). Since $(t\ u)^{\sigma_2} \xrightarrow[R]{} t'\ u$, 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 σ.

(i) If $SC(t^{\sigma})$ then $\mathcal{SN}(t)$.
(ii) $SC(x^{\sigma})$ for all $x \in \mathcal{V}$.

Let $\overline{\sigma_n} \to \sigma_0$ be the canonical form of σ. The case $n = 0$ is trivial. Suppose that $n > 0$.
(i): From the induction hypothesis (ii), an arbitrary variable $z_1^{\sigma_1}$ 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 $\neg SC(z^{\sigma})$ for some variable z. From (SC3), there exist strongly computable terms $u_1^{\sigma_1}, \ldots, u_n^{\sigma_n}$ such that $z\ \overline{u_n}$ is not strongly computable. From the induction hypothesis (i), each u_i is terminating, hence so is $z\ \overline{u_n}$. Since $(z\ \overline{u_n})^{\sigma_0}$ is not strongly computable and $\sigma_0 \in \mathcal{S}_{nfun}$, we have $\sigma_0 \in PT_{\geq}$ and there exist terms v' and v such that $z\ \overline{u_n} \xrightarrow[R]{*} v'$, $v \in args(v')$, and v is not strongly computable. Since $root(l) \notin \mathcal{V}$ for all $l \to r \in R$, there exists i such that $u_i \xrightarrow[R]{*} v$. 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 $t, u \in \mathcal{T}^{cls}$, $u \in Safe(t)$ and $t \in \mathcal{T}_{SC}^{args}$. We prove $SC(u)$.
If $u \in \{u \mid t \rhd_{sub} u^{\sigma},\ \sigma$ is a data type such that $\sigma \notin PT_{\geq}\}$, then $SC(u)$ follows from $u \lhd_{sub} t \in \mathcal{T}_{SC}^{args}$ and (SC1).
Suppose that $u^{\sigma} \in Sub_{PT}^{\gtrsim_{\mathcal{S}}}(t)$. Then we have either $u \in args(t)$ or there exists $v^{\sigma'} \in Sub_{PT}^{\gtrsim_{\mathcal{S}}}(t)$ such that $u \in args(v)$, $\sigma' \in PT_{\geq}$, and $\sigma' \gtrsim_{\mathcal{S}} \sigma$. In the former case, we have $SC(u)$ because of $t\theta \in \mathcal{T}_{SC}^{args}$. 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 $\rhd_{sub}$ is closed under term substitutions, and any term substitution does not change type information.

(S3) It is obvious because $\gtrsim_{\mathcal{S}}$ and $PT_{\geq}$ are closed under closed type substitutions. □

Example 3.8: We show that $R_{\texttt{foldl}}$ 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 $\texttt{tp}_n(\sigma_1, \ldots, \sigma_n)$), we construct the peeling order $\gtrsim_{\mathcal{S}}$ by using the recursive path order $>_{rpo}$ [5] with the argument filtering method [1] over first-order term rewriting systems. We take the argument filtering function by $\pi(\texttt{tp}_n) = n$, $\pi(\to) = [1, 2]$, and $\pi(c) = [1, \ldots, arity(c)]$ for any $c \in TC$. Then the order $\gtrsim_{rpo}^{\pi}$, defined as $\sigma_1 \gtrsim_{rpo}^{\pi} \sigma_2$ iff $\pi(\sigma_1) \gtrsim_{rpo} \pi(\sigma_2)$, becomes a peeling order. We take PT as the set of all data types.

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

- we have $f, e, \texttt{cons}\ (x, xs) \in Sub_{PT}^{\gtrsim_{\mathcal{S}}}(t)$ because of $args(t) \subseteq Sub_{PT}^{\gtrsim_{\mathcal{S}}}(t)$,
- we have $(x, xs) \in Sub_{PT}^{\gtrsim_{\mathcal{S}}}(t)$ because of $(\texttt{cons}\ (x, xs)^{\alpha \times \texttt{list}(\alpha)})^{\texttt{list}(\alpha)} \in Sub_{PT}^{\gtrsim_{\mathcal{S}}}(t)$, $\texttt{list}(\alpha) \in PT$, and $\pi(\texttt{list}(\alpha)) = \texttt{list}(\alpha) \gtrsim_{rpo} \texttt{list}(\alpha) = \pi(\alpha \times \texttt{list}(\alpha))$, and
- we have $x, xs \in Sub_{PT}^{\gtrsim_{\mathcal{S}}}(t)$ because of $(x^{\alpha}, xs^{\texttt{list}(\alpha)})^{\alpha \times \texttt{list}(\alpha)} \in Sub_{PT}^{\gtrsim_{\mathcal{S}}}(t)$, $\alpha \times \texttt{list}(\alpha) \in PT$, $\pi(\alpha \times \texttt{list}(\alpha)) = \texttt{list}(\alpha) \gtrsim_{rpo} \texttt{list}(\alpha) = \pi(\texttt{list}(\alpha))$, and $\pi(\alpha \times \texttt{list}(\alpha)) = \texttt{list}(\alpha) \gtrsim_{rpo} \alpha = \pi(\alpha)$.

Hence we have $Safe(t) = \{f, e, \texttt{cons}\ (x, xs), (x, xs), x, xs\}$, and then the second rule of $R_{\texttt{foldl}}$ also satisfies the desired property. Therefore $R_{\texttt{foldl}}$ 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 HRSs [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 C_R, are *constructors*.

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

A pair $\langle\, l^{\sharp},\ a^{\sharp}\ \overline{r_n} \,\rangle$, denoted by $l^{\sharp} \to a^{\sharp}\ \overline{r_n}$, is said to be an *outer static dependency pair* in R if there exists a rule $l \to a\ \overline{r_n} \in R$ satisfying the following conditions:

- $a \in \mathcal{D}_R$, and
- $a\ \overline{r_k} \notin \mathcal{S}afe(l)$ for all $k\ (\leq n)$.

A pair $\langle\, l^\sharp,\ a^\sharp\ \overline{r_n}\,\rangle$, denoted by $l^\sharp \to a^\sharp\ \overline{r_n}$, is said to be an *inner static dependency pair* in R if there exist a non-empty leaf-context $C[\,]$ and $l \to C[a\ \overline{r_n}] \in R$ 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 $R_{\mathtt{sigma}}$, that is the union of $R_{\mathtt{foldl}}$, $R_{\mathtt{map}}$ and the following rules:

$$\left\{\begin{array}{rcl} \mathtt{add}\ (0, y) & \to & y \\ \mathtt{add}\ (\mathtt{succ}\ x, y) & \to & \mathtt{succ}\ (\mathtt{add}\ (x, y)) \\ \mathtt{sum} & \to & \mathtt{foldl}\ \mathtt{add}\ 0 \\ \mathtt{sigma}\ f\ xs & \to & \mathtt{sum}\ (\mathtt{map}\ f\ xs) \end{array}\right.$$

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

$$\begin{array}{rcl} \mathtt{foldl}^\sharp\ f\ e\ (\mathtt{cons}\ (x, xs)) & \to & \mathtt{foldl}^\sharp\ f\ (f\ (e, x))\ xs \\ \mathtt{sum}^\sharp & \to & \mathtt{foldl}^\sharp\ \mathtt{add}\ 0 \\ \mathtt{sigma}^\sharp\ f\ xs & \to & \mathtt{sum}^\sharp\ (\mathtt{map}\ f\ xs) \end{array}$$

and there are four inner static dependency pairs:

$$\begin{array}{rcl} \mathtt{map}^\sharp\ f\ (\mathtt{cons}\ (x, xs)) & \to & \mathtt{map}^\sharp\ f\ xs \\ \mathtt{add}^\sharp\ (\mathtt{succ}\ x, y) & \to & \mathtt{add}^\sharp\ (x, y) \\ \mathtt{sum}^\sharp & \to & \mathtt{add}^\sharp \\ \mathtt{sigma}^\sharp\ f\ xs & \to & \mathtt{map}^\sharp\ f\ xs \end{array}$$

Definition 4.3: Let R be an SFP-RFP. For any outer static dependency pair $u^\sharp \to v^\sharp$, we define the set $Act(u^\sharp \to v^\sharp)$ of *actual outer static dependency pairs* as: $s^\sharp \to t^\sharp \in Act(u^\sharp \to v^\sharp : \sigma)$ iff $s^\sharp$ and $t^\sharp$ are closed terms, and there is a type substitution ξ such that $s^\sharp \equiv u^\sharp\xi\ \overline{z_n}$, $t^\sharp \equiv v^\sharp\xi\ \overline{z_n}$, and the canonical form of $\xi(\sigma)$ is $\overline{\tau_n} \to \tau$, where each $z_i^{\tau_i}$ is a fresh variable.

For any inner static dependency pair $u^\sharp \to v^\sharp$, we define the set $Act(u^\sharp \to v^\sharp)$ of *actual inner static dependency pairs* as: $s^\sharp \to t^\sharp \in Act(u^\sharp : \sigma' \to v^\sharp : \sigma)$ iff $s^\sharp$ and $t^\sharp$ are closed terms, and there is a type substitution ξ such that $s^\sharp \equiv u^\sharp\xi\ \overline{z'_n}$, $t^\sharp \equiv v^\sharp\xi\ \overline{z_m}$, and the canonical form of $\xi(\sigma')$ and $\xi(\sigma)$ are $\overline{\tau'_n} \to \tau'$ and $\overline{\tau_m} \to \tau$, respectively, where each $z'_i : \tau'_i$ and $z_i : \tau_i$ are fresh variables.

An *actual static dependency pair* in R is an actual outer/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 $u_1^\sharp \to v_1^\sharp, u_2^\sharp \to v_2^\sharp, \ldots$ of static dependency pairs in R is said to be a *static dependency chain* in R if there exist $s_1^\sharp \to t_1^\sharp \in Act(u_1^\sharp \to v_1^\sharp)$, $s_2^\sharp \to t_2^\sharp \in Act(u_2^\sharp \to v_2^\sharp)$, ..., and term substitutions $\theta_1, \theta_2, \ldots$ such that for any i, $t_i^\sharp\theta_i \xrightarrow[R]{*} s_{i+1}^\sharp\theta_{i+1}$, $s_i\theta_i, t_i\theta_i \in \mathcal{T}_{SC}^{args}$, and $s_i\theta_i, t_i\theta_i \notin \mathcal{T}_{SC}$.

Lemma 4.5: If an SFP-RFP R is not terminating then $\mathcal{T}_{nfun}^{cls} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}_{SC}^{args} \neq \emptyset$.

Proof.: From Proposition 2.4, we have $\mathcal{T}^{cls} \cap \mathcal{T}_{\neg SN} \neq \emptyset$. From (SC1), we have $\mathcal{T}^{cls} \cap \mathcal{T}_{\neg SC} \neq \emptyset$.

Let s be a minimal term in $\mathcal{T}^{cls} \cap \mathcal{T}_{\neg SC}$ with respect to term size. From the minimality, we have $s \in \mathcal{T}_{SC}^{args}$. Hence, we have $\mathcal{T}^{cls} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}_{SC}^{args} \neq \emptyset$.

Let t^σ be a minimal term in $\mathcal{T}^{cls} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}_{SC}^{args}$ with respect to type size. It suffices to show that $t \in \mathcal{T}_{nfun}$. Assume that $\sigma = \sigma_1 \to \sigma_2$. From (SC3), there is $u^{\sigma_1} \in \mathcal{T}^{cls} \cap \mathcal{T}_{SC}$ such that $(t\ u)^{\sigma_2} \in \mathcal{T}_{\neg SC}$. From $t \in \mathcal{T}_{SC}^{args}$, we have $(t\ u)^{\sigma_2} \in \mathcal{T}_{SC}^{args}$. Since the size σ_2 is less than the size $\sigma_1 \to \sigma_2$, we have $(t\ u) \notin \mathcal{T}^{cls} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}_{SC}^{args}$. It is a contradiction. □

Lemma 4.6: Let R be an SFP-RFP. If $t^\sigma \in \mathcal{T}^{cls} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}_{SC}^{args}$ then $root(t) \in \mathcal{D}_R$.

Proof.: Assume that $root(t) \notin \mathcal{D}_R$. Let $\overline{\sigma_n} \to \sigma_0$ be the canonical form of σ. From (SC3), there are $u_1^{\sigma_1}, \ldots, u_n^{\sigma_n}$ such that $\forall i.SC(u_i)$ and $\neg SC(t\ \overline{u_n})$. Then the termination of $t\ \overline{u_n}$ follows from $root(t) \notin \mathcal{D}_R$, $t\ \overline{u_n} \in \mathcal{T}_{SC}^{args}$ and (SC1).

From (SC5) there exists $t_1 \in \mathcal{T}^{cls}$ such that $t\ \overline{u_n} \xrightarrow[R]{} t_1$ with $\neg SC(t_1)$, because $t\ \overline{u_n} \in \mathcal{T}_{nfun}^{cls} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}_{SC}^{args}$. From $root(t\ \overline{u_n}) \notin \mathcal{D}_R$ and (SC4), we have $t_1 \in \mathcal{T}_{nfun}^{cls} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}_{SC}^{args}$ and $root(t_1) \notin \mathcal{D}_R$. In a similar way, there is a $t_2 \in \mathcal{T}_{nfun}^{cls} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}_{SC}^{args}$ such that $t_1 \xrightarrow[R]{} t_2$ and $root(t_2) \notin \mathcal{D}_R$. By applying this procedure repeatedly, we construct an infinite sequence $t\ \overline{u_n} \xrightarrow[R]{} t_1 \xrightarrow[R]{} t_2 \xrightarrow[R]{} \cdots$, which leads to a contradiction with the termination of $t\ \overline{u_n}$. □

Lemma 4.7: Let R be an SFP-RFP. If $t \in \mathcal{T}_{SN} \cap \mathcal{T}_{nfun}^{cls} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}_{SC}^{args}$ then there exist $l \to r \in Act(R)$ and θ' such that $t^\sharp \xrightarrow[R]{*} l^\sharp\theta'$, $l\theta' \in \mathcal{T}_{nfun}^{cls} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}_{SC}^{args}$, and $r\theta' \in \mathcal{T}_{\neg SC}$.

Proof.: We proceed by induction on t ordered by $\xrightarrow[R]{}$. From $t \in \mathcal{T}_{SC}^{args}$ and (SC5), there exist $t' \in \mathcal{T}_{\neg SC}$ such that $t \xrightarrow[R]{} t'$. In the case where a root redex is rewritten in $t \xrightarrow[R]{} t'$, there exist $l \to r \in Act(R)$ and θ' such that $t \equiv l\theta' \xrightarrow[R]{} r\theta' \equiv t'$. Then the desired property holds. In other cases, since $t' \in \mathcal{T}_{SC}^{args}$ follows from (SC4), the desired property follows from the induction hypothesis. □

Lemma 4.8: Let R be an SFP-RFP. For any $t \in \mathcal{T}_{nfun}^{cls} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}_{SC}^{args}$, there exist $u^\sharp \to v^\sharp \in Act(SDP(R))$ and term substitution θ such that $t^\sharp \xrightarrow[R]{*} u^\sharp\theta$ and $u\theta, v\theta \in \mathcal{T}_{nfun}^{cls} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}_{SC}^{args}$.

Proof.: Let $t^{\sigma'} \in \mathcal{T}_{nfun}^{cls} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}_{SC}^{args}$. Then $t \in \mathcal{T}_{SN}^{args}$ follows from $t \in \mathcal{T}_{SC}^{args}$ and (SC1).

- Consider the case of $t \notin \mathcal{T}_{SN}$. Since $t \in \mathcal{T}_{SN}^{args}$, there exist closed rewrite rule $l \to r \in Act(R)$ and closed term substitution θ' such that $t^\sharp \xrightarrow[R]{*} l^\sharp\theta'$ and $l\theta', r\theta' \in \mathcal{T}_{\neg SN}$. From (SC1) and (SC4), we have $l\theta' \in \mathcal{T}_{nfun}^{cls} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}_{SC}^{args}$ and $r\theta' \in \mathcal{T}_{\neg SC}$.

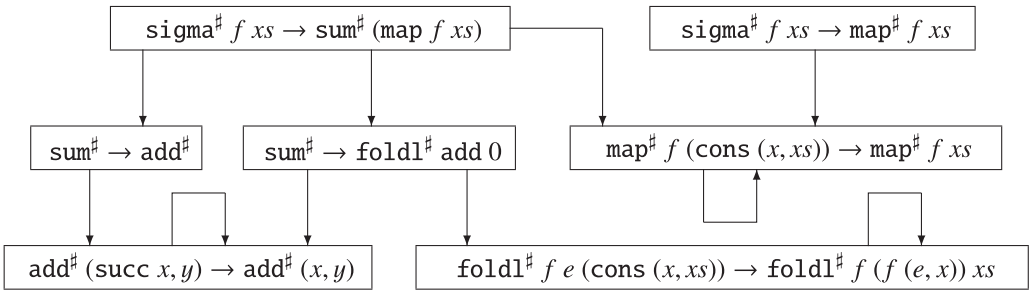

Fig. 1 The static dependency graph of $R_{\mathtt{sigma}}$.

- Consider the case of $t \in \mathcal{T}_{SN}$. From Lemma 4.7, there exist closed rewrite rule $l \to r \in Act(R)$ and closed term substitution θ' such that $t^\sharp \xrightarrow[R]{*} l^\sharp\theta'$, $l\theta' \in \mathcal{T}^{cls}_{nfun} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}^{args}_{SC}$, and $r\theta' \in \mathcal{T}_{\neg SC}$.

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

Now take v by $a\ \overline{r_n}\ \overline{z_m}$, where each $z_i^{\sigma_i}$ is a fresh variable, and $\theta(x)$ is defined by v_i if $x = z_i$ $(i = 1, \ldots, m)$; otherwise by $\theta'(x)$. Then we have $l\theta \equiv l\theta'$ and $v\theta \equiv v'\theta'\ \overline{v_m}$. Hence we have $t^\sharp \xrightarrow[R]{*} l^\sharp\theta$ and $l\theta, v\theta \in \mathcal{T}^{cls}_{nfun} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}^{args}_{SC}$. It suffices to show that $l^\sharp \to v^\sharp \in Act(SDP(R))$.

Since $l \to r \in Act(R)$, there exist $f\ \overline{l_q} \to r' \in R$ and ξ such that $l \equiv f\ \overline{l_q\xi}\ \overline{y_k}$ and $r \equiv r'\xi\ \overline{y_k}$ where each y_i is a fresh variable.

Consider the case of $v' \equiv r$. Then r' has the form of $a\ \overline{r'_{n-k}}$ such that $r_i \equiv r'_i\xi$ for any $i = 1, \ldots, n-k$, and $r_{k+i} \equiv y_i$ for any $i = n-k+1, \ldots, k$. Assume $a\ \overline{r'_p} \in \mathcal{S}afe(l')$ for some $p\ (\leq n-k)$. From (S3), we have $a\ \overline{r_p} \in \mathcal{S}afe(l)$. From (S1), (S2) and $l\theta \in \mathcal{T}^{args}_{SC}$, we have $\mathcal{SC}(a\ \overline{r_p\theta})$. From (SC2), we have $\mathcal{SC}(v\theta)$, which leads to a contradiction. Hence $l^\sharp \to v^\sharp$ is an actual outer static dependency pair.

Consider the case of $v' \equiv y_i$ for some $i \leq k$. Since v' has the form of $a\ \overline{r_n}$, this is not the case.

Consider the case of $v' \in Sub(r'\xi) \setminus \{r'\xi\}$. Then there exists $a\ \overline{r'_n} \in Sub(r')$ such that $v' \equiv a\ \overline{r_n} \equiv a\ \overline{r'_n\xi}$. Assume $a\ \overline{r'_p} \in \mathcal{S}afe(l')$ for some $p\ (\leq n)$. From (S3), we have $a\ \overline{r_p} \in \mathcal{S}afe(l)$. From (S1), (S2) and $l\theta \in \mathcal{T}^{args}_{SC}$, we have $\mathcal{SC}(a\ \overline{r_p\theta})$. From (SC2), we have $\mathcal{SC}(v\theta)$, which leads to a contradiction. Hence $l^\sharp \to v^\sharp$ is an actual inner static dependency pair. □

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

Theorem 4.9: Let R be an SFP-RFP. If there exists no infinite static dependency chain then R is terminating.

Proof.: Assume that R is not terminating. From Lemma 4.5, there exists $t \in \mathcal{T}^{cls}_{nfun} \cap \mathcal{T}_{\neg SC} \cap \mathcal{T}^{args}_{SC}$. By applying Lemma 4.8 repeatedly, we have an infinite static dependency chain, which leads to a contradiction. □

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 $SDP(R)$ 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.

Example 4.11: The static dependency graph of the SFP-RFP $R_{\mathtt{sigma}}$ (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 $SRC(R)$ we denote the set of static recursion components in R.

Example 4.13: The static dependency graph of $R_{\mathtt{sigma}}$ (cf. Example 4.11) has three strongly connected subgraphs. Thus, the set $SRC(R_{\mathtt{sigma}})$ consists of the following three components:

$\{\mathtt{add}^\sharp\ (\mathtt{succ}\ x, y) \to \mathtt{add}^\sharp\ (x, y)\}$,
$\{\mathtt{map}^\sharp\ f\ (\mathtt{cons}\ (x, xs)) \to \mathtt{map}^\sharp\ f\ xs\}$,
$\{\mathtt{foldl}^\sharp\ f\ e\ (\mathtt{cons}\ (x, xs)) \to \mathtt{foldl}^\sharp\ f\ (f\ (e, x))\ xs\}$

Definition 4.14: Let R be an SFP-RFP. A static recursion component $C \in SRC(R)$ is said to be *non-looping* if there exists no infinite static dependency chain in which only pairs in C occur and every $u^\sharp \to v^\sharp \in C$ occurs infinitely many times.

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 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 the notion of the weak-reduction orders [1]. In this section, we extend the notions to RFPs.

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

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

In particular, $\gtrsim$ is said to be a *weak reduction order* if $(\gtrsim, \gtrsim \setminus \lesssim)$ 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 π from $\mathcal{D}_R$ to non-empty sequences of positive integers such that:

(i) $u|_{\pi(root(u))} \rhd_{sub} v|_{\pi(root(v))}$ for some $u^\sharp \to v^\sharp \in C$, and
(ii) the following conditions hold for any $u^\sharp \to v^\sharp \in C$:

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

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

- There is a reduction pair $(\gtrsim, >)$ such that $R \subseteq \gtrsim$, $Act(C) \subseteq \gtrsim \cup >$, and $Act(u^\sharp \to v^\sharp) \subseteq >$ for some $u^\sharp \to v^\sharp \in C$.
- C satisfies the subterm criterion.

Proof.: Assume that there exists an infinite static dependency chain $u_0^\sharp \to v_0^\sharp, u_1^\sharp \to v_1^\sharp, u_2^\sharp \to v_2^\sharp, \cdots$ in which only pairs in C occur and every $u^\sharp \to v^\sharp \in C$ occurs infinitely many times. From the definition of the static dependency chain, there are $s_1^\sharp \to t_1^\sharp \in Act(u_1^\sharp \to v_1^\sharp)$, $s_2^\sharp \to t_2^\sharp \in Act(u_2^\sharp \to v_2^\sharp), \ldots$, and term substitutions $\theta_1, \theta_2, \ldots$ such that for any i, $t_i^\sharp\theta_i \xrightarrow[R]{*} s_{i+1}^\sharp\theta_{i+1}$, $s_i\theta_i, t_i\theta_i \in \mathcal{T}_{SC}^{args}$, and $s_i\theta_i, t_i\theta_i \notin \mathcal{T}_{SC}$.

- Suppose that there is a reduction pair $(\gtrsim, >)$ such that $R \subseteq \gtrsim$, $Act(C) \subseteq \gtrsim \cup >$, and $Act(u^\sharp \to v^\sharp) \subseteq >$ for some $u^\sharp \to v^\sharp \in C$. Since $\gtrsim$ is closed under contexts, type substitutions and term substitutions, $\xrightarrow[R]{*} \subseteq \gtrsim$ follows from $R \subseteq \gtrsim$. Since $>$ is closed term substitutions, $s_i^\sharp\theta_i(\gtrsim \cup >)t_i^\sharp\theta_i$ for any i follows from $Act(C) \subseteq \gtrsim \cup >$. Hence we have $s_0^\sharp\theta_0(\gtrsim\cup>)t_0^\sharp\theta_0 \gtrsim s_1^\sharp\theta_1(\gtrsim\cup>)t_1^\sharp\theta_1 \gtrsim \cdots$. From $Act(u^\sharp \to v^\sharp) \subseteq >$ for some $u^\sharp \to v^\sharp \in C$, this sequence contains infinitely many $>$. This is a contradiction with the well-foundedness of $>$ and $\gtrsim \cdot > \subseteq >$ or $> \cdot \gtrsim \subseteq >$.
- Suppose that C satisfies the subterm criterion. We note that $u|_p \trianglerighteq_{sub} v|_p$ (resp. $u|_p \rhd_{sub} v|_p$) guarantees $s|_p \trianglerighteq_{sub} t|_p$ (resp. $s|_p \rhd_{sub} t|_p$) for any static dependency pair $u^\sharp \to v^\sharp$ and $s^\sharp \to t^\sharp \in Act(u^\sharp \to v^\sharp)$.
  We denote $\pi(root(u_i))$ by p_i for each i. Since $t_i^\sharp\theta_i \xrightarrow[R]{*} s_{i+1}^\sharp\theta_{i+1}$, we have $root(t_i) = root(s_{i+1})$. From the last two conditions in (ii) of the subterm criterion, we have $t_i\theta_i|_{p_{i+1}} \xrightarrow[R]{*} s_{i+1}\theta_{i+1}|_{p_{i+1}}$ for each i. Hence, from the first condition in (ii) of the subterm criterion, we have $s_0\theta_0|_{p_0} \trianglerighteq_{sub} t_0\theta_0|_{p_1} \xrightarrow[R]{*} s_1\theta_1|_{p_1} \trianglerighteq_{sub} t_1\theta_1|_{p_2} \xrightarrow[R]{*} s_2\theta_2|_{p_2} \trianglerighteq_{sub} t_2\theta_1|_{p_3} \xrightarrow[R]{*} \cdots$. From the condition (i) of the subterm criterion, this sequence contains infinitely many $\rhd_{sub}$. Since $\rhd_{sub}$ is well-founded and $\rhd_{sub} \cdot \xrightarrow[R]{} \subseteq \xrightarrow[R]{} \cdot \rhd_{sub}$, there exists an infinite rewriting relation starting from $s_0\theta_0|_{p_0}$, that is, $s_0\theta_0|_{p_0}$ is not terminating. Since p_0 is non-empty, we have $s_0\theta_0 \notin \mathcal{T}_{SN}^{args}$. From the definition of the static dependency chain, we have $s_0\theta_0 \in \mathcal{T}_{SC}^{args}$. From (SC1), we have $s_0\theta_0 \in \mathcal{T}_{SN}^{args}$, which leads to a contradiction. □

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

$$\{\mathtt{add}^\sharp\ (\underline{\mathtt{succ}\ x}, y) \to \mathtt{add}^\sharp\ (\underline{x}, y)\},$$
$$\{\mathtt{map}^\sharp\ f\ \underline{(\mathtt{cons}\ (x, xs))} \to \mathtt{map}^\sharp\ f\ \underline{xs}\},$$
$$\{\mathtt{foldl}^\sharp\ f\ e\ \underline{(\mathtt{cons}\ (x, xs))} \to \mathtt{foldl}^\sharp\ f\ (f\ (e, x))\ \underline{xs}\}$$

Hence, from Theorem 5.3 these static recursion components are non-looping. Therefore the termination of $R_{\mathtt{sigma}}$ 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:

$$\begin{cases} (\mathtt{S}\ f^{\alpha\to\beta\to\gamma}\ g^{\alpha\to\beta}\ x^\alpha)^\gamma & \to & f\ x\ (g\ x) \\ (\mathtt{K}\ x^\alpha\ y^\beta)^\alpha & \to & x \\ (\mathtt{I}\ x^\alpha)^\alpha & \to & x \\ (\mathtt{B}\ x^{\alpha\to\beta}\ y^{\gamma\to\alpha}\ z^\gamma)^\beta & \to & x\ (y\ z) \\ (\mathtt{B'}\ x^{\alpha\to\beta}\ y^{\beta\to\gamma}\ z^\alpha)^\gamma & \to & y\ (x\ z) \\ (\mathtt{C}\ x^{\alpha\to\beta\to\gamma}\ y^\beta\ z^\alpha)^\gamma & \to & x\ z\ y \\ (\mathtt{J}\ x^{\alpha\to\beta\to\beta}\ y^\alpha\ z^\beta\ w^\alpha)^\beta & \to & x\ y\ (x\ w\ z) \\ (\mathtt{W}\ x^{\alpha\to\alpha\to\beta}\ y^\alpha)^\beta & \to & x\ y\ y \end{cases}$$

Since any variable occurs in an argument position on the left-hand sides, R is trivially safe function-passing. Since $SDP(R) = \emptyset$ and hence $SRC(R) = \emptyset$, 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

[1] T. Arts and J. Giesl, "Termination of term rewriting using dependency pairs," Theor. Comput. Sci., vol.236, pp.133–178, 2000.
[2] F. Blanqui, J.-P. Jouannaud, and M. Okada, "Inductive-data-type systems," Theor. Comput. Sci., vol.272, pp.41–68, 2002.
[3] F. Blanqui, "Computability closure: Ten years later," in Essay in Honour of Jean-Pierre Jouannaud's 60 Birthday, LNCS 4600, pp.68–88, 2007.
[4] F. Blanqui, J.-P. Jouannaud, and A. Rubio, "The computability path ordering: The end of a quest," Proc. 17th EACSL Annual Conf. on Computer Science Logic (CSL2008), LNCS 5213, pp.1–14, 2008.
[5] N. Dershowitz, "Orderings for term-rewriting systems," Theor. Comput. Sci., vol.17, no.3, pp.279–301, 1982.
[6] 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.
[7] J.-Y. Girard, "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur," PhD thesis, University of Paris VII, 1972.
[8] J.R. Hindley and J.P. Seldin, Introduction to Combinators and λ-Calculus, Cambridge Univ. Press, 1986.
[9] N. Hirokawa and A. Middeldorp, "Tyrolean termination tool: Techniques and features," Inf. Comput., vol.205, no.4, pp.474–511, 2007.
[10] J.-P. Jouannaud and M. Okada, "A computation model for executable higher-order algebraic specification languages," Proc. LICS'91, pp.350–361, 1991.
[11] J.-P. Jouannaud and A. Rubio, "Polymorphic higher-order recursive path orderings," JACM, vol.54, no.1, pp.1–48, 2007.
[12] J.W. Klop, "Combinatory reduction systems," PhD thesis, Utrecht Universiteit, The Netherlands, 1980. (Published as Mathematical Center Tract 129.)
[13] K. Bimbó, Combinatory Logic: Pure, Applied and Typed, Chapman and Hall/CRC, 2011.
[14] K. Kusakari, "On proving termination of term rewriting systems with higher-order variables," IPSJ Transactions on Programming, vol.42, no.SIG 7 (PRO 11), pp.35–45, 2001.
[15] K. Kusakari, M. Nakamura, and Y. Toyama, "Elimination transformations for associative-commutative rewriting systems," J. Automated Reasoning, vol.37, no.3, pp.205–229, 2006.
[16] 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.
[17] K. Kusakari and M. Sakai, "Static dependency pair method for simply-typed term rewriting and related techniques," IEICE Trans. Inf. & Syst., vol.E92-D, no.2, pp.235–247, Feb. 2009.
[18] K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui, "Static dependency pair method based on strong computability for higher-order rewrite systems," IEICE Trans. Inf. & Syst., vol.E92-D, no.10, pp.2007–2015, Oct. 2009.
[19] N. Nipkow, "Higher-order critical pairs," Proc. 6th Annual IEEE Symposium on Logic in Computer Science, pp.342–349, 1991.
[20] V. van Oostrom, "Confluence for abstract and higher-order rewriting," PhD thesis, Vrije Universiteit Amsterdam, The Netherlands, 1994.
[21] 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.
[22] S. Suzuki, K. Kusakari, and F. Blanqui, "Argument filterings and usable rules in higher-order rewrite systems," IPSJ Transactions on Programming, vol.4, no.2, pp.1–12, 2011.
[23] W.W. Tait, "Intensional interpretation of functionals of finite type," J. Symbolic Logic, vol.32, pp.198–212, 1967.
[24] Terese, Term rewriting systems, Cambridge Tracts in Theoretical Computer Science, vol.55, Cambridge University Press, 2003.

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.