PAPER Special Section on Formal Approaches

Static Dependency Pair Method in Functional Programs

Keiichirou KUSAKARI[†a)], ***Member***

SUMMARY We have previously introduced the static dependency pair method that proves termination by analyzing the static recursive structure of various extensions of term rewriting systems for handling higher-order functions. The key is to succeed with the formalization of recursive structures based on the notion of strong computability, which is introduced for the termination of typed λ-calculi. To bring the static dependency pair method close to existing functional programs, we also extend the method to term rewriting models in which functional abstractions with patterns are permitted. Since the static dependency pair method is not sound in general, we formulate a class; namely, accessibility, in which the method works well. The static dependency pair method is a very natural reasoning; therefore, our extension differs only slightly from previous results. On the other hand, a soundness proof is dramatically difficult.

key words: *functional program, term rewriting system, termination, recursive definition, static dependency pair method*

1. Introduction

The static dependency pair method (SDP-method) [13], [16]–[19], [22] is a powerful method to prove the termination of various extensions of term rewriting systems (TRSs) [24] for handling higher-order functions. The method shows the termination by analyzing a static recursive structure. The principle of the SDP-method is such that if any recursion is suitably defined, then it must terminate. To bring the method close to existing functional programs, we extend the method to term rewriting models for functional programs (TRFPs) in which functional abstractions with patterns are permitted.

We first consider primitive recursion in higher-order settings:

$$\mathrm{prec} : \mathrm{Nat} \to \alpha \to (\mathrm{Nat} \to \alpha \to \alpha) \to \alpha$$

The function can be represented by the following TRFP:

$$\begin{cases} \mathrm{prec}\ 0\ z\ f \to z \\ \mathrm{prec}\ (\mathrm{suc}\ x)\ z\ f \to f\ x\ (\mathrm{prec}\ x\ z\ f) \end{cases}$$

Although it is well-known that the Ackermann function $\mathrm{ack} : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}$ is not primitively recursive in first-order settings, the following TRFP can represent the function by using higher-order primitive recursion twice:

$$\begin{cases} \mathrm{iter}\ f\ x \to \mathrm{prec}\ x\ (f\ (\mathrm{suc}\ 0))\ (\mathsf{fn}\ x' \Rightarrow \mathsf{fn}\ z \Rightarrow f\ z) \\ \mathrm{ack}\ x \to \mathrm{prec}\ x\ \mathrm{suc}\ (\mathsf{fn}\ x' \Rightarrow \mathsf{fn}\ f \Rightarrow \mathrm{iter}\ f) \end{cases}$$

In a functional programming way, we implement programs by defining functions. Hence, the termination of functional programs means that all defined functions are totally defined.

The SDP-method first analyzes a static recursive structure. For example, the method reveals that there is only one recursion,

$$\mathrm{prec}^\sharp\ (\mathrm{suc}\ x)\ z\ f \to \mathrm{prec}^\sharp\ x\ z\ f$$

called the static recursion component, in the TRFP that consists of the above four rules. Then the SDP-method proves the termination by proving the non-loopingness of the static recursion components. In this example, we can prove the non-loopingness because the function "prec" is appropriately recursively programmed on data types, that is, on the first argument in the underlined position below.

$$\mathrm{prec}^\sharp\ (\underline{\mathrm{suc}\ x})\ z\ f \to \mathrm{prec}^\sharp\ \underline{x}\ z\ f$$

By recapitulating such a termination proof of the SDP-method, we obtain the following assertion:

> The function that is appropriately recursively programmed is totally defined.

Although it may be very natural reasoning, the assertion is not correct in general; therefore, the SDP-method is not sound in general, either. The meaning of the assertion is such that:

> Any part other than the recursive parts never destroy the termination.

However, we consider a counterexample to this assertion. For example, Combinatory Logic, which can be represented as:

$$\begin{cases} \mathrm{S}\ f\ g\ x & \to & f\ x\ (g\ x) \\ \mathrm{K}\ x\ y & \to & x \end{cases}$$

is non-terminating [9], but there exist no recursive structures because combinators S and K do not occur on the right-hand sides. On the other hand, typed Combinatory Logic is terminating [9].

Why is untyped Combinatory Logic non-terminating while typed Combinatory Logic is terminating? From a technical viewpoint, we can introduce the notion of strong

Manuscript received August 21, 2017.
Manuscript revised January 28, 2018.
Manuscript publicized March 16, 2018.

[†]The author is with Informatics Course, Department of Electrical, Electronic and Computer Engineering, Faculty of Engineering, Gifu University, Gifu-shi, 501–1193 Japan.
a) E-mail: kurasaki@gifu-u.ac.jp

DOI: 10.1587/transinf.2017FOP0004

computability in typed systems. The notion was introduced to show the termination of typed λ-calculi [8], [23]. Because the notion is inductively defined on types, it is well-defined on typed systems, but not on untyped systems. We note that a theoretical basis for our SDP-method is also given by the notion of strong computability.

Therefore, we may assume that the SDP-method is sound in typed systems. However there exists the following counterexample:

$$\text{foo (bar } f) \to f \text{ (bar } f)$$

Although this system is typable under foo, $f : \alpha \to \beta$ and bar : $(\alpha \to \beta) \to \alpha$, it has the self-loop: foo (bar foo) $\to$ foo (bar foo), and hence, is non-terminating. On the other hand, the SDP-method mistakenly reveals that no recursive structure exists, and hence, is terminating, because the function "foo" does not occur on the right-hand side. From a technical viewpoint, the problem arises because strong computability is not closed under the subterm relation. More precisely, if a receiving argument (bar t) is strongly computable in evaluating the function "foo", the subterm t of (bar t), might not be strongly computable, and t is passed to the right-hand side. In this paper, we formalize this condition as accessibility (cf. Definition 4.3), which guards such passing, and hence, guarantees strong computability of any term that is passed to the right-hand side through variables. We also prove the soundness of the SDP-method in the class of accessible TRFPs.

We note that our SDP-method can prove the termination of polymorphic-typed Combinatory Logic using the following two easily checked reasons:

- S and K do not occur on the right-hand sides.
- Any variable occurs in an argument position on the left-hand sides.

Although several proofs of the termination of polymorphic-typed Combinatory Logic are known [9], we believe that our proof is very elegant, despite permitting functional abstraction with patterns.

As discussed previously, the SDP-method is very natural reasoning so that our extension in this paper may differ only slightly [19]. On the other hand, the soundness proof for the SDP-method is dramatically difficult. To show soundness, it is necessary to wholly rebuild the soundness proof in [19] (cf. the last half of Sect. 5). From a technical viewpoint, our soundness proof is an extension of the termination proof of typed λ-calculi by using the notion of strong computability. Understandably, a try of such extension is broken. Under the restriction of accessibility, our soundness proof characterizes the first break points of the try and then a recursive structure, which generates an infinite reduction, emerges by bridging these points.

The remainder of this paper is organized as follows. Section 2 provides term rewriting models for functional programs (TRFPs) in which functional abstractions with patterns are permitted. In Sect. 3, we present ways for the technical decomposition of terms. In Sect. 4, we discuss the notion of strong computability, which provides a theoretical basis for the SDP-method. We also show the class of accessible TRFPs in which the SDP-method is sound. In Sect. 5, we show the SDP-method on TRFPs. In Sect. 6, we introduce the notion of the subterm criterion and reduction pairs that prove the non-loopingness of static recursion components. Concluding remarks are presented in Sect. 7.

2. Term Rewriting Model for Functional Programs

We introduced term rewriting models for functional programs (TRFPs) [19], as an extension of term rewriting systems [24]. In this paper, we extend TRFP to allow functional abstraction with pattern. For simplicity, we use the short notation $\overline{a_n}$ for a sequence of either $a_1, \ldots, a_n$ or $a_1 \ \cdots \ a_n$.

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 $\mathsf{arity}(c) = n$. 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 $\mathsf{arity}(c) = n$ then $c(\sigma_1, \ldots, \sigma_n) \in \mathcal{S}$.

A *functional type* or *higher-order type* is a type of the form $(\sigma_1 \to \sigma_2)$. We denote by $\mathcal{S}_{nfun}$ the set of non-functional types. 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 $c(\sigma_1, \ldots, \sigma_n)$. For $c \in TC$ with $\mathsf{arity}(c) = 0$, we simply denote $c()$ by c. 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 $\sigma_0 \in \mathcal{S}_{nfun}$, 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 $\sigma = \xi(\sigma')$.

Let $\mathcal{D}$, $\mathcal{C}$, and $\mathcal{V}$ be a set of *defined symbols*, *constructors*, and *term variables*, respectively. A *type environment* is a pair (Σ, Γ) of functions $\Sigma : \mathcal{C} \cup \mathcal{D} \to \mathcal{S}$ and $\Gamma : \mathcal{V} \to \mathcal{S}$. For given type environment (Σ, Γ), we define the set $\mathcal{P}(\Sigma, \Gamma)$ of *patterns* as follows:

- If $\Gamma(x) = \sigma$ then $x^\sigma \in \mathcal{P}(\Sigma, \Gamma)$.
- If $c \in \mathcal{C}$, $\Sigma(c) \geq \overline{\sigma_n} \to \sigma_0$ and $p_1^{\sigma_1}, \ldots, p_n^{\sigma_n} \in \mathcal{P}(\Sigma, \Gamma)$ then $(c\ p_1^{\sigma_1} \ \cdots \ p_n^{\sigma_n})^{\sigma_0} \in \mathcal{P}(\Sigma, \Gamma)$.
- If $p_1^{\sigma_1}, \ldots, p_n^{\sigma_n} \in \mathcal{P}(\Sigma, \Gamma)$ then $(p_1^{\sigma_1}, \ldots, p_n^{\sigma_n})^{\sigma_1 \times \cdots \times \sigma_n} \in \mathcal{P}(\Sigma, \Gamma)$.

For given type environment (Σ, Γ), we define the set $\mathcal{T}(\Sigma, \Gamma)$ of *typed terms* (*terms* for short) as follows:

- If $\Gamma(x) = \sigma$ then $x^\sigma \in \mathcal{T}(\Sigma, \Gamma)$.
- If $\Sigma(f) \geq \sigma$ then $f^\sigma \in \mathcal{T}(\Sigma, \Gamma)$.

- If $t_1^{\sigma_1}, \ldots, t_n^{\sigma_n} \in \mathcal{T}(\Sigma, \Gamma)$ then $(t_1^{\sigma_1}, \ldots, t_n^{\sigma_n})^{\sigma_1 \times \cdots \times \sigma_n} \in \mathcal{T}(\Sigma, \Gamma)$.
- If $p_1^{\sigma}, \ldots, p_m^{\sigma} \in \mathcal{P}(\Sigma, \Gamma)$ and $r_1^{\sigma'}, \ldots, r_m^{\sigma'} \in \mathcal{T}(\Sigma, \Gamma)$ then $(\mathsf{fn}\ p_1^{\sigma} \Rightarrow r_1^{\sigma'} \mid \cdots \mid p_m^{\sigma} \Rightarrow r_m^{\sigma'})^{\sigma \to \sigma'} \in \mathcal{T}(\Sigma, \Gamma)$.[†]
- If $t^{\sigma_1 \to \sigma_2}, u^{\sigma_1} \in \mathcal{T}(\Sigma, \Gamma)$ then $(t\ u^{\sigma_1})^{\sigma_2} \in \mathcal{T}(\Sigma, \Gamma)$.

For $t \equiv (t_1^{\sigma_1}, \ldots, t_n^{\sigma_n})^{\sigma_1 \times \cdots \times \sigma_n}$, we identify t as $t_1^{\sigma_1}$ in case of $n = 1$, and $t \equiv ()^{\mathrm{unit}}$ in case of $n = 0$, where unit is a special type constructor. The *α-equivalence* of terms is denoted by $\equiv$. The set of *free variables* in a term t is denoted by $FV(t)$, and the set of *bound variables* in a term t is denoted by $BV(t)$. We also define the notions of *term/type substitution* and *term/type context* in the usual way. For simplicity, we assume that functional application is left-associative and the body of a functional abstraction extends as far right as possible. We may drop type information in a term whenever no confusion arises. We shortly denote $\mathsf{fn}\ p_1 \Rightarrow \mathsf{fn}\ p_2 \Rightarrow \cdots \Rightarrow \mathsf{fn}\ p_m \Rightarrow r$ by $\mathsf{fn}\ \overline{p_m} \Rightarrow r$ or $\mathsf{fn}\ \overline{p} \Rightarrow r$, and $u\ t_1 \cdots t_n$ by $u\ \overline{t_n}$ or $u\ \overline{t}$. For convenience, we also introduce the "variable convention", that is, we assume that bound variables in a term are all different, and are disjoint from free variables. Under this convention, $(\mathsf{fn}\ p \Rightarrow r)\theta \equiv \mathsf{fn}\ p \Rightarrow r\theta$ holds for any term substitution θ. The set $Pos(t)$ of *positions*, which are sequences of natural numbers, in a term t is defined as follows:

- $Pos(a\ \overline{t_n}) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{iq \mid q \in Pos(t_i)\}$ if $a \in \mathcal{D} \cup \mathcal{C} \cup \mathcal{V}$
- $Pos((t_1, \ldots, t_n)) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{iq \mid q \in Pos(t_i)\}$
- $Pos(\mathsf{fn}\ p_1 \Rightarrow r_1 \mid \cdots \mid p_m \Rightarrow r_m) = \{\varepsilon\} \cup \bigcup_{i=1}^{m} \{iq \mid q \in Pos(\mathsf{fn}\ p_i \Rightarrow r_i)\}$ if $m > 1$
- $Pos(\mathsf{fn}\ p \Rightarrow r) = \{\varepsilon\} \cup \{1q \mid q \in Pos(r)\}$
- $Pos(u\ \overline{t_n}) = \{\varepsilon\} \cup \{0q_0 \mid q_0 \in Pos(u) \setminus \{\varepsilon\}\} \cup \bigcup_{i=1}^{n} \{iq \mid q \in Pos(t_i)\}$ if $n > 0$ where $u \equiv (\mathsf{fn}\ p_1 \Rightarrow r_1 \mid \cdots \mid p_m \Rightarrow r_m)$

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 a *root position*, and a position q in t is said to be a *leaf position* if $q \in Pos(t)$ and $q1 \notin Pos(t)$. A context is said to be a *leaf context* if any hole occurs in a leaf position. The subterm of t at position p is denoted by $t|_p$, and the symbol at position p in t is denoted by $(t)_p$. For the sake of convenience, we interpret $(t)_q = \mathsf{tp}$ whenever $t|_q \equiv (t_1, \ldots, t_n)$, and interpret $(t)_q = \mathsf{fn}$ whenever $t|_q$ is a functional abstraction. Specially, the root symbol $(t)_\varepsilon$ is also denoted by $\mathsf{root}(t)$. The size $|t|$ of t is defined as the cardinality of $Pos(t)$. We denote the *(proper) subterm relation* by $\trianglerighteq_{sub}$ ($\triangleright_{sub}$). We define $t^{\sigma} \in \mathcal{T}_{nfun}$ iff σ is not a functional type, and $t \in \mathcal{T}^{cls}$ iff σ is closed for any $v^{\sigma} \trianglelefteq_{sub} t$.

As a matter of course, we fix a type environment Σ for defined symbols and constructors. A triple $(l^{\sigma}, r^{\sigma}, \Gamma)$ is said to be a *rewrite rule*, denoted by $l^{\sigma} \to r^{\sigma} : \Gamma$ ($l^{\sigma} \to r^{\sigma}$ for short) if l^{σ} and r^{σ} are terms of the same type σ under the type environment (Σ, Γ), $\mathsf{root}(l) \in \mathcal{D}$, and $FV(l) \supseteq FV(r)$.

[†]In this paper, we only study the termination but not confluence. Hence, we give no restriction for functional abstractions with pattern. In order to guarantee the confluence, we need suitable restrictions [12].

A *term rewriting model for functional programs* (TRFP) is a set of rewrite rules. 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 $u \equiv l\xi\ \overline{z_n}$, $v \equiv r\xi\ \overline{z_n}$, and $\xi(\sigma) = \overline{\sigma_n} \to \sigma'$, where each $z_i^{\sigma_i}$ is a fresh variable. We also denote by $Act(R)$ the set $\bigcup_{l \to r \in R} Act(l \to r)$. The relation $\xrightarrow[R]{}$ of a TRFP R is defined by $s \xrightarrow[R]{} t$ 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. We also define $s \xrightarrow[\mathsf{fn}]{} t$ iff there exists a context $C[\,]$ such that $s \equiv C[(\mathsf{fn}\ p_1 \Rightarrow r_1 \mid \cdots \mid p_m \Rightarrow r_m)\ p_i\theta]$ and $t \equiv C[r_i\theta]$. The *rewrite relation* $\xrightarrow[R,\mathsf{fn}]{}$ is the union of $\xrightarrow[R]{}$ and $\xrightarrow[\mathsf{fn}]{}$. Especially, we denote $s \xrightarrow[R,\mathsf{fn}]{}^{\varepsilon} t$ if a rewrite $\xrightarrow[R,\mathsf{fn}]{}$ occurs at the root position; otherwise we denote $s \xrightarrow[R,\mathsf{fn}]{}^{>\varepsilon} t$.

Example 2.1 Consider the following TRFP R_{len}:

$$\begin{cases} \mathrm{foldl}\ f\ e\ \mathrm{nil} \to e \\ \mathrm{foldl}\ f\ e\ (\mathrm{cons}\ (x, xs)) \to \mathrm{foldl}\ f\ (f\ (e, x))\ xs \\ \mathrm{len}\ xs \to \mathrm{foldl}\ (\mathsf{fn}\ (x, y) \Rightarrow \mathrm{suc}\ x)\ 0\ xs \end{cases}$$

The function $\mathrm{foldl} : (\alpha \times \beta \to \alpha) \to \alpha \to \mathrm{list}(\beta) \to \alpha$ is a typical higher-order function that is widely used in existing functional programs, where α and β are type variables, and list is a type constructor. The TRFP R_{len} gives a representation of a function that calculates the length of lists. We demonstrate the calculation of len (cons (t, nil)) as follows:

$$\begin{aligned} &\underline{\mathrm{len}\ (\mathrm{cons}\ (t, \mathrm{nil}))} \\ &\xrightarrow[R]{} \underline{\mathrm{foldl}\ (\mathsf{fn}\ (x, y) \Rightarrow \mathrm{suc}\ x)\ 0\ (\mathrm{cons}\ (t, \mathrm{nil}))} \\ &\xrightarrow[R]{} \mathrm{foldl}\ (\mathsf{fn}\ (x, y) \Rightarrow \mathrm{suc}\ x) \\ &\qquad\qquad \underline{((\mathsf{fn}\ (x, y) \Rightarrow \mathrm{suc}\ x)\ (0, t))}\ \mathrm{nil} \\ &\xrightarrow[\mathsf{fn}]{} \underline{\mathrm{foldl}\ (\mathsf{fn}\ (x, y) \Rightarrow \mathrm{suc}\ x)\ (\mathrm{suc}\ 0)\ \mathrm{nil}} \\ &\xrightarrow[R]{} \mathrm{suc}\ 0 \end{aligned}$$

A term t is said to be *terminating* if there exists no infinite rewrite sequence of $\xrightarrow[R,\mathsf{fn}]{}$ starting from t. We write $SN(t)$ if t is terminating. A TRFP R is said to be *terminating* if so is any t.

We naturally assume that there is a closed type other than the special type unit. Then, since actual rewrite rules are closed under type substitution, we obtain the following proposition.

Proposition 2.2 Let R be a TRFP. For any type substitution ξ, we have $s \xrightarrow[R,\mathsf{fn}]{} t \Rightarrow s\xi \xrightarrow[R,\mathsf{fn}]{} t\xi$. Hence R is terminating if any closed-typed term is terminating.

3. Term Decomposition

In order to define various notions and to prove various properties, we introduce adequate decompositions for terms. In this section, we present various types of decomposition for terms.

Since we permit functional abstraction with patterns, we can bundle several terms into one term. For example, we can bundle terms $(\mathsf{fn}\ 0 \Rightarrow 0)$ and $(\mathsf{fn}\ \mathrm{suc}\ x \Rightarrow x)$ in the term

$(\mathsf{fn}\ 0 \Rightarrow 0 \mid \mathrm{suc}\ x \Rightarrow x)$. To uncouple a bundle such as:

$$\mathsf{hdec}(\mathsf{fn}\ 0 \Rightarrow 0 \mid \mathrm{suc}\ x \Rightarrow x) = \{\mathsf{fn}\ 0 \Rightarrow 0, \mathsf{fn}\ \mathrm{suc}\ x \Rightarrow x\},$$

we introduce the notion of the head decomposition.

Definition 3.1 We define the function hdec as follows:

$$\begin{aligned}&\mathsf{hdec}((\mathsf{fn}\ p_1 \Rightarrow r_1 \mid \cdots \mid p_m \Rightarrow r_m)\ \overline{t}) \\ &\qquad = \textstyle\bigcup_{i=1}^{m} \mathsf{hdec}((\mathsf{fn}\ p_i \Rightarrow r_i)\ \overline{t}) \quad \text{if } m > 1 \\ &\mathsf{hdec}((\mathsf{fn}\ p \Rightarrow r)\ \overline{t}) = \{(\mathsf{fn}\ p \Rightarrow r')\ \overline{t} \mid r' \in \mathsf{hdec}(r)\} \\ &\mathsf{hdec}(t) = \{t\} \quad \text{otherwise}\end{aligned}$$

A term t is said to be a *single head binding term* if $t \in \mathsf{hdec}(t')$ for some t'.

We also introduce the notion of the head part in single head binding terms.

Definition 3.2 For single head binding terms, we define the function hd as follows:

- $\mathsf{hd}((\mathsf{fn}\ p \Rightarrow r)\ \overline{t}) = \mathsf{fn}\ p \Rightarrow \mathsf{hd}(r)$
- $\mathsf{hd}(t) = t \quad \text{if } \mathsf{root}(t) \neq \mathsf{fn}$

For the proof of the soundness of the dependency pair method on first-order settings, a term of the form $d(t_1, \ldots, t_n)$ such that $d \in \mathcal{D}$ and each argument t_i is terminating plays an important role [1]. To extend the dependency pair method onto higher-order settings, that is, to design the static dependency pair method, we introduce the notion of strong computability, which is a stronger property than termination and is closed under functional application: if t and u are strongly computable, then so is $t\ u$. For the soundness proof of the static dependency pair method without functional abstraction with a pattern, a term of the form $d\ \overline{t_n}$ $(\equiv d\ t_1 \cdots t_n)$ such that $d \in \mathcal{D}$ and each argument t_i is strongly computable also plays an important role [16]–[19], [22].

In this paper, we permit functional abstraction with patterns so that we need to decompose the terms of the form $\mathsf{fn}\ \overline{p} \Rightarrow d\ \overline{t_n}$. The most natural way may be to decompose the terms into $\mathsf{fn}\ \overline{z_n} \Rightarrow \mathsf{fn}\ \overline{p} \Rightarrow d\ \overline{z_n}$ and the arguments $t_1, \ldots, t_n$. Then we have $(\mathsf{fn}\ \overline{z_n} \Rightarrow \mathsf{fn}\ \overline{p} \Rightarrow d\ \overline{z_n})\ \overline{t_n} \xrightarrow[\mathsf{fn}]{*} \mathsf{fn}\ \overline{p} \Rightarrow d\ \overline{t_n}$, and $\mathsf{fn}\ \overline{p} \Rightarrow d\ \overline{t_n}$ is strongly computable whenever the decomposed terms are strongly computable. However, such decomposition has two problems that prevent a soundness proof for the static dependency pair method. One is that bound variables in t_i by $\overline{p}$ may become free variables. Another problem is that the size of $\mathsf{fn}\ \overline{z_n} \Rightarrow \mathsf{fn}\ \overline{p} \Rightarrow d\ \overline{z_n}$ may be larger than the size of $\mathsf{fn}\ \overline{p} \Rightarrow d\ \overline{t_n}$. To avoid such difficulties, we technically interpret $\mathsf{fn}\ \overline{p} \Rightarrow t_i$ $(i = 1, \ldots, n)$ as arguments of $\mathsf{fn}\ \overline{p} \Rightarrow d\ \overline{t_n}$. Indeed, $\mathsf{fn}\ \overline{p} \Rightarrow d\ \overline{t_n}$ is strongly computable whenever $\mathsf{fn}\ \overline{p} \Rightarrow d$ and each argument $\mathsf{fn}\ \overline{p} \Rightarrow t_i$ are strongly computable (cf. Lemma 5.12 with the empty substitution). We formalize such arguments as:

$$\mathsf{args}(\mathsf{fn}\ p \Rightarrow d\ \overline{t_n}) = \{\mathsf{fn}\ p \Rightarrow t_i \mid i = 1, \ldots, n\}$$

Definition 3.3 The function args is defined as follows:

- $\mathsf{args}(a\ \overline{t_n}) = \{\overline{t_n}\}$
- $\mathsf{args}((t_1, \ldots, t_n)) = \{\overline{t_n}\}$
- $\mathsf{args}((\mathsf{fn}\ p_1 \Rightarrow r_1 \mid \cdots \mid p_m \Rightarrow r_m)\ \overline{t_n})$
  $= \{\overline{t_n}\} \cup \bigcup_{i=1}^{m} \{\mathsf{fn}\ p_i \Rightarrow r_i' \mid r_i' \in \mathsf{args}(r_i)\}$

Finally, we introduce the notion of bind preserving subterms, which embody hdec, hd and args.

Definition 3.4 We inductively define the set $\mathsf{Sub}_{bp}(t)$ of *bind preserving subterms* as follows:

- $\mathsf{Sub}_{bp}(a\ \overline{t_n}) = \{a\ \overline{t_n}\} \cup \bigcup_{i=1}^{n} \mathsf{Sub}_{bp}(t_i)$
- $\mathsf{Sub}_{bp}(\,(t_1, \ldots, t_n)\,) = \{(t_1, \ldots, t_n)\} \cup \bigcup_{i=1}^{n} \mathsf{Sub}_{bp}(t_i)$
- $\mathsf{Sub}_{bp}(\,(\mathsf{fn}\ p_1 \Rightarrow r_1 \mid \cdots \mid p_m \Rightarrow r_m)\ \overline{t_n}\,)$
  $= \{\mathsf{fn}\ p_i \Rightarrow u_i \mid \exists i, u_i \in \mathsf{Sub}_{bp}(r_i)\} \cup \bigcup_{i=1}^{n} \mathsf{Sub}_{bp}(t_i)$

For instance, we consider the following term t such that $a_i \in \mathcal{D} \cup \mathcal{C} \cup \mathcal{V}$ for each i.

$$\begin{aligned}t \equiv\ &(\mathsf{fn}\ p_1 \Rightarrow a_1\ (a_2, a_3) \\ &\mid p_2 \Rightarrow (\mathsf{fn}\ p_3 \Rightarrow a_4\ a_5 \mid p_4 \Rightarrow a_6)\ a_7) \\ &\quad (\mathsf{fn}\ p_5 \Rightarrow a_8 \mid p_6 \Rightarrow a_9)\end{aligned}$$

Then $\mathsf{hdec}(t)$ consists of the following three terms:

$$\begin{aligned}t_1 \quad &\equiv \quad (\mathsf{fn}\ p_1 \Rightarrow a_1\ (a_2, a_3))\ (\mathsf{fn}\ p_5 \Rightarrow a_8 \mid p_6 \Rightarrow a_9) \\ t_2 \quad &\equiv \quad (\mathsf{fn}\ p_2 \Rightarrow (\mathsf{fn}\ p_3 \Rightarrow a_4\ a_5)\ a_7) \\ &\qquad\qquad (\mathsf{fn}\ p_5 \Rightarrow a_8 \mid p_6 \Rightarrow a_9) \\ t_3 \quad &\equiv \quad (\mathsf{fn}\ p_2 \Rightarrow (\mathsf{fn}\ p_4 \Rightarrow a_6)\ a_7) \\ &\qquad\qquad (\mathsf{fn}\ p_5 \Rightarrow a_8 \mid p_6 \Rightarrow a_9)\end{aligned}$$

Each $\mathsf{hd}(t_i)$ is as follows:

$$\begin{aligned}\mathsf{hd}(t_1) \quad &\equiv \quad \mathsf{fn}\ p_1 \Rightarrow a_1\ (a_2, a_3) \\ \mathsf{hd}(t_2) \quad &\equiv \quad \mathsf{fn}\ p_2 \Rightarrow \mathsf{fn}\ p_3 \Rightarrow a_4\ a_5 \\ \mathsf{hd}(t_3) \quad &\equiv \quad \mathsf{fn}\ p_2 \Rightarrow \mathsf{fn}\ p_4 \Rightarrow a_6\end{aligned}$$

Then $\mathsf{args}(t)$ consists of the following four terms:

$$\begin{aligned}t_4 \quad &\equiv \quad \mathsf{fn}\ p_1 \Rightarrow (a_2, a_3) \\ t_5 \quad &\equiv \quad \mathsf{fn}\ p_2 \Rightarrow \mathsf{fn}\ p_3 \Rightarrow a_5 \\ t_6 \quad &\equiv \quad \mathsf{fn}\ p_2 \Rightarrow a_7 \\ t_7 \quad &\equiv \quad \mathsf{fn}\ p_5 \Rightarrow a_8 \mid p_6 \Rightarrow a_9\end{aligned}$$

The set of bind preserving subterms $\mathsf{Sub}_{bp}(t)$ consists of ten terms: $\mathsf{hd}(t_1), \mathsf{hd}(t_2), \mathsf{hd}(t_3)$, t_4, t_5, t_6, and two elements of $\mathsf{args}(t_4)$:

$$\mathsf{fn}\ p_1 \Rightarrow a_2 \qquad \mathsf{fn}\ p_1 \Rightarrow a_3$$

and two elements of $\mathsf{hdec}(t_7)$:

$$\mathsf{fn}\ p_5 \Rightarrow a_8 \qquad \mathsf{fn}\ p_6 \Rightarrow a_9$$

4. Strong Computability and Accessibility

The theoretical basis of the SDP-method is given by the notion of strong computability, and the soundness of the SDP-method is guaranteed by the notion of accessibility. In

this section, we introduce these key notions [19]. By using these notions, we formulate the class, namely accessible TRFPs (ATRFPs), in which the soundness of the SDP-method holds. 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. 4.2). It will refer only to the abstract framework (Sect. 4.1).

4.1 Abstract Framework

Definition 4.1 A predicate P over $\mathcal{T}^{cls}$ is said to be a *strong computability predicate* if the following properties hold:

(SC1) For any $t \in \mathcal{T}^{cls}$, if $P(t)$ then $SN(t)$.
(SC2) For any $t^{\sigma_1 \to \sigma_2}$, $u^{\sigma_1} \in \mathcal{T}^{cls}$, if $P(t)$ and $P(u)$ then $P(t\,u)$.
(SC3) For any $t^{\sigma_1 \to \sigma_2} \in \mathcal{T}^{cls}$, if $\forall u^{\sigma_1} \in \mathcal{T}^{cls}[P(u) \Rightarrow P(t\,u)]$ then $P(t)$.
(SC4) For any $t, u \in \mathcal{T}^{cls}$, if $P(t)$ and $t \underset{R,\mathsf{fn}}{\longrightarrow} u$ then $P(u)$.
(SC5) For any $t \in \mathcal{T}^{cls}_{nfun}$, if $\forall u \in \mathcal{T}^{cls} \cap (\{t' \mid t \underset{R,\mathsf{fn}}{\longrightarrow} t'\} \cup T).P(u)$ then $P(t)$, where $T = \mathsf{args}(t)$ if $\mathsf{root}(t) \neq \mathsf{fn}$; otherwise $T = \emptyset$.

Throughout the paper, we use notations $\mathcal{T}_{SC} = \{t \mid SC(t)\}$, $\mathcal{T}_{\neg SC} = \{t \mid \neg SC(t)\}$, and $\mathcal{T}^{\mathsf{args}}_{SC} = \{t \mid \forall u \in \mathsf{args}(t), SC(u)\}$ for each strong computability predicate SC. We also use notations $\mathcal{T}_{SN} = \{t \mid SN(t)\}$, $\mathcal{T}_{\neg SN} = \{t \mid \neg SN(t)\}$, and $\mathcal{T}^{\mathsf{args}}_{SN} = \{t \mid \forall u \in \mathsf{args}(t), SN(u)\}$.

Definition 4.2 For a strong computability predicate SC, a function A from $\mathcal{T}$ to sets of $\mathcal{T}$ is said to be an *accessible function* if the following properties hold:

(Acc1) For any $t, u \in \mathcal{T}^{cls}$, if $u \in A(t)$ and $t \in \mathcal{T}^{\mathsf{args}}_{SC}$ then $SC(u)$.
(Acc2) For any $t, u \in \mathcal{T}^{cls}$ and term substitution θ, if $u \in A(t)$ then $u\theta \in A(t\theta)$.
(Acc3) For any $t, u \in \mathcal{T}$ and type substitution ξ, if $u \in A(t)$ then $u\xi \in A(t\xi)$.

Definition 4.3 A TRFP R is said to be *accessible*, if there exist a strong computability predicate SC and an accessible function Acc such that

- C is *accessible*, that is, $t_i \in Acc(c\ \overline{t_n})$ for any i and $c \in C$, and
- for any rule $l \to r \in R$ and $a\ \overline{r_n} \lhd_{sub} r$ with $a \in FV(r)$, there exists k ($\leq n$) such that $a\ \overline{r_k} \in Acc(l)$.

An accessible TRFP is often shortly denoted by ATRFP.

In the introduction, we explained that the SDP-method is not sound in general. The accessibility gives a sufficient condition of the soundness of the SDP-method. In the following, we show that the non-terminating TRFP $R = \{\text{foo (bar } f) \to f \text{ (bar } f)\}$ displayed in the introduction is not accessible.

Assume that the TRFP is accessible with a strong computability predicate SC and an accessible function Acc. From the assumption, we have $f \in Acc(\text{foo (bar } f))$. Thanks to (Acc3), we suppose that these terms are in $\mathcal{T}^{cls}$.

In case of $SC(\text{bar foo})$, since $\text{foo} \in Acc(\text{foo (bar foo)})$ by (Acc2), we have $SC(\text{foo})$ by (Acc1). From (SC2), we have $SC(\text{foo (bar foo)})$. It is a contradiction with (SC1).

In case of $\neg SC(\text{bar foo})$, we have $\neg SC(\text{foo})$ by (SC5). From (SC3), there is $u \in \mathcal{T}^{cls}$ such that $SC(u)$ and $\neg SC(\text{foo } u)$. From (SC5), (SC1) and (SC4), there is a reduction sequence $\text{foo } u \underset{R,\mathsf{fn}}{\overset{*}{\longrightarrow}} \text{foo (bar } u') \underset{R}{\to} u' \text{ (bar } u')$ such that $SC(\text{bar } u')$ and $\neg SC(u' \text{ (bar } u'))$. Since $u' \in Acc(\text{foo (bar } u'))$ by (Acc2), $SC(u')$ follows from (Acc1). However, $\neg SC(u')$ follows from (SC2). It is a contradiction.

4.2 Construction

In order to construct a strongly computable predicate and an accessible function, we introduced the notion of peeled subterms [17], which was extended to TRFPs without functional abstractions [19]. The result can be used in this paper because we do not change the type systems. In this section, we slightly improve the result by paying attention to features of product types. The benefit by this improvement will be demonstrated using an example at the end of this section.

Definition 4.4 We define the function pcomp as follows:

$$\mathsf{pcomp}(\sigma_1 \times \cdots \times \sigma_n) = \bigcup_{i=1}^{n} \mathsf{pcomp}(\sigma_i) \quad \text{if } n \geq 2$$
$$\mathsf{pcomp}(\sigma) = \{\sigma\} \quad \text{otherwise}$$

Definition 4.5 A set PT of *peeling types* is a subset of all data types. We define $PT_{\geq}$ as follows:

$$\{\sigma \mid \exists \sigma' \in PT, \sigma' \geq \sigma\} \cup \{\sigma \mid \sigma \text{ is a product type}\}$$

A well-founded quasi order $\gtrsim$ on types is said to be a *peeling order* if the following properties hold:

- If $\sigma' \gtrsim \sigma$ then $\xi(\sigma') \gtrsim \xi(\sigma)$ for any closed-typed substitution ξ
- $\sigma_1 \times \cdots \times \sigma_n \gtrsim \sigma_i$ for any closed types $\sigma_1, \ldots, \sigma_n$
- $\sigma_1 \to \sigma_2 \gtrsim \sigma_i$ $(i = 1, 2)$ for any closed types σ_1 and σ_2, where $\gtrsim$ is the strict part of $\gtrsim$

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

- $\mathsf{args}(t) \subseteq Sub^{\gtrsim}_{PT}(t)$,
- if $(t_1, \ldots, t_n) \in Sub^{\gtrsim}_{PT}(t)$ then $\forall i, t_i \in Sub^{\gtrsim}_{PT}(t)$, and
- if $u \equiv (a\ \overline{u_n^{\sigma_n}})^{\sigma} \in Sub^{\gtrsim}_{PT}(t)$, $\sigma \in PT_{\geq}$, and $\forall \sigma' \in \mathsf{pcomp}(\sigma_i), \sigma \gtrsim \sigma'$ then $u_i^{\sigma_i} \in Sub^{\gtrsim}_{PT}(t)$.

Definition 4.6 For a set PT of peeling types and peeling order $\gtrsim$, we define $SC(t^{\sigma})$ as follows:

- In case of $t^{\sigma} \in \mathcal{T}^{cls}_{nfun}$ and $\sigma \notin PT_{\geq}$, $SC(t)$ is defined as $SN(t)$.
- In case of $t^{\sigma} \in \mathcal{T}^{cls}_{nfun}$ and $\sigma \in PT_{\geq}$, $SC(t)$ is defined as $SN(t)$ and $SC(u)$ for any $u^{\sigma'} \in \mathcal{T}^{cls} \cap (\{t' \mid t \underset{R}{\to} t'\} \cup T)$

such that $\sigma \gtrsim \sigma'$, where $T = \mathsf{args}(t)$ if $\mathsf{root}(t) \neq \mathsf{fn}$; otherwise $T = \emptyset$.
- 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 $SC(u)$.

The well-definedness of SC can be shown as similar to [19]. We note that the value of $SC(t)$ for non-terminating term t is set to false, and the value of $SC(t)$ for terminating term t^σ is inductively defined on (σ, t) with respect to the lexicographic combination of $(\gtrsim, \underset{R,\mathsf{fn}}{\longrightarrow} \cup \rhd_{sub})$.

Definition 4.7 For a set PT of peeling types and a peeling order $\gtrsim$, we define the function Acc as follows:

$$Acc(t) = Sub^{\gtrsim}_{PT}(t) \cup \{u \mid t \rhd_{sub} u^\sigma \in \mathcal{T}^{cls}_{nfun}, \sigma \notin PT_{\gtrsim}\}$$

Theorem 4.8 The predicate SC given in Definition 4.6 is a strong computability predicate, and the function Acc given in Definition 4.7 is an accessible function.

Proof. We can prove the claim as similar to Theorem 3.6 and 3.7 in [19]. □

Under SC and Acc introduced in this section, the TRFP for Ackermann function discussed in Sect. 1 becomes ATRFP. The TRFP R_{len} in Example 2.1 is also ATRFP.

Example 4.9 Consider the TRFP R_{len} in Example 2.1. 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 $\mathrm{tp}_n(\sigma_1, \ldots, \sigma_n)$), we construct the peeling order $\gtrsim$ by using the recursive path order $\gtrsim_{rpo}$ with the empty precedence [6]. Then the order $\gtrsim_{rpo}$ becomes a peeling order. We take PT as the set of all data types. Then the TRFP R_{len} becomes accessible. In fact, the first and third rules trivially satisfy the desired property because all variables occur in argument positions. Suppose that $t \equiv \mathrm{foldl}\ f\ e\ (\mathrm{cons}\ (x, xs))$. Then:

- we have $f, e, \mathrm{cons}\ (x, xs) \in Sub^{\gtrsim}_{PT}(t)$ because of $\mathsf{args}(t) \subseteq Sub^{\gtrsim}_{PT}(t)$,
- we have $(x, xs) \in Sub^{\gtrsim}_{PT}(t)$ because of $(\mathrm{cons}\ (x, xs)^{\alpha \times \mathrm{list}(\alpha)})^{\mathrm{list}(\alpha)} \in Sub^{\gtrsim}_{PT}(t)$, $\mathsf{pcomp}(\alpha \times \mathrm{list}(\alpha)) = \{\alpha, \mathrm{list}(\alpha)\}$, $\mathrm{list}(\alpha) \gtrsim_{rpo} \alpha$ and $\mathrm{list}(\alpha) \gtrsim_{rpo} \mathrm{list}(\alpha)$, and
- we have $x, xs \in Sub^{\gtrsim}_{PT}(t)$ because of $(x, xs) \in Sub^{\gtrsim}_{PT}(t)$.

Since PT is the set of all data types, we have $\{u \mid t \rhd_{sub} u^\sigma \in \mathcal{T}^{cls}_{nfun}, \sigma \notin PT_{\gtrsim}\} = \emptyset$. Hence we have

$$Acc(t) = Sub^{\gtrsim}_{PT}(t) = \{f, e, \mathrm{cons}\ (x, xs), (x, xs), x, xs\}.$$

Since $FV(\mathrm{foldl}\ f\ (f\ (e, x))\ xs) = \{f, e, x, xs\} \subseteq Acc(t)$, the second rule

$$\mathrm{foldl}\ f\ e\ (\mathrm{cons}\ (x, xs)) \to \mathrm{foldl}\ f\ (f\ (e, x))\ xs$$

also satisfies the desired property. Therefore R_{len} is accessible.

In this section, we slightly improve the result [19] by paying attention to feature of product types, that is, a subterm $t_i^{\sigma_i}$ of a product typed term $(\ldots, t_i^{\sigma_i}, \ldots)^{\cdots \times \sigma_i \times \cdots}$ is of syntactical subtypes σ_i of the product type $\cdots \times \sigma_i \times \cdots$. In the framework in [19], we had to design a peeling order that satisfies $\mathrm{list}(\alpha) \gtrsim \alpha$, $\mathrm{list}(\alpha) \gtrsim \mathrm{list}(\alpha)$, and $\mathrm{list}(\alpha) \gtrsim \alpha \times \mathrm{list}(\alpha)$. Indeed, the above example does not require:

$$\mathrm{list}(\alpha) \gtrsim \alpha \times \mathrm{list}(\alpha)$$

Although designing such orders is possible, it is very cumbersome. This is a benefit by our improvement.

5. Static Dependency Pair Method

The SDP-method was introduced on simply-typed term rewriting systems (STRSs) by us [16], [17]. Then we extended the method on higher-order rewrite systems (HRSs) by Nipkow [20] in which functional abstraction restricted to β-normal η-long forms is permitted [18], [22]. Moreover, to bring the method close to the existing functional programs, we extend the method onto term rewriting models for functional programs with product, algebraic data, and ML-polymorphic types [19]. In this section, we extend the SDP-method on TRFPs in which functional abstraction with pattern is permitted. We note that our extension permits the representation for higher-order primitive recursion in Sect. 1 although it could not be represented in [19].

Firstly, we introduce the notion of static dependency pairs, which is the most basic notion in the static dependency pair method. In the following, we assume that there exist strongly computable predicate SC and accessible function Acc.

Definition 5.1 For each $f \in \mathcal{D}$, we provide a new function symbol $f^\sharp$, called the *marked-symbol* of f. For each $t \equiv a\ \overline{t_n}$ with $a \in \mathcal{D}$, we define the *marked term* $t^\sharp$ by $a^\sharp\ \overline{t_n}$. We assume that the marking does not change the type information, that is, t and $t^\sharp$ have the same type. For t^σ, we may write $t^\sharp : \sigma$ instead of $t^{\sharp\sigma}$ in order to avoid any confusion.

A pair $\langle l^\sharp,\ a^\sharp\ \overline{r_n} \rangle$ 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:

- $a \in \mathcal{D}$
- $a\ \overline{r_k} \notin Acc(l)$ for all $k\ (\leq n)$

A pair $\langle l^\sharp,\ a^\sharp\ \overline{r_n} \rangle$ is said to be an *inner static dependency pair* in R if it is not an outer static dependency pair and there exists a rule $l \to r \in R$ satisfying the following conditions:

- $\mathsf{fn}\ \overline{p} \Rightarrow a\ \overline{r_n} \in \mathsf{Sub}_{bp}(r)$
- $a \in \mathcal{D}$
- $a\ \overline{r_k} \notin Acc(l)$ for all $k\ (\leq n)$

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. We may denote a static dependency pair $\langle l^\sharp, r^\sharp \rangle$ by $l^\sharp \to r^\sharp$.

Example 5.2 Consider the TRFP R_{ack} displayed in Sect. 1.

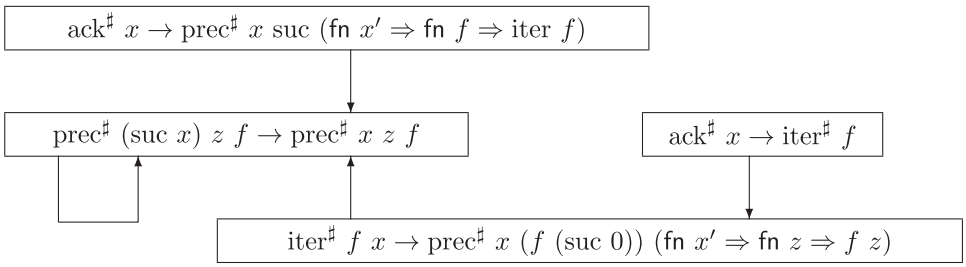

Fig. 1 static dependency graph in R_{ack}

$$\begin{cases} \text{prec } 0\ z\ f \to z \\ \text{prec } (\text{suc } x)\ z\ f \to f\ x\ (\text{prec } x\ z\ f) \\ \text{iter } f\ x \to \text{prec } x\ (f\ (\text{suc } 0))\ (\text{fn } x' \Rightarrow \text{fn } z \Rightarrow f\ z) \\ \text{ack } x \to \text{prec } x\ \text{suc } (\text{fn } x' \Rightarrow \text{fn } f \Rightarrow \text{iter } f) \end{cases}$$

Then the set $SDP(R_{\text{ack}})$ of static dependency pairs consists of the following four pairs:

$$\begin{cases} \text{prec}^\sharp\ (\text{suc } x)\ z\ f \to \text{prec}^\sharp\ x\ z\ f \\ \text{iter}^\sharp\ f\ x \to \text{prec}^\sharp\ x\ (f\ (\text{suc } 0))\ (\text{fn } x' \Rightarrow \text{fn } z \Rightarrow f\ z) \\ \text{ack}^\sharp\ x \to \text{prec}^\sharp\ x\ \text{suc } (\text{fn } x' \Rightarrow \text{fn } f \Rightarrow \text{iter } f) \\ \text{ack}^\sharp\ x \to \text{iter}^\sharp\ f \end{cases}$$

We note that the second and third pairs are outer static dependency pairs, and the first and fourth pairs are inner static dependency pairs.

Definition 5.3 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: $s^\sharp \to t^\sharp \in Act(u^\sharp : \sigma \to v^\sharp : \sigma)$ iff $s^\sharp, t^\sharp \in \mathcal{T}^{cls}$, and there is a type substitution ξ such that $s^\sharp \equiv u^\sharp\xi\ \overline{z_n}$, $t^\sharp \equiv v^\sharp\xi\ \overline{z_n}$, where the canonical form of $\xi(\sigma)$ is $\overline{\tau_n} \to \tau$ and 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 $s^\sharp, t^\sharp \in \mathcal{T}^{cls}$, and there is a type substitution ξ such that $s^\sharp \equiv u^\sharp\xi\ \overline{z'_n}$, $t^\sharp \equiv v^\sharp\xi\ \overline{z_m}$, where the canonical forms of $\xi(\sigma')$ and $\xi(\sigma)$ are $\overline{\tau'_n} \to \tau'$ and $\overline{\tau_m} \to \tau$, respectively, and 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 5.4 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 $s_1^\sharp \to t_1^\sharp \in Act(u_1^\sharp \to v_1^\sharp)$, $s_2^\sharp \to t_2^\sharp \in Act(u_2^\sharp \to v_2^\sharp), \ldots$, and term substitutions $\theta_1, \theta_2, \ldots$ such that for any i, $t_i^\sharp\theta_i \xrightarrow[R,\text{fn}]{*} s_{i+1}^\sharp\theta_{i+1}$, and $s_i\theta_i, t_i\theta_i \in \mathcal{T}_{SC}^{\text{args}} \cap \mathcal{T}_{\neg SC}$

We give the fundamental theorem of the SDP-method. Its proof is mentioned later.

Theorem 5.5 Let R be an ATRFP. If there exists no infinite static dependency chain then R is terminating.

Each static dependency pair expresses nothing but the local dependency of functions based on dependency relationships displayed in rules. To analyze the global dependency of functions, in other words, to analyze the static recursive structure, we introduce notions of a static dependency graph and a static recursion component.

Definition 5.6 The *static dependency graph in 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.

A *static recursion component* in R is a set of nodes either in a finite strongly connected subgraph, or in an infinite path that include infinitely many kind of static dependency pairs.

Using $SRC(R)$ we denote the set of static recursion components in R.

Example 5.7 The static dependency graph of the TRFP R_{ack} is shown in Fig. 1. The static dependency graph in R_{ack} has only the static recursion component:

$$\{\text{prec}^\sharp\ (\text{suc } x)\ z\ f \to \text{prec}^\sharp\ x\ z\ f\}$$

Similar to other dependency pair methods, the static dependency pair method proves the termination by proving the non-loopingness of each static recursion component.

Definition 5.8 A static recursion component C in a TRFP R is said to be *non-looping* if there exists no infinite static dependency chain such that only pairs in C occur, and either C is infinite or every $u^\sharp \to v^\sharp \in C$ occurs infinitely many times.

As a corollary of Theorem 5.5, we obtain the following:

Corollary 5.9 Let R be an ATRFP. If any static recursion components in R is non-looping then R is terminating.

We will discuss in the next section how to show the non-loopingness.

At the front in Sect. 1, we stated that the principle of the SDP-method is that if any recursion is suitably defined, then it is terminating. This corollary is a formulation of this principle.

We also obtain the following corollary of Theorem 5.5 by considering the case of $R = \emptyset$.

Corollary 5.10 If $\mathcal{C}$ is accessible then λ-calculi with pattern is terminating.

Here λ-calculi with pattern denotes the rewrite relation $\xrightarrow[\mathsf{fn}]{}$. We note that it may not be terminating in case that $\mathcal{C}$ is not accessible. In fact, the rule foo (bar f) $\to$ f (bar f) discussed in Sect. 1 is not terminating. By using functional abstraction with pattern, this function "foo" can be represented as the term "fn bar $f \Rightarrow f$ (bar f)". They indicate that the empty TRFP $\emptyset$ may be non-terminating, and so is λ-calculi with pattern. Note that the SDP-method cannot apply to this example, since the constructor "bar" is not accessible.

Soundness Proof

In the remainder of this section, we show the soundness of the SDP-method on ATRFPs, that is, we give a proof of Theorem 5.5. With the introduction of functional abstraction with pattern, it is necessary to wholly rebuild the soundness proof in [19]. For strong computability and accessibility, it will refer only to the abstract framework (Definition 4.1 and 4.2). Through the section, we assume that any TRFP is accessible and any term is of closed types.

Firstly we present basic properties of strong computability. Here, we define $\mathsf{hargs}(t) = \{\overline{r_m}\}$ and $\mathsf{targs}(t) = \{\overline{u_n}\}$ if $t \equiv (\mathsf{fn}\ p_1 \Rightarrow r_1 \mid \cdots \mid p_m \Rightarrow r_m)\ \overline{u_n}$; otherwise $\mathsf{hargs}(t) = \emptyset$ and $\mathsf{targs}(t) = \mathsf{args}(t)$.

Lemma 5.11

(1) If $t \in \mathcal{T}_{\neg SC}$, $\mathsf{targs}(t) \subseteq \mathcal{T}_{SC}$ and $\mathsf{hargs}(t) \subseteq \mathcal{T}_{SN}$ then there exist $\overline{u} \in \mathcal{T}_{SC}$ and v such that $t\ \overline{u} \xrightarrow[R,\mathsf{fn}]{}^{*} \xrightarrow[R,\mathsf{fn}]{}^{\varepsilon} v \in \mathcal{T}_{nfun} \cap \mathcal{T}_{\neg SC}$.
(2) If $\overline{t} \in \mathcal{T}_{SC}$ then $(\overline{t}) \in \mathcal{T}_{SC}$ and $a\ \overline{t} \in \mathcal{T}_{SC}$ for any $a \in \mathcal{C} \cup \mathcal{V}$.
(3) For any pattern p, if $p\theta \in \mathcal{T}_{SC}$ then $x\theta \in \mathcal{T}_{SC}$ for all $x \in FV(p)$.
(4) If $\mathsf{fn}\ p \Rightarrow r \in \mathcal{T}_{SC}$ then $r \in \mathcal{T}_{SC}$.
(5) If $r \in \mathcal{T}_{SC}$ and $FV(p) \cap FV(r) = \emptyset$ then $\mathsf{fn}\ p \Rightarrow r \in \mathcal{T}_{SC}$
(6) $\mathsf{fn}\ \overline{p_m} \Rightarrow x \in \mathcal{T}_{SC}$ for any $x \in \mathcal{V}$.

Proof.

(1) From (SC3), there exists $\overline{u} \in \mathcal{T}_{SC}$ such that $t\ \overline{u} \in \mathcal{T}_{nfun} \cap \mathcal{T}_{\neg SC}$. Then the existence of a desired sequence follows from (SC4), (SC1) and (SC5).
(2) Assume that $a\ \overline{t} \in \mathcal{T}_{\neg SC}$ for some $a \in \mathcal{C} \cup \mathcal{V}$. From $\overline{t} \in \mathcal{T}_{SC}$ and (1), there exist a sequence $a\ \overline{t}\ \overline{u} \xrightarrow[R,\mathsf{fn}]{}^{*} \xrightarrow[R,\mathsf{fn}]{}^{\varepsilon} a\ \overline{t'}\ \overline{u'} \xrightarrow[R,\mathsf{fn}]{}^{\varepsilon} v$. Since $\forall l \to r \in R, \mathsf{root}(l) \in \mathcal{D}$, we have $a \in \mathcal{D}$. It is a contradiction. As similar, we can also prove $(\overline{t}) \in \mathcal{T}_{SC}$.
(3) It is easily proved by induction on pattern p with the accessibility of $\mathcal{C}$.
(6') The assumption $x \notin \mathcal{T}_{SC}$ derives a contradiction with (1). Hence the claim (6) holds for $m = 0$.
(4) Since $(\mathsf{fn}\ p \Rightarrow r)\ p \xrightarrow[\mathsf{fn}]{} r$, thanks to (SC2) and (SC4), it suffices to show that any pattern p is strongly computable, which is easily proved by induction on p by using (6') and (2).
(5) Assume that $\mathsf{fn}\ p \Rightarrow r \notin \mathcal{T}_{SC}$. From (SC1), (1) and $FV(p) \cap FV(r) = \emptyset$, there exist $u, \overline{v} \in \mathcal{T}_{SC}$ such that $(\mathsf{fn}\ p \Rightarrow r)\ u\ \overline{v} \xrightarrow[R,\mathsf{fn}]{}^{*} (\mathsf{fn}\ p \Rightarrow r')\ p\theta\ \overline{v'} \xrightarrow[\mathsf{fn}]{} r'\ \overline{v'} \notin \mathcal{T}_{SC}$. Then we have $r\ \overline{v} \xrightarrow[R,\mathsf{fn}]{}^{*} r'\ \overline{v'}$. From (SC4), we have $r\ \overline{v} \notin \mathcal{T}_{SC}$. On the other hand, $r\ \overline{v} \in \mathcal{T}_{SC}$ follows from (SC2). This is a contradiction.
(6) We proceed by induction on m. The case $m = 0$ has already shown in (6'). Suppose that $(\mathsf{fn}\ \overline{p_m} \Rightarrow x) \equiv (\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow x)$.
Assume that $\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow x \in \mathcal{T}_{\neg SC}$. From $\mathsf{fn}\ \overline{q} \Rightarrow x \in \mathcal{T}_{SN}$ and (1), there exist $\overline{u} \in \mathcal{T}_{SC}$ such that $(\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow x)\ \overline{u} \xrightarrow[R,\mathsf{fn}]{}^{*} (\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow x)\ p\theta\ \overline{w} \xrightarrow[\mathsf{fn}]{} (\mathsf{fn}\ \overline{q} \Rightarrow x\theta)\ \overline{w} \notin \mathcal{T}_{SC}$. In case of $x\theta \equiv x$, we have $\mathsf{fn}\ \overline{q} \Rightarrow x\theta \equiv \mathsf{fn}\ \overline{q} \Rightarrow x \in \mathcal{T}_{SC}$ from the induction hypothesis. In case of $x\theta \not\equiv x$, we have $x\theta \in \mathcal{T}_{SC}$ by (SC4) and (3), and hence $\mathsf{fn}\ \overline{q} \Rightarrow x\theta \in \mathcal{T}_{SC}$ from the variable convention and (5). In both cases, $(\mathsf{fn}\ \overline{q} \Rightarrow x\theta)\ \overline{w} \in \mathcal{T}_{SC}$ follows from (SC4) and (SC2). This is a contradiction. □

For the soundness of the SDP-method, we require the accessibility of $\mathcal{C}$. The property of Lemma 5.11 (3) is the essence of such a requirement.

Next, we show the property that $\mathsf{fn}\ \overline{p} \Rightarrow t\ \overline{u_n}$ is strongly computable whenever $\mathsf{fn}\ \overline{p} \Rightarrow t$ and each $\mathsf{fn}\ \overline{p} \Rightarrow u_i$ are strongly computable (cf. Lemma 5.12 with the empty substitution). We define a *single head binding context* as a context generated by the grammar: $H ::= \Box \mid (\mathsf{fn}\ p \Rightarrow H)\ \overline{t}$.

Lemma 5.12 For any strongly computable substitution θ, (i.e. $\forall x \in \mathcal{V}.\ x\theta \in \mathcal{T}_{SC}$) such that $(\mathsf{fn}\ \overline{p_m} \Rightarrow t^{\overline{\alpha_n} \to \beta})\theta \in \mathcal{T}_{SC}$, and $(\mathsf{fn}\ \overline{p_m} \Rightarrow u_i^{\alpha_i})\theta \in \mathcal{T}_{SC}$ $(i = 1, \ldots, n)$, we have $(\mathsf{fn}\ \overline{p_m} \Rightarrow t\ \overline{u_n})\theta \in \mathcal{T}_{SC}$.

Proof. We proceed by induction on m. The case $m = 0$ follows from (SC2). In case of $m > 0$, we suppose that $(\mathsf{fn}\ \overline{p_m} \Rightarrow t\ \overline{u_n})\theta \equiv (\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow t\ \overline{u_n})\theta \equiv (\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow t\theta\ \overline{u_n\theta})$. Assume that $(\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow t\theta\ \overline{u_n\theta})$ is not strongly computable.

From Lemma 5.11 (4), $\mathsf{fn}\ \overline{q} \Rightarrow t\theta$ and each $\mathsf{fn}\ \overline{q} \Rightarrow u_i\theta$ are strongly computable. From the induction hypothesis, $\mathsf{fn}\ \overline{q} \Rightarrow t\theta\ \overline{u_n\theta}$ is strongly computable. From (SC1) and Lemma 5.11 (1), there exist $v, \overline{w} \in \mathcal{T}_{SC}$ such that $(\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow t\theta\ \overline{u_n\theta})\ v\ \overline{w} \xrightarrow[R,\mathsf{fn}]{}^{*} (\mathsf{fn}\ p \Rightarrow t')\ p\theta_p\ \overline{w'} \xrightarrow[\mathsf{fn}]{} t'\theta_p\ \overline{w'} \in \mathcal{T}_{\neg SC}$. From (SC4) and Lemma 5.11 (3), θ_p is strongly computable. Thanks to the variable convention, we can define the substitution $\theta' = \theta \cup \theta_p$, which is strongly computable. Then we have $(\mathsf{fn}\ \overline{q} \Rightarrow t\theta\ \overline{u_n\theta})\theta_p\ \overline{w} \xrightarrow[R,\mathsf{fn}]{}^{*} t'\theta_p\ \overline{w'}$. From (SC4), we have $(\mathsf{fn}\ \overline{q} \Rightarrow t\theta\ \overline{u_n\theta})\theta_p\ \overline{w} \equiv (\mathsf{fn}\ \overline{q} \Rightarrow t\theta'\ \overline{u_n\theta'})\ \overline{w} \in \mathcal{T}_{\neg SC}$. From (SC2), we have $\mathsf{fn}\ \overline{q} \Rightarrow t\theta'\ \overline{u_n\theta'} \in \mathcal{T}_{\neg SC}$.

Since $(\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow t\theta)$ and v are strongly computable, strong computability of $(\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow t\theta)\ v$ follows from (SC2), and hence strong computability of $\mathsf{fn}\ \overline{q} \Rightarrow t\theta'$ follows from (SC4). As similarity, each $\mathsf{fn}\ \overline{q} \Rightarrow u_i\theta'$ is strongly computable. Hence we have $\mathsf{fn}\ \overline{q} \Rightarrow t\theta'\ \overline{u_n\theta'} \in \mathcal{T}_{SC}$ from the induction hypothesis. It is a contradiction. □

We have shown the basic properties for strong com-

putability. We now prove the soundness of the static dependency pair method. First we will characterize minimal counterexamples for termination (cf. Lemma 5.18).

Lemma 5.13 If $t'\theta \in \mathcal{T}_{SC}$ for any $t' \in \mathsf{hdec}(t)$ then $t\theta \in \mathcal{T}_{SC}$.

Proof. We proceed by induction on $|t|$. Since the case $t \in \mathsf{hdec}(t)$ is trivial, it suffices to show the case that t has the form of $H'[(\mathsf{fn}\ p_1 \Rightarrow r_1 \mid \cdots \mid p_m \Rightarrow r_m)\ \overline{u}]$ with $m > 1$, where $H'[\,]$ is a single head binding context. Suppose that $H[\,] \equiv H'[\,]\theta$.

Assume that $t\theta \notin \mathcal{T}_{SC}$. From (SC3), there exist $\overline{v} \in \mathcal{T}_{SC}$ such that $t\theta\ \overline{v} \in \mathcal{T}_{nfun} \cap \mathcal{T}_{\neg SC}$. Since $\mathsf{hdec}(t) = \bigcup_{i=1}^{m} \mathsf{hdec}(H'[(\mathsf{fn}\ p_i \Rightarrow r_i)\ \overline{u}])$, we have $\forall i, H[(\mathsf{fn}\ p_i \Rightarrow r_i\theta)\ \overline{u\theta}] \in \mathcal{T}_{SC}$ from the induction hypothesis. From (SC2) and (SC1), each $H[(\mathsf{fn}\ p_i \Rightarrow r_i\theta)\ \overline{u\theta}]\ \overline{v}$ is terminating. Hence it is obvious that $t\theta\ \overline{v} \equiv H[(\mathsf{fn}\ p_1 \Rightarrow r_1\theta \mid \cdots \mid p_m \Rightarrow r_m\theta)\ \overline{u\theta}]\ \overline{v}$ is also terminating. From (SC4) and (SC5), there exists a sequence $t\theta\ \overline{v} \xrightarrow[R,\mathsf{fn}]{*} w \notin \mathcal{T}_{SC}$ with $\mathsf{root}(w) \neq \mathsf{fn}$. Since $H[\,]$ is a single head binding context, there exists i such that $H[(\mathsf{fn}\ p_i \Rightarrow r_i\theta)\ \overline{u\theta}]\ \overline{v} \xrightarrow[R,\mathsf{fn}]{*} w \notin \mathcal{T}_{SC}$. It is a contradiction with (SC4). □

Lemma 5.14 Let $\mathsf{fn}\ \overline{p_m} \Rightarrow t_i \in \mathcal{T}_{SC}$ $(i = 1, \ldots, n)$. Then $\mathsf{fn}\ \overline{p_m} \Rightarrow (t_1, \ldots, t_n) \in \mathcal{T}_{SC}$ and $\mathsf{fn}\ \overline{p_m} \Rightarrow a\ \overline{t_n} \in \mathcal{T}_{SC}$ for any $a \in \mathcal{C} \cup \mathcal{V}$.

Proof. First we show that $\mathsf{fn}\ \overline{p_m} \Rightarrow (\overline{t_n}) \in \mathcal{T}_{SC}$ by induction on m. The case of $m = 0$ follows from Lemma 5.11 (2). Suppose that $\mathsf{fn}\ \overline{p_m} \Rightarrow (\overline{t_n}) \equiv \mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow (\overline{t_n})$. From the induction hypothesis, we have $\mathsf{fn}\ \overline{q} \Rightarrow (\overline{t_n}) \in \mathcal{T}_{SC}$. Assume that $\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow (\overline{t_n}) \in \mathcal{T}_{\neg SC}$. Then, from (SC1) and Lemma 5.11 (1), there exist $u, \overline{w} \in \mathcal{T}_{SC}$ such that $(\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow (\overline{t_n}))\ u\ \overline{w} \xrightarrow[R,\mathsf{fn}]{*} (\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow (\overline{t'_n}))\ p\theta\ \overline{w'} \xrightarrow[\mathsf{fn}]{} (\mathsf{fn}\ \overline{q} \Rightarrow (\overline{t'_n\theta}))\ \overline{w'} \in \mathcal{T}_{\neg SC}$. From (SC4) and (SC2), we have $\mathsf{fn}\ \overline{q} \Rightarrow (\overline{t'_n\theta}) \in \mathcal{T}_{\neg SC}$. Since $(\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow t_i)\ u \xrightarrow[R,\mathsf{fn}]{*} \mathsf{fn}\ \overline{q} \Rightarrow t'_i\theta$, it follows from (SC2) and (SC4) that $\mathsf{fn}\ \overline{q} \Rightarrow t'_i\theta \in \mathcal{T}_{SC}$ for each i. Hence, from the induction hypothesis, we have $\mathsf{fn}\ \overline{q} \Rightarrow (\overline{t'_n\theta}) \in \mathcal{T}_{SC}$. It is a contradiction.

Next we show that $\mathsf{fn}\ \overline{p_m} \Rightarrow a\ \overline{t_n} \in \mathcal{T}_{SC}$. The case $a \in \mathcal{C}$ can be proved as similar to the above case. In case $a \in \mathcal{V}$, we have $\mathsf{fn}\ \overline{p_m} \Rightarrow a\ \overline{t_n} \in \mathcal{T}_{SC}$ from Lemma 5.11 (6) and Lemma 5.12 with the empty substitution. □

Lemma 5.15 Let t be a single head binding term. If $\mathsf{hd}(t)\theta \in \mathcal{T}_{SC}$ and $t'\theta \subseteq \mathcal{T}_{SC}$ for any $t' \in \mathsf{args}(t)$ then $t\theta \in \mathcal{T}_{SC}$.

Proof. We proceed by induction on $|t|$.

In case that t has the form of $\mathsf{fn}\ \overline{p} \Rightarrow t'$ with $\mathsf{root}(t') \neq \mathsf{fn}$, the desired property follows from $t \equiv \mathsf{hd}(t)$.

In case that t has the form of $s\ \overline{u_n}$ with $n > 0$. Since $\mathsf{hd}(s) \equiv \mathsf{hd}(t)$ and $\mathsf{args}(s) \subseteq \mathsf{args}(t)$, we have $s\theta \in \mathcal{T}_{SC}$ from the induction hypothesis. Hence $t\theta \equiv s\theta\ \overline{u_n\theta} \in \mathcal{T}_{SC}$ follows from (SC2).

In the remaining case, t has the form of $\mathsf{fn}\ \overline{p} \Rightarrow (\mathsf{fn}\ q \Rightarrow r)\ \overline{u_n}$ with $n > 0$. Then we have $\mathsf{hd}(t) \equiv \mathsf{hd}(\mathsf{fn}\ \overline{p} \Rightarrow \mathsf{fn}\ q \Rightarrow r)$ and $\mathsf{args}(t) = \mathsf{args}(\mathsf{fn}\ \overline{p} \Rightarrow \mathsf{fn}\ q \Rightarrow r) \cup \{\mathsf{fn}\ \overline{p} \Rightarrow u_i \mid i = 1, \ldots, n\}$. From the induction hypothesis, we have $\mathsf{fn}\ \overline{p} \Rightarrow \mathsf{fn}\ q \Rightarrow r \in \mathcal{T}_{SC}$. Hence, by considering Lemma 5.12 with the empty substitution, we obtain $\mathsf{fn}\ \overline{p} \Rightarrow (\mathsf{fn}\ q \Rightarrow r)\ \overline{u_n} \in \mathcal{T}_{SC}$. □

Lemma 5.16 Suppose that for any $\mathsf{fn}\ \overline{p} \Rightarrow a\ \overline{u_n} \in \mathsf{Sub}_{bp}(t)$ with $a \in FV(t)$, there exists $k \leq n$ such that $\mathsf{fn}\ \overline{p} \Rightarrow (a\ \overline{u_k})\theta \in \mathcal{T}_{SC}$. If $t\theta \in \mathcal{T}_{\neg SC}$ then there exist $\mathsf{fn}\ \overline{p} \Rightarrow d\ \overline{v_m} \in \mathsf{Sub}_{bp}(t)$ such that $d \in \mathcal{D}$, $\mathsf{fn}\ \overline{p} \Rightarrow v_i\theta \in \mathcal{T}_{SC}$ for any i, and $\mathsf{fn}\ \overline{p} \Rightarrow d\ \overline{v_k\theta} \in \mathcal{T}_{\neg SC}$ for any $k \leq m$.

Proof. We proceed by induction on $|t|$.

In case of $|\mathsf{hdec}(t)| > 1$, we have $|s| < |t|$ for any $s \in \mathsf{hdec}(t)$. From Lemma 5.13, there exists $s \in \mathsf{hdec}(t)$ such that $s\theta \in \mathcal{T}_{\neg SC}$. Hence the desired property follows from the induction hypothesis.

Suppose that $\mathsf{hdec}(t) = \{t\}$. From Lemma 5.15, there exists $u \in \{\mathsf{hd}(t)\} \cup \mathsf{args}(t)$ such that $u\theta \in \mathcal{T}_{\neg SC}$. In case of $u \in \mathsf{args}(t)$, the desired property follows from the induction hypothesis because of $|u| < |t|$. In case of $u \equiv \mathsf{hd}(t)$ and $|\mathsf{hd}(t)| < |t|$, the desired property follows from the induction hypothesis as similar. Hence it suffices to show the case $\mathsf{hd}(t) \equiv t$ under the assumption $t\theta \in \mathcal{T}_{SC}^{\mathsf{args}}$. Thanks to Lemma 5.14 and $t\theta \in \mathcal{T}_{SC}^{\mathsf{args}}$, we can denote $t \equiv \mathsf{fn}\ \overline{p} \Rightarrow a\ \overline{v_m}$ with $a\theta \in \mathcal{D}$. Then $a \in \mathcal{D} \cup FV(t)$ and each $\mathsf{fn}\ \overline{p} \Rightarrow v_i\theta$ is strongly computable.

Assume that there exists $k \leq n$ such that $\mathsf{fn}\ \overline{p} \Rightarrow (a\ \overline{v_k})\theta \in \mathcal{T}_{SC}$. From Lemma 5.12, we have $t\theta \in \mathcal{T}_{SC}$. It is a contradiction. Moreover $a \in \mathcal{D}$ follows from the assumption for free variables. □

Lemma 5.17 If $\mathsf{fn}\ \overline{p_m} \Rightarrow d\ \overline{v} \in \mathcal{T}_{SC}^{\mathsf{args}} \cap \mathcal{T}_{\neg SC}$ with $d \in \mathcal{D}$ then there exist $\overline{w} \in \mathcal{T}_{SC}$ and a strongly computable substitution θ such that $d\ \overline{v\theta}\ \overline{w} \in \mathcal{T}_{nfun} \cap \mathcal{T}_{SC}^{\mathsf{args}} \cap \mathcal{T}_{\neg SC}$.

Proof. We proceed by induction on m. The case $m = 0$ directly follows from (SC3) with the empty substitution. Suppose that $(\mathsf{fn}\ \overline{p_m} \Rightarrow d\ \overline{v}) \equiv (\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow d\ \overline{v})$.

In case of $\mathsf{fn}\ \overline{q} \Rightarrow d\ \overline{v} \in \mathcal{T}_{\neg SC}$, the desired property follows from the induction hypothesis, because $\mathsf{fn}\ \overline{q} \Rightarrow d\ \overline{v} \in \mathcal{T}_{SC}^{\mathsf{args}}$ follows from Lemma 5.11 (4).

Suppose that $\mathsf{fn}\ \overline{q} \Rightarrow d\ \overline{v} \in \mathcal{T}_{SC}$. From (SC1) and Lemma 5.11 (1), there exist $u, \overline{w} \in \mathcal{T}_{SC}$ such that $(\mathsf{fn}\ p \Rightarrow \mathsf{fn}\ \overline{q} \Rightarrow d\ \overline{v})\ u\ \overline{w} \xrightarrow[R,\mathsf{fn}]{*} (\mathsf{fn}\ p \Rightarrow v')\ p\theta'\ \overline{w'} \xrightarrow[\mathsf{fn}]{} v'\theta'\ \overline{w'} \in \mathcal{T}_{\neg SC}$. Then $(\mathsf{fn}\ \overline{q} \Rightarrow d\ \overline{v\theta'})\ \overline{w} \xrightarrow[R,\mathsf{fn}]{*} v'\theta'\ \overline{w'}$. From (SC4) and (SC2), we have $\mathsf{fn}\ \overline{q} \Rightarrow d\ \overline{v\theta'} \in \mathcal{T}_{\neg SC}$. From $\mathsf{fn}\ p \Rightarrow \overline{q} \Rightarrow d\ \overline{v} \in \mathcal{T}_{SC}^{\mathsf{args}}$, we have $\mathsf{fn}\ p \Rightarrow \overline{q} \Rightarrow v_i \in \mathcal{T}_{SC}$ for each i. From (SC2) and (SC4), we have $(\mathsf{fn}\ p \Rightarrow \overline{q} \Rightarrow v_i)\ u \xrightarrow[R,\mathsf{fn}]{*} \mathsf{fn}\ \overline{q} \Rightarrow v_i\theta' \in \mathcal{T}_{SC}$. Hence $\mathsf{fn}\ \overline{q} \Rightarrow d\ \overline{v\theta'} \in \mathcal{T}_{SC}^{\mathsf{args}} \cap \mathcal{T}_{\neg SC}$. From the induction hypothesis, there exist a strongly computable substitution θ'' and $\overline{w''} \in \mathcal{T}_{SC}$ such that $d\ \overline{v\theta'\theta''}\ \overline{w''} \in \mathcal{T}_{nfun} \cap \mathcal{T}_{SC}^{\mathsf{args}} \cap \mathcal{T}_{\neg SC}$. Here, strong computability of θ' follows from Lemma 5.11 (3). Thanks to the variable convention, the substitution $\theta = \theta' \cup \theta''$ satisfies the desired

property. □

Lemma 5.18 If ATRFP R is not terminating then $\{d\ \overline{t} \in \mathcal{T}^{cls}_{nfun} \cap \mathcal{T}^{\mathsf{args}}_{SC} \cap \mathcal{T}_{\neg SC} \mid d \in \mathcal{D}\} \neq \emptyset$.

Proof. From Proposition 2.2, 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 t be a minimal size term in $\mathcal{T}^{cls} \cap \mathcal{T}_{\neg SC}$.

From Lemma 5.11 (6) and the minimality, the term t with the empty substitution satisfy the condition of Lemma 5.16. Hence there exist $d \in \mathcal{D}$ and $\mathsf{fn}\ \overline{p} \Rightarrow d\ \overline{v} \in \mathsf{Sub}_{bp}(t)$ such that $\mathsf{fn}\ \overline{p} \Rightarrow d\ \overline{v} \in \mathcal{T}^{\mathsf{args}}_{SC} \cap \mathcal{T}_{\neg SC}$. Therefore the desired property follows from Lemma 5.17. □

This lemma characterizes minimal counterexamples for termination. Next, we bridge such counterexamples. Then an infinite static dependency chain emerges.

Lemma 5.19 Let R be an ATRFP. For any $d \in \mathcal{D}$ and $d\ \overline{t} \in \mathcal{T}^{cls}_{nfun} \cap \mathcal{T}^{\mathsf{args}}_{SC} \cap \mathcal{T}_{\neg SC}$, there exist $u^\sharp \to v^\sharp \in Act(SDP(R))$ and term substitution θ such that $d^\sharp\ \overline{t} \xrightarrow[R,\mathsf{fn}]{*} u^\sharp\theta$ and $u\theta, v\theta \in \mathcal{T}^{cls}_{nfun} \cap \mathcal{T}^{\mathsf{args}}_{SC} \cap \mathcal{T}_{\neg SC}$.

Proof. From (SC1) and Lemma 5.11 (1), there exists a sequence: $d\ \overline{t} \xrightarrow[R,\mathsf{fn}]{>\varepsilon,*} d\ \overline{t'} \xrightarrow[R]{\varepsilon} t'' \in \mathcal{T}_{\neg SC}$. Hence there exist $l \to r \in Act(R)$ and θ' such that $d\ \overline{t'} \equiv l\theta'$ and $t'' \equiv r\theta'$. From (SC4), we have $l\theta' \in \mathcal{T}^{\mathsf{args}}_{SC}$. Since R is accessible, it follows from the axiom of accessible function, the variable convention and Lemma 5.11 (5) that r and θ' satisfy the condition of Lemma 5.16. Hence there exist $g \in \mathcal{D}$ and $\mathsf{fn}\ \overline{p} \Rightarrow g\ \overline{v_m} \in \mathsf{Sub}_{bp}(r)$ such that $\mathsf{fn}\ \overline{p} \Rightarrow v_i\theta' \in \mathcal{T}_{SC}$ for any i, and $\mathsf{fn}\ \overline{p} \Rightarrow g\ \overline{v_k\theta'} \in \mathcal{T}_{\neg SC}$ for any $k \leq m$. From Lemma 5.17, there exist θ'' and $\overline{w_n} \in \mathcal{T}_{SC}$ such that $g\ \overline{v_m\theta'\theta''}\ \overline{w_n} \in \mathcal{T}_{nfun} \cap \mathcal{T}^{\mathsf{args}}_{SC} \cap \mathcal{T}_{\neg SC}$. We define θ as $\theta(z_i) = w_i$ for fresh variables $\overline{z_n}$; otherwise $\theta(x) = \theta''(\theta'(x))$. From Lemma 5.11 (4), we have $v_i\theta \equiv v_i\theta' \in \mathcal{T}_{SC}$ for any i. Then we have $l^\sharp \to g^\sharp\ \overline{v_m}\ \overline{z_n} \in Act(SDP(R))$ and $(g\ \overline{v_m}\ \overline{z_n})\theta \in \mathcal{T}^{cls}_{nfun} \cap \mathcal{T}^{\mathsf{args}}_{SC} \cap \mathcal{T}_{\neg SC}$. Since $l\theta \equiv l\theta' \in \mathcal{T}^{cls}_{nfun} \cap \mathcal{T}^{\mathsf{args}}_{SC} \cap \mathcal{T}_{\neg SC}$, the desired property holds. □

All preparations are complete so that we can now show the fundamental theorem of the static dependency pair method.

Proof of Theorem 5.5. Assume that R is not terminating. From Lemma 5.18, there exists $t \in \mathcal{T}^{cls}_{nfun} \cap \mathcal{T}^{\mathsf{args}}_{SC} \cap \mathcal{T}_{\neg SC}$. By applying Lemma 5.19 repeatedly, we obtain an infinite static dependency chain. This is a contradiction. □

6. Proving Non-Loopingness

When proving termination by dependency pair methods, not only our static dependency pair methods, non-loopingness[†] should be shown for each recursion component (cf. Corollary 5.9). To prove the non-loopingness, the notions of subterm criterion and reduction pair have been proposed. The subterm criterion was introduced on TRSs [10], and slightly improved by extending the subterms permitted by the criterion on simply-typed TRSs (STRSs) [16], and extended on higher-order rewrite systems (HRSs) [18]. Reduction pairs [15] are an abstraction of the notion of the weak-reduction orders [1]. In [19], we extended both notions to TRFPs without functional abstraction. This result also works well on TRFPs. In this section, we introduce these notions.

Intuitively, a static recursion component whose non-loopingness is proved by the subterm criterion guarantees that the function is explicitly recursively programmed on data types, while a component whose non-loopingness is proved by a reduction pair guarantees that the function is appropriately recursively programmed by using a decreasing function.

Definition 6.1 A pair $(\gtrsim, >)$ of relations on terms is a *reduction pair* if $\gtrsim$ and $>$ satisfy the following properties:

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

Definition 6.2 A set C of static dependency pairs satisfies the *subterm criterion* if there exists a function π from $\mathcal{D}$ to non-empty sequences of positive integers such that:

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

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

Theorem 6.3 Let R be a finite ATRFP. 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. Based on Theorem 5.5, we can prove the claim as similar to Theorem 5.3 in [19].

Example 6.4 As finale of the running example, we will prove the termination of ATRFP R_{ack}. We take $\pi(\mathrm{prec}) = 1$. Then the only static recursion component satisfies the subterm criterion in the underlined position below.

$$\{\mathrm{prec}^\sharp\ (\underline{\mathrm{suc}\ x})\ z\ f \to \mathrm{prec}^\sharp\ \underline{x}\ z\ f\}$$

Hence from Theorem 6.3, the static recursion component is non-looping. Therefore the termination of R_{ack} follows from

[†]In the research area of term rewriting systems, there is a different use for the term "non-loopingness": there is no reduction sequence such as $t \xrightarrow{+} C[t\theta]$.

Corollary 5.9.

Example 6.5 We will prove the termination of ATRFP R_{len} given in Example 2.1. Then the static dependency pairs $SDP(R_{\mathrm{len}})$ consist of the following two pairs:

$$\begin{cases} \mathrm{foldl}^\sharp\ f\ e\ (\mathrm{cons}\ (x, xs)) \to \mathrm{foldl}^\sharp\ f\ (f\ (e, x))\ xs \\ \mathrm{len}^\sharp\ xs \to \mathrm{foldl}^\sharp\ (\mathsf{fn}\ (x, y) \Rightarrow \mathrm{suc}\ x)\ 0\ xs \end{cases}$$

We take $\pi(\mathrm{foldl}) = 3$. Then the only static recursion component satisfies the subterm criterion in the underlined position below.

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

Hence from Theorem 6.3, the static recursion component is non-looping. Therefore the termination of R_{len} follows from Corollary 5.9.

Example 6.6 We will prove the termination of TRFP R, which represents a typical higher-order function "filter" and its application.

$$\begin{cases} \mathrm{filter}\ p\ \mathrm{nil} \to \mathrm{nil} \\ \mathrm{filter}\ p\ (\mathrm{cons}\ (x, xs)) \to \\ \quad \mathrm{if}\ p\ x\ \mathrm{then\ cons}\ (x, \mathrm{filter}\ p\ xs)\ \mathrm{else\ filter}\ p\ xs \\ \mathrm{plist}\ xs \to \mathrm{filter}\ (\mathsf{fn}\ 0 \Rightarrow \mathrm{false} \mid \mathrm{suc}\ x \Rightarrow \mathrm{true})\ xs \end{cases}$$

Here, the expression

$$\mathrm{if}\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3$$

is a syntax sugar for the following term:

$$(\mathsf{fn}\ \mathrm{true} \Rightarrow e_2 \mid \mathrm{false} \Rightarrow e_3)\ e_1$$

Similar to Example 4.9, we can show that TRFP R is accessible. Then the static dependency pairs consist of the following two pairs:

$$\begin{cases} \mathrm{filter}^\sharp\ p\ (\mathrm{cons}\ (x, xs)) \to \mathrm{filter}^\sharp\ p\ xs \\ \mathrm{plist}^\sharp\ xs \to \mathrm{filter}^\sharp\ (\mathsf{fn}\ 0 \Rightarrow \mathrm{false} \mid \mathrm{suc}\ x \Rightarrow \mathrm{true})\ xs \end{cases}$$

We take $\pi(\mathrm{filter}) = 2$. Then the only static recursion component satisfies the subterm criterion in the underlined position below.

$$\{\mathrm{filter}^\sharp\ p\ \underline{(\mathrm{cons}\ (x, xs))} \to \mathrm{filter}^\sharp\ p\ \underline{xs}\}$$

Hence from Theorem 6.3, the static recursion component is non-looping. Therefore the termination follows from Corollary 5.9.

In Sect. 1, we said that the polymorphic-typed Combinatory Logic, in which functional abstraction with pattern is permitted, is an example that shows the strong efficacy of the SDP-method. Finally together with other well-known combinators [2], we give an elegant termination proof by the SDP-method.

Example 6.7 Let R be the following TRFP:

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

Since any variable occurs in an argument position on the left-hand sides, TRFP R is trivially accessible. Since $SDP(R) = \emptyset$ and hence $SRC(R) = \emptyset$, the termination of R follows from Corollary 5.9.

7. Concluding Remarks

In this paper, we have extended the SDP-method onto TRFPs, in which functional abstraction with pattern is permitted. Since the syntax of TRFP is very close to SML-like functional programs, from our result we expect the effective applicability to verification for existing functional programs.

On the other hand, in order that the SDP-method gives full play to its ability, it is indispensable to design reduction orders, the argument filtering method, and the notion of usable rules.

An effective and practicable reduction order, namely higher-order recursive path orderings, was introduced [4], [5], [11]. Since these orderings do not handle functional abstraction with pattern, we will extend these orderings to TRFPs in the future.

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]. Since these results do not handle functional abstraction with pattern and polymorphic type systems, we will extend the method to TRFPs in the future.

The notion of usable rules optimizes the constraints generated by the dependency pair methods. This analysis was first conducted for TRSs [7], [10], and has been extended to STRSs [17], [21], and to HRSs [22]. Since these results do not handle functional abstraction with pattern and polymorphic type systems, we will extend the notion to TRFPs in the future.

Acknowledgments

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

References

[1] T. Arts and J. Giesl, "Termination of Term Rewriting Using Dependency Pairs," Theoretical Computer Science, vol.236, no.1-2, pp.133–178, 2000.
[2] K. Bimbó, Combinatory Logic: Pure, Applied and Typed, Chapman and Hall/CRC, 2011.
[3] F. Blanqui, J.-P. Jouannaud, and M. Okada, "Inductive-Data-Type Systems," Theoretical Computer Science, vol.272, no.1-2, pp.41–68, 2002.

[4] F. Blanqui, "Computability Closure: Ten Years Later," In Essay in Honour of Jean-Pierre Jouannaud's 60 Birthday, LNCS 4600, pp.68–88, 2007.
[5] 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, LNCS 5213 (CSL2008), pp.1–14, 2008.
[6] N. Dershowitz, "Orderings for Term-Rewriting Systems," Theoretical Computer Science, vol.17, no.3, pp.279–301, 1982.
[7] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke, "Mechanizing and Improving Dependency Pairs," Journal of Automated Reasoning, vol.37, no.3, pp.155–203, 2006.
[8] J.-Y. Girard, "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur," PhD thesis, University of Paris VII, 1972.
[9] J.R. Hindley and J.P. Seldin, Introduction to Combinators and λ-Calculus, Cambridge Univ. Press, 1986.
[10] N. Hirokawa and A. Middeldorp, "Tyrolean Termination Tool: Techniques and Features," Information and Computation, vol.205, no.4, pp.474–511, 2007.
[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, V. van Oostrom, and R. de Vrijer, "Lambda Calculus with Patterns," Theoretical Computer Science, vol.398, no.1-3, pp.16–31, 2008.
[13] C. Kop, "Higher Order Termination," Ph.D. thesis, Vrije Universiteit Amsterdam, 2012.
[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," Journal of 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," 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, 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, 2009.
[19] K. Kusakari, "Static Dependency Pair Method in Rewriting Systems for Functional Programs with Product, Algebraic Data, and ML-Polymorphic Types," IEICE Trans. Inf. & Syst., vol.E96-D, no.3, pp.472–480, 2013.
[20] T. Nipkow, "Higher-order Critical Pairs," Proc. 6th Annual IEEE Symposium on Logic in Computer Science, pp.342–349, 1991.
[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., vol.J90-D, no.4, pp.978–989, 2007. (in Japanese)
[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 Interpretations of Functionals of Finite Type," Journal of Symbolic Logic, vol.32, no.2, 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. He transferred to Gifu University in 2014 as a professor. His research interests include term rewriting systems, program theory, and automated theorem proving. He is a member of IPSJ and JSSST.