Static Dependency Pair Method in Functional Programs

KUSAKARI Keiichirou
Informatics Course, Faculty of Engineering, Gifu University

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 $\lambda$-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:

\[
\text{prec} : \text{Nat} \rightarrow \text{Nat} \rightarrow (\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat}
\]

The function can be represented by the following TRFP:

\[
\begin{cases}
\text{prec } 0 \ z \ f \rightarrow z \\
\text{prec } (\text{suc} \ x) \ z \ f \rightarrow f \ x \ (\text{prec } x \ z \ f)
\end{cases}
\]

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

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

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,

\[
\text{prec}^2 (\text{suc} \ x) \ z \ f \rightarrow \text{prec}^2 x \ z \ f
\]

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.

\[
\text{prec}^2 (\text{suc} \ x) \ z \ f \rightarrow \text{prec}^2 x \ z \ f
\]

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:

\[
\begin{cases}
\text{S } f \ g \ x \rightarrow f \ x \ (g \ x) \\
\text{K } x \ y \rightarrow x
\end{cases}
\]

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 computability in typed systems. The notion was introduced to show the termination of typed $\lambda$-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
computation.

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

\[ \text{foo } (\text{bar } f) \rightarrow f \ (\text{bar } f) \]

Although this system is typable under foo, \( f : \alpha \rightarrow \beta \) and bar : \((\alpha \rightarrow \beta) \rightarrow \alpha\), it has the self-loop: foo (bar foo) \rightarrow 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 \( \text{bar } t \)) 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 Section 5). From a technical viewpoint, our soundness proof is an extension of the termination proof of typed \( \lambda \)-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 Section 3, we present ways for the technical decomposition of terms. In Section 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 Section 5, we show the SDP-method on TRFPs. In Section 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 Section 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 \( \overline{s} \) for a sequence of either \( a_1, \ldots, a_n \) or \( a_1 \ldots a_n \).

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

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

A functional type or higher-order type is a type of the form \((\sigma_1 \rightarrow \sigma_2)\). We denote by \( S_{nfun} \) 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 \( c \) is either a product type or a type of the form \( c(\sigma_1, \ldots, \sigma_n) \). For \( c \in TC \) with \( \text{arity}(c) = 0 \), we simply denote \( c \) by \( c \). To minimize the number of parentheses, we assume \( \rightarrow \) is right-associative and \( \rightarrow \) has lower precedence than \( \times \). We shortly denote \( \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma_0 \) by \( \overline{\sigma_0} \rightarrow \sigma_0 \). Under these conventions, any type \( \sigma \) is uniquely denoted by the form \( \overline{\sigma_0} \rightarrow \sigma_0 \) with \( \sigma_0 \in S_{nfun} \), which we call the canonical form. A type \( \sigma \) is said to be closed if no type variable occurs in \( \sigma \). A type \( \sigma \) is said to be an instance of a type \( \sigma' \), denoted by \( \sigma' \supseteq \sigma \), if there is a type substitution \( \xi \) such that \( \sigma = \xi(\sigma') \).

Let \( D, C \), and \( V \) be a set of defined symbols, constructors, and term variables, respectively. A type environment is a pair \((\Sigma, \Gamma)\) of functions \( \Sigma : C \cup D \rightarrow S \) and \( \Gamma : V \rightarrow S \). For given type environment \((\Sigma, \Gamma)\), 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 \( c \in C, \Sigma(c) \supseteq \overline{\sigma_0} \rightarrow \sigma_0 \) 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 \((p_1^{\sigma_1} \times \cdots \times \sigma_n) \in \mathcal{P}(\Sigma, \Gamma) \).

For given type environment \((\Sigma, \Gamma)\), we define the set
\( T(\Sigma, \Gamma) \) of typed terms (terms for short) as follows:

- If \( \Gamma(x) = \sigma \) then \( x^\sigma \in T(\Sigma, \Gamma) \).
- If \( \Sigma(f) \supseteq \sigma \) then \( f^\sigma \in T(\Sigma, \Gamma) \).
- If \( t_1^\sigma_1, \ldots, t_n^\sigma_n \in T(\Sigma, \Gamma) \) then \((t_1^\sigma_1, \ldots, t_n^\sigma_n)^{x_1 \times \cdots \times x_n} \in T(\Sigma, \Gamma) \).
- If \( p_1^\sigma, \ldots, p_m^\sigma \in \mathcal{P}(\Sigma, \Gamma) \) and \( r_1^\sigma, \ldots, r_m^\sigma \in T(\Sigma, \Gamma) \) then \((f_n p_1 \Rightarrow r_1^\sigma \mid \cdots \mid p_m \Rightarrow r_m^\sigma)^{\sigma \rightarrow \sigma'} \in T(\Sigma, \Gamma) \).
- If \( t^\sigma \rightarrow t''^\sigma, u^\sigma \in T(\Sigma, \Gamma) \) then \((t \ t')^\sigma \in T(\Sigma, \Gamma) \).

For \( t = (t_1^\sigma_1, \ldots, t_n^\sigma_n)^{x_1 \times \cdots \times x_n} \), we identify \( t \) as \( t_1^\sigma_1 \) in case of \( n = 1 \), and \( t \equiv ()^\mu \) in case of \( n = 0 \), where unit is a special type constructor. The \( \alpha \)-equivalence of terms is denoted by \( \equiv \). The set of free 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 \( fn \ p_1 \Rightarrow fn \ p_2 \Rightarrow \cdots \Rightarrow fn \ p_m \Rightarrow r \) by \( fn \ p_m \Rightarrow r \) or \( fn \ p \Rightarrow r \), and \( u \ t_1 \cdots \ t_n \) by \( u \ t_n \) or \( u \ t \). 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, \((fn \ p \Rightarrow r)^\theta \equiv fn \ p \Rightarrow r^\theta \) holds for any term substitution \( \theta \). The set \( Pos(t) \) of positions, which are sequences of natural numbers, in a term \( t \) is defined as follows:

- \( Pos(\Gamma_n) = \{ \varepsilon \} \cup \bigcup_{i=1}^n \{ iq \mid q \in Pos(t_i) \} \) if \( a \in D \cup C \cup V \)
- \( Pos((t_1, \ldots, t_n)) = \{ \varepsilon \} \cup \bigcup_{i=1}^n \{ iq \mid q \in Pos(t_i) \} \)
- \( Pos(fn \ p_1 \Rightarrow r_1 \mid \cdots \mid p_m \Rightarrow r_m) = \{ \varepsilon \} \cup \bigcup_{i=1}^m \{ iq \mid q \in Pos(t_i) \} \)
- \( Pos(u \ t_m) = \{ \varepsilon \} \cup \{ 0q_0 \mid q_0 \in Pos(u) \} \cup \bigcup_{i=1}^m \{ iq \mid q \in Pos(t_i) \} \) if \( n > 0 \) where \( u \equiv (fn \ p_1 \Rightarrow r_1 \mid \cdots \mid p_m \Rightarrow r_m) \)

The prefix order < on positions is defined by \( p < q \) iff \( pw = q \) for some \( w \neq \varepsilon \). The position \( \varepsilon \) 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 \( q \notin Pos(t) \). 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 \( (t)_q = tp \) whenever \( t_q = (t_1, \ldots, t_n) \), and interpret \( (t)_q = fn \) whenever \( t_q \) is a functional abstraction. Specially, the root symbol \( (t)_e \) is also denoted by \( root(t) \). The size \( |t| \) of \( t \) is defined as the cardinality of \( Pos(t) \). We denote the \( (proper) subterm relation by \( \subseteq_{sub} \) (\( \supseteq_{sub} \)). We define \( t^\sigma \in T_{nf\mu} \) iff \( \sigma \) is not a functional type, and \( t \in T_{dis} \) iff \( \sigma \) is closed for any \( v^\sigma \subseteq_{sub} t \).

As a matter of course, we fix a type environment \( \Sigma \) for defined symbols and constructors. A triple \((\ell^\sigma, r^\sigma, \Gamma)\) is said to be a rewrite rule, denoted by \( \ell^\sigma \rightarrow r^\sigma : \Gamma \) \((\ell^\sigma \rightarrow r^\sigma \) for short) if \( \ell^\sigma \) and \( r^\sigma \) are terms of the same type \( \sigma \) under the type environment \((\Sigma, \Gamma), root(l) \in D, \) and \( FV(l) \supseteq FV(r) \). A term rewriting model for functional programs (TRFP) is a set of rewrite rules. For any rewrite rule \( \ell^\sigma \rightarrow r^\sigma \), we define the set \( Act(l \rightarrow r) \) of actual rewrite rules as: \( u^\sigma \rightarrow v^\sigma \in Act(l^\sigma \rightarrow r^\sigma) \) iff there is a type substitution \( \xi \) such that \( u \equiv l^\xi \mathcal{P} \xi \), \( v \equiv r^\xi \mathcal{P} \xi \), and \( \xi(\sigma) = \mathcal{P} \xi \rightarrow \sigma' \), where each \( z_i^\sigma \) is a fresh variable. We also denote by \( Act(R) \) the set \( \bigcup_{l \rightarrow r \in R} Act(l \rightarrow r) \).

The relation \( \rightarrow_R \) of a TRFP \( R \) is defined by \( s \rightarrow_R t \) iff \( s \equiv C[\theta] \) and \( t \equiv C[r\theta] \) for some actual rewrite rule \( l \rightarrow r \in Act(R) \), leaf context \( C[\cdot] \), and term substitution \( \theta \). We also define \( s \rightarrow^R t \) iff there exists a context \( C[\cdot] \) such that \( s \equiv C[fn \ p_1 \Rightarrow r_1 \mid \cdots \mid p_m \Rightarrow r_m \] \( p_\theta \) and \( t \equiv C[r_\theta] \). The rewrite relation \( \rightarrow^R \) is the union of \( \rightarrow_R \) and \( \rightarrow^\mu \). Especially, we denote \( s \rightarrow^R t \) if a rewrite \( s \rightarrow^R t \) occurs at the root position; otherwise we denote \( s \rightarrow^R t \) when \( t \) is not a root position.

Example 2.1 Consider the following TRFP \( R_{len} \):

\[
\begin{align*}
&\{ foldl \ f \ e \ nil \Rightarrow e \\
&foldl \ f \ e \ (\cons \ (x, xs)) \Rightarrow foldl \ f \ (f \ (e, x)) \ (xs) \\
&len \ xs \Rightarrow foldl \ (fn \ (x, y) \Rightarrow suc \ x) \ 0 \ (xs)
\end{align*}
\]

The function \( foldl : (\alpha \times \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow list(\beta) \rightarrow \alpha \) is a typical higher-order function that is widely used in existing functional programs, where \( \alpha \) and \( \beta \) 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 \( \text{len}(\cons \ (t, nil)) \) as follows:

\[
\begin{align*}
&\rightarrow^-R foldl \ (fn \ (x, y) \Rightarrow suc \ x) \ 0 \ (\cons \ (t, nil)) \\
&\rightarrow^-R foldl \ (fn \ (x, y) \Rightarrow suc \ x) \ (0, t) \ nil \\
&\rightarrow^-R foldl \ (fn \ (x, y) \Rightarrow suc \ x) \ (suc \ 0) \ nil \\
&\rightarrow^-R suc \ 0
\end{align*}
\]

A term \( t \) is said to be terminating if there exists no infinite rewrite sequence of \( \rightarrow^R \) 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 $\xi$, we have $s \frac{R}{\xi} t \Rightarrow s \frac{R}{\xi} t$. 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 terms into one term. For example, we can bundle terms ($\text{fn } 0 \Rightarrow 0$) and ($\text{fn suc } x \Rightarrow x$) in the term ($\text{fn } 0 \Rightarrow 0 \ | \ \text{suc } x \Rightarrow x$). To uncouple a bundle such as:

$$\text{fn } d$$

we introduce the notion of the head decomposition.

For example, we can bundle terms ($\text{fn suc } x \Rightarrow x$) and the arguments $\text{fn suc } x \Rightarrow x$. To uncouple a bundle such as:

$$\text{hd}(\text{fn } 0 \Rightarrow 0 \ | \ \text{suc } x \Rightarrow x) = \{\text{fn } 0 \Rightarrow 0, \text{fn suc } x \Rightarrow x\},$$

we introduce the notion of the head decomposition.

Definition 3.1 We define the function $\text{hdec}$ as follows:

$$\text{hdec}(\text{fn } p_1 \Rightarrow r_1 \ | \cdots | p_m \Rightarrow r_m) = \bigcup_{i=1}^{m} \text{hdec}(\text{fn } p_i \Rightarrow r_i)$$

if $m > 1$

$$\text{hdec}(\text{fn } p \Rightarrow r) = \{(\text{fn } p \Rightarrow r') \bigcup r' \in \text{hdec}(r)\}$$

$$\text{hdec}(t) = \{t\} \quad \text{otherwise}$$

A term $t$ is said to be a single head binding term if $t \in \text{hdec}(t')$ 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 $\text{hd}$ as follows:

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

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 $d \in D$ 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 strongly computable, 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 $d_{\text{fn } n}$ ($d = d t_1 \cdots t_n$) such that $d \in D$ and each argument $t_i$ is strongly computable is also 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 $\text{fn } p \Rightarrow d_{\text{fn } n}$. The most natural way may be to decompose the terms into $\text{fn } n \Rightarrow \text{fn } p \Rightarrow d_{\text{fn } n}$ and the arguments $t_1, \ldots, t_n$. Then we have $\text{fn } n \Rightarrow \text{fn } p \Rightarrow d_{\text{fn } n}$ 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 $p$ may become free variables. Another problem is that the size of $\text{fn } n \Rightarrow \text{fn } p \Rightarrow d_{\text{fn } n}$ may be larger than the size of $\text{fn } p \Rightarrow d_{\text{fn } n}$. To avoid such difficulties, we technically interpret $\text{fn } p \Rightarrow d_{\text{fn } n}$ as arguments of $\text{fn } p \Rightarrow d_{\text{fn } n}$.

Finally, we introduce the notion of bind preserving subterms, which embody $\text{hdec}$, $\text{hd}$, and $\text{args}$.

Definition 3.3 The function $\text{args}$ is defined as follows:

- $\text{args}(a_{\text{fn } n}) = \{a_{\text{fn } n}\}$
- $\text{args}(\langle t_1, \ldots, t_n \rangle) = \{t_1, \ldots, t_n\}$
- $\text{args}(\langle\text{fn } p_1 \Rightarrow r_1, \cdots, \text{fn } p_m \Rightarrow r_m\rangle) = \{t_1, \ldots, t_n\} \cup \bigcup_{i=1}^{m} \text{args}(r_i)$

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

$$t \equiv (\text{fn } p_1 \Rightarrow a_1 (a_2, a_3) \mid (t_2 \equiv (\text{fn } p_2 \Rightarrow (\text{fn } p_3 \Rightarrow a_4 a_5) \mid (t_3 \equiv (\text{fn } p_2 \Rightarrow a_6 a_7)\mid (t_4 \equiv (\text{fn } p_2 \Rightarrow a_8))$$

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

$$t_1 \equiv (\text{fn } p_1 \Rightarrow a_1 (a_2, a_3) \mid (\text{fn } p_2 \Rightarrow a_4 a_5) \mid (\text{fn } p_3 \Rightarrow a_6))$$

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

$$\text{hd}(t_1) = \text{fn } p_1 \Rightarrow a_1 (a_2, a_3)$$

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

$$t_4 \equiv \text{fn } p_1 \Rightarrow (a_2, a_3)$$

$$t_5 \equiv \text{fn } p_2 \Rightarrow \text{fn } p_3 \Rightarrow a_5$$

$$t_6 \equiv \text{fn } p_2 \Rightarrow a_7$$

$$t_7 \equiv \text{fn } p_2 \Rightarrow a_8 \mid (\text{fn } p_3 \Rightarrow a_9)$$
The set of bind preserving subterms $\text{Sub}_{\beta\eta}(t)$ consists of ten terms: $\text{hd}(t_1), \text{hd}(t_2), \text{hd}(t_3), t_4, t_5, t_6,$ and two elements of $\text{args}(t_4)$:

$$\text{fn } p_1 \Rightarrow a_2 \quad \text{fn } p_1 \Rightarrow a_3$$

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

$$\text{fn } p_5 \Rightarrow a_8 \quad \text{fn } p_6 \Rightarrow a_9$$

### 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 (Section 4.2). It will refer only to the abstract framework (Section 4.1).

#### 4.1 Abstract Framework

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

- (SC1) For any $t \in \mathcal{T}^{\text{cls}}$, if $P(t)$ then $\text{SN}(t)$.
- (SC2) For any $t'^{\rightarrow} \rightarrow t'^{\rightarrow}$, $u'^{\rightarrow} \in \mathcal{T}^{\text{cls}}$, if $P(t)$ and $P(u)$ then $P(t'u)$.
- (SC3) For any $t'^{\rightarrow} \rightarrow t'^{\rightarrow}$, if $\forall u'^{\rightarrow} \in \mathcal{T}^{\text{cls}}[P(u) \Rightarrow P(t'u)]$ then $P(t)$.
- (SC4) For any $t, u \in \mathcal{T}^{\text{cls}}$, if $P(t)$ and $t \overset{\text{R}, u}{\rightarrow} u$ then $P(u)$.
- (SC5) For any $t \in \mathcal{T}^{\text{cls}}_{\text{fam}}$, if $\forall u \in \mathcal{T}^{\text{cls}}(\{ t' \mid t \overset{\text{R}, u}{\rightarrow} t' \} \cup T).P(u)$ then $P(t)$, where $T = \text{args}(t)$ if root($t$) \neq \text{fn};$ otherwise $T = \emptyset$.

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

**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 $t, u \in \mathcal{T}^{\text{cls}}$, if $u \in A(t)$ and $t \in \mathcal{T}^{\text{args}}_{\text{SC}}$ then $\text{SC}(u)$.
- (Acc2) For any $t, u \in \mathcal{T}^{\text{cls}}$ and term substitution $\theta$, if $u \in A(t)$ then $u\theta \in A(t\theta)$.
- (Acc3) For any $t, u \in \mathcal{T}$ and type substitution $\xi$, 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 $\text{Acc}$ such that

- $C$ is accessible, that is, $t_i \in \text{Acc}(c_\mathcal{T})$ for any $i$ and $c \in C$, and
- for any rule $l \rightarrow r \in R$ and $a_\mathcal{T} \triangleleft_{\text{sub}} r$ with $a \in \text{FV}(r)$, there exists $k (\leq n)$ such that $a_\mathcal{T} \in \text{Acc}(l)$.

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 $R = \{ \text{foo (bar } f \rightarrow f \text{ (bar } f) \}$ displayed in the introduction is not accessible.

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

In case of $SC(\text{bar } foo)$, since $\text{foo} \in \text{Acc}(\text{foo (bar } foo))$ 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 $u \in T^{\text{cls}}$ such that $SC(\text{u})$ and $\neg SC(\text{foo } u)$. From (SC5), (SC1) and (SC4), there is a reduction sequence $\text{foo } u \overset{\text{R}, n}{\rightarrow} \text{foo } u'$ (bar $u'$) such that $SC(\text{bar } u')$ and $\neg SC(\text{bar } u')$. Since $u' \in \text{Acc}(\text{foo (bar } u'))$ by (Acc2), $SC(\text{u'})$ follows from (Acc1). However, $\neg SC(\text{u'})$ 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 $\text{pcomp}$ as follows:

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

$$\text{pcomp}(\sigma) = \{ \sigma \} \quad \text{otherwise}$$

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

$$\{ \sigma \mid \exists \sigma' \in PT, \sigma' \preceq \sigma \} \cup \{ \sigma \mid \sigma \text{ is a product type} \}$$
A well-founded quasi order \(\succsim\) on types is said to be a peeling order if the following properties hold:

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

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

- \(\text{args}(t) \subseteq \text{Sub}_{\succsim}(t)\).
- If \((t_1, \ldots, t_n) \in \text{Sub}_{\succsim}(t)\) then \(\forall i, t_i \in \text{Sub}_{\succsim}(t)\), and
- If \(u \equiv (a \ u^n)\sigma \in \text{Sub}_{\succsim}(t)\), \(\sigma \in \text{PT}\), and \(\forall \sigma' \in \text{pcomp}(\sigma_1), \sigma \succeq \sigma'\) then \(u^\sigma \in \text{Sub}_{\succsim}(t)\).

**Definition 4.6** For a set \(\text{PT}\) of peeling types and peeling order \(\succsim\), we define \(\text{SC}(t)\) as follows:

- In case of \(t^\sigma \in \text{T}^{\text{cls}}_{\text{fun}}\) and \(\sigma \notin \text{PT}\), \(\text{SC}(t)\) is defined as \(\text{SN}(t)\).
- In case of \(t^\sigma \in \text{T}^{\text{cls}}_{\text{fun}}\) and \(\sigma \in \text{PT}\), \(\text{SC}(t)\) is defined as \(\text{SN}(t)\) and \(\text{SC}(u)\) for any \(u^\sigma \in \text{T}^{\text{cls}} \cap \{t' \mid t \succsim t'\} \cup T\) such that \(\sigma \succeq \sigma'\), where \(T = \text{args}(t)\) if \(\text{root}(t) \neq \text{fn}\); otherwise \(T = \emptyset\).
- In case of \(t^\sigma_1 \to t^\sigma_2 \in \text{T}^{\text{cls}}\), \(\text{SC}(t)\) is defined as \(\text{SC}(t^\sigma_1)\) for all \(u^\sigma_1 \in \text{T}^{\text{cls}}\) with \(\text{SC}(u)\).

The well-definedness of \(\text{SC}\) can be shown as similar to [19]. We note that the value of \(\text{SC}(t)\) for non-terminating term \(t\) is set to false, and the value of \(\text{SC}(t)\) for terminating term \(t^\sigma\) is inductively defined on \((\sigma, t)\) with respect to the lexicographic combination of \(\langle \succsim, \prec \rangle \cup T_{\text{sub}}\).

**Definition 4.7** For a set \(\text{PT}\) of peeling types and a peeling order \(\succsim\), we define the function \(\text{Acc}\) as follows:

\[
\text{Acc}(t) = \text{Sub}_{\succsim}(t) \cup \{u \mid t \succsim u \in \text{T}^{\text{cls}}_{\text{fun}}, \sigma \notin \text{PT}\}
\]

The predicate \(\text{SC}\) given in Definition 4.6 is a strong computability predicate, and the function \(\text{Acc}\) 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]. \(\square\)

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

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

- we have \(f, e, \text{cons}\ (x,\ xs) \in \text{Sub}_{\succsim}(t)\) because of \(\text{args}(t) \subseteq \text{Sub}_{\succsim}(t)\),
- we have \((x,\ xs) \in \text{Sub}_{\succsim}(t)\) because of \((\text{cons}\ (x,\ xs)) \cdot \text{list}(\alpha)) \in \text{Sub}_{\succsim}(t), \text{pcomp}(\alpha \times \text{list}(\alpha)) = \{\alpha, \text{list}(\alpha)\}, \text{list}(\alpha) \succeq_{\text{rpo}} \alpha\) and \(\text{list}(\alpha) \succeq_{\text{rpo}} \text{list}(\alpha)\), and
- we have \(x,\ xs \in \text{Sub}_{\succsim}(t)\) because of \((x,\ xs) \in \text{Sub}_{\succsim}(t)\).

Since \(\text{PT}\) is the set of all data types, we have \(\{u \mid t \succsim u \in \text{T}^{\text{cls}}_{\text{fun}}, \sigma \notin \text{PT}\} = \emptyset\). Hence we have

\[
\text{Acc}(t) = \text{Sub}_{\succsim}(t) = \{f, e, \text{cons}\ (x,\ xs), (x,\ xs), x, x\}.
\]

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

\[
\text{foldl}\ f\ e\ (\text{cons}\ (x,\ xs)) \to \text{foldl}\ f\ (f\ (e,\ x))\ xs
\]

also satisfies the desired property. Therefore \(R_{\text{len}}\) is accessible.

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

\[
\text{list}(\alpha) \succeq \alpha \times \text{list}(\alpha)
\]

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 \(\beta\)-normal \(\eta\)-long forms is permitted [18], [22]. Moreover, to bring the method close to the existing functional programs, we extend the method on 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 Section 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 \( f \in D \), we provide a new function symbol \( f^\# \), called the marked-symbol of \( f \). For each \( t \models a \overline{r}_n \) with \( a \in D \), we define the marked term \( t^\# \) by \( a^\# \overline{r}_n \). We assume that the marking does not change the type information, that is, \( t \) and \( t^\# \) have the same type. For \( t^\# \), we may write \( t^\#: \sigma \) instead of \( t^{\#\sigma} \) in order to avoid any confusion.

A pair \( \langle l^\#, a^\# \overline{r}_n \rangle \) is said to be an outer static dependency pair in \( R \) if there exists a rule \( l \rightarrow a \overline{r}_n \in R \) satisfying the following conditions:

1. \( a \in D \)
2. \( a \overline{r}_k \notin Acc(l) \) for all \( k \leq n \)

A pair \( \langle l^\#, a^\# \overline{r}_n \rangle \) 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 \( l \rightarrow r \in R \) satisfying the following conditions:

1. \( fn \overline{p} \Rightarrow a \overline{r}_n \in Sub_{bp}(r) \)
2. \( a \in D \)
3. \( a \overline{r}_k \notin Acc(l) \) 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 \( \langle l^\#, r^\# \rangle \) by \( l^\# \rightarrow r^\# \).

**Example 5.2** Consider the TRFP \( \text{R}_{ack} \) displayed in Section 1.

\[
\begin{align*}
\text{prec} \ 0 \ z \ f \rightarrow z \\
\text{prec} \ (\text{succ} \ x) \ z \ f \rightarrow f \ x \ (\text{prec} \ x \ z \ f) \\
\text{iter} \ x \ x \rightarrow \text{prec} \ x \ (f \ (\text{succ} \ 0)) \ (fn \ x' \Rightarrow fn \ z \Rightarrow f \ z) \\
\text{ack} \ x \ x \rightarrow \text{prec} \ x \ succ \ (fn \ x' \Rightarrow fn \ f \Rightarrow \text{iter} \ f)
\end{align*}
\]

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

\[
\begin{align*}
\text{prec}^\# \ (\text{succ} \ x) \ z \ f \rightarrow \text{prec}^\# \ x \ z \ f \\
\text{iter}^\# \ x \ x \rightarrow \text{prec}^\# \ x \ (f \ (\text{succ} \ 0)) \ (fn \ x' \Rightarrow fn \ z \Rightarrow f \ z) \\
\text{ack}^\# \ x \ x \rightarrow \text{prec}^\# \ x \ succ \ (fn \ x' \Rightarrow fn \ f \Rightarrow \text{iter} \ f) \\
\text{ack}^\# \ x \ x \rightarrow \text{iter}^\# \ f
\end{align*}
\]

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 \( u^\# \rightarrow v^\# \), we define the set \( Act(u^\# \rightarrow v^\#) \) of actual outer static dependency pairs as:

\[
s^\# \rightarrow t^\# \in Act(u^\# : \sigma \rightarrow v^\#: \sigma) \text{ iff } s^\# , t^\# \in T^{\# \sigma}, \text{ and there is a type substitution } \xi
\]

such that \( s^\# \equiv u^\# \xi \overline{z}_n \), \( t^\# \equiv v^\# \xi \overline{z}_n \), where the canonical form of \( \xi(\sigma) \) is \( \overline{r}_m \rightarrow \tau \) and each \( z^\#_i \) is a fresh variable.

For any inner static dependency pair \( u^\# \rightarrow v^\# \), we define the set \( Act(u^\# \rightarrow v^\#) \) of actual inner static dependency pairs as:

\[
s^\# \rightarrow t^\# \in Act(u^\# : \sigma \rightarrow v^\#: \sigma) \text{ iff } s^\#, t^\# \in T^{\# \sigma}, \text{ and there is a type substitution } \xi(\sigma) \text{ such that } s^\# \equiv u^\# \xi \overline{z}_n \), \( t^\# \equiv v^\# \xi \overline{z}_m \), where the canonical forms of \( \xi(\sigma') \) and \( \xi(\tau) \) are \( \overline{r}_n \rightarrow \tau' \) and \( \overline{r}_m \rightarrow \tau \), respectively, and each \( z^\#_i : \tau_i \) is a fresh variable.

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 \( u_1^\# \rightarrow v_1^\#, u_2^\# \rightarrow v_2^\#, \ldots \) of static dependency pairs in \( R \) is said to be a static dependency chain in \( R \) if there exist \( s_1^\# \rightarrow t_1^\# \in Act(u_1^\# \rightarrow v_1^\#) \), \( s_2^\# \rightarrow t_2^\# \in Act(u_2^\# \rightarrow v_2^\#), \ldots \), and term substitutions \( \theta_1, \theta_2, \ldots \) such that for any \( i \), \( t_i' \theta_i \overmapsto_{R,m} s_{i+1}' \theta_{i+1} \), and \( s_i, \theta_i, t_i \overmapsto \in T_{SC}\cap T_{SC} \).

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 \( u^\# \rightarrow v^\# \) to \( u'^\# \rightarrow v'^\# \) if \( u^\# \rightarrow v^\# \) 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 \( \text{R}_{ack} \) is shown in Fig. 1. The static dependency graph in \( \text{R}_{ack} \) has only the static recursion component:

\[
\{ \text{prec}^\# (\text{succ} \ x) \ z \ f \rightarrow \text{prec}^\# \ x \ z \ f \}
\]

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 $u^\ell \to v^4 \in C$ 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 Section 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 $C$ is accessible then $\lambda$-calculi with pattern is terminating.

Here $\lambda$-calculi with pattern denotes the rewrite relation $\overline{\pi}$. We note that it may not be terminating in case that $C$ is not accessible. In fact, the rule $\text{foo} (\text{bar} f) \to f (\text{bar} f)$ discussed in Section 1 is not terminating. By using functional abstraction with pattern, this function “foo” can be represented as the term “$\text{fn} \text{bar} f \to f (\text{bar} f)$”. They indicate that the empty TRFP $\emptyset$ may be non-terminating, and so is $\lambda$-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 $\text{hargs}(t) = \{\overline{m}\}$ and $\text{targs}(t) = \{\overline{n}\}$ if $t \equiv (\text{fn} \, p_1 \Rightarrow r_1 | \cdots | p_m \Rightarrow r_m) \, \overline{m}$; otherwise $\text{hargs}(t) = \emptyset$ and $\text{targs}(t) = \text{args}(t)$.

**Lemma 5.11**

(1) If $t \in \mathcal{T}_{-SC}$, $\text{targs}(t) \subseteq \mathcal{T}_{SC}$ and $\text{hargs}(t) \subseteq \mathcal{T}_{SN}$ then there exist $\overline{u} \in \mathcal{T}_{SC}$ and $v$ such that $t \overline{u} \overset{\pi - \overline{m} \rightarrow \overline{n}}{\rightarrow}^{\overline{e}} v \in \mathcal{T}_{\text{fun}} \cap \mathcal{T}_{-SC}$.

(2) If $\overline{t} \in \mathcal{T}_{SC}$ then $(\overline{t}) \in \mathcal{T}_{SC}$ and $\overline{t} \in \mathcal{T}_{SC}$ 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 $\text{fn} \, p \Rightarrow r \in \mathcal{T}_{SC}$ then $r \in \mathcal{T}_{SC}$.

(5) If $r \in \mathcal{T}_{SC}$ and $FV(p) \cap FV(r) = \emptyset$ then $\text{fn} \, p \Rightarrow r \in \mathcal{T}_{SC}$.

(6) $\text{fn} \overline{p_m} \Rightarrow x \in \mathcal{T}_{SC}$ for any $x \in \mathcal{V}$.

**Proof.**

(1) From (SC3), there exists $\overline{u} \in \mathcal{T}_{SC}$ such that $t \overline{u} \in \mathcal{T}_{\text{fun}} \cap \mathcal{T}_{-SC}$. Then the existence of a desired sequence follows from (SC4), (SC1) and (SC5).

(2) Assume that $a \overline{t} \in \mathcal{T}_{SC}$ for some $a \in \mathcal{C} \cup \mathcal{V}$. From $\overline{t} \in \mathcal{T}_{SC}$ and (1), there exist a sequence $a \overline{t} \overset{\pi - \overline{m} \rightarrow \overline{n}}{\rightarrow}^{\overline{e}} a \overline{u} \overset{\overline{r} - \overline{m} \rightarrow \overline{n}}{\rightarrow} \overline{v}$ such that $\forall \, \overline{v} \rightarrow r \in \mathcal{R}, \text{root}(l) \in \mathcal{D}$, 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 $C$.

(6') The assumption $x \notin \mathcal{T}_{SC}$ derives a contradiction with (1). Hence the claim (6) holds for $m = 0$.

(4) Since $\text{fn} \, p \Rightarrow r \neq \emptyset$, $\overline{u} \not\rightarrow \overline{r}$, 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 $\text{fn} \, p \Rightarrow r \notin \mathcal{T}_{SC}$. From (SC1), (1) and $FV(p) \cap FV(r) = \emptyset$, there exist $u, \overline{v} \in \mathcal{T}_{SC}$ such that $\text{fn} \, p \Rightarrow r \overset{u, \overline{v}}{\rightarrow}^{\overline{e}} \overline{r} \Rightarrow \overline{v} \not\rightarrow \overline{r} \notin \mathcal{T}_{SC}$. Then we have $u \overset{\overline{r} - \overline{m} \rightarrow \overline{n}}{\rightarrow}^{\overline{e}} \overline{v}$. From (SC4), we have $u \overline{v} \notin \mathcal{T}_{SC}$. On the other hand, $u \overline{v} \in \mathcal{T}_{SC}$ 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 $(\text{fn} \, \overline{p_m} \Rightarrow x) \equiv \text{fn} \, \overline{q} \Rightarrow x).$

Assume that $\text{fn} \, p \Rightarrow \text{fn} \, \overline{q} \Rightarrow x \in \mathcal{T}_{-SC}$. From $\text{fn} \, \overline{q} \Rightarrow x \in \mathcal{T}_{SN}$ and (1), there exist $\overline{u} \in \mathcal{T}_{SC}$ such

![Diagram](Fig. 1 static dependency graph in $\text{Rack}$)
that \((fnp \Rightarrow fnq \Rightarrow x) \triangleright \triangleright_{\hat{\hat{\hat{n}},\hat{n}}} (fnp \Rightarrow fnq \Rightarrow x) p\theta \triangleright \triangleright_{\hat{\hat{\hat{n}},\hat{n}}} (fnp \Rightarrow fnq \Rightarrow x) p\theta \triangleright \triangleright_{\hat{\hat{\hat{n}},\hat{n}}} \triangleright_{T_{SC}}\). In case of \(x\theta \equiv x\), we have \(fnq \Rightarrow x\theta \equiv fnq \Rightarrow x \in T_{SC}\) from the induction hypothesis. In case of \(x\theta \neq x\), we have \(x\theta \in T_{SC}\) by (SC4) and (3), and hence \(fnq \Rightarrow x\theta \in T_{SC}\) from the variable convention and (5). In both cases, \((fnq \Rightarrow x\theta) w \in T_{SC}\) follows from (SC4) and (SC2). This is a contradiction. □

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

Next, we show the property that \(fnp \Rightarrow t \triangleright \triangleright_{\hat{\hat{\hat{n}}}m}\) is strongly computable whenever \(fnp \Rightarrow t\) and each \(fnp \Rightarrow u_{i}\) 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: \(H := \Box | (fnp \Rightarrow H') \triangleright\).

**Lemma 5.12** For any strongly computable substitution \(\theta\), (i.e. \(\forall x \in \mathcal{V}, x\theta \in T_{SC}\)) such that \((fnp \Rightarrow \triangleright_{\hat{\hat{\hat{n}},\hat{n}}}m)\theta \in T_{SC}\), and \((fnp \Rightarrow \triangleright_{\hat{\hat{\hat{n}}}m}u_{i})\theta \in T_{SC}\) (i.e. \(i = 1, \ldots, n\)), we have \((fnp \Rightarrow t \triangleright \triangleright_{\hat{\hat{\hat{n}}}m})\theta \in T_{SC}\).

**Proof.** We proceed by induction on \(m\). The case \(m = 0\) follows from (SC2). In case of \(m > 0\), we suppose that \((fnp \Rightarrow t \triangleright \triangleright_{\hat{\hat{\hat{n}}}m})\theta \equiv (fnp \Rightarrow fnq \Rightarrow t \triangleright \triangleright_{\hat{\hat{\hat{n}}}m})\theta \equiv (fnp \Rightarrow fnq \Rightarrow t \theta u_{\theta})\theta\). Assume that \((fnp \Rightarrow fnq \Rightarrow t \theta u_{\theta})\theta\) is not strongly computable.

From Lemma 5.11 (4), \((fnq \Rightarrow t\theta\) and each \(fnq \Rightarrow u_{\theta}\) are strongly computable. From the induction hypothesis, \((fnq \Rightarrow t\theta\) u_{\theta})\theta\) is strongly computable. From (SC1) and Lemma 5.11 (1), there exist \(v, w \in T_{SC}\) such that \((fnp \Rightarrow fnq \Rightarrow t\theta u_{\theta}) v \triangleright \triangleright_{\hat{\hat{\hat{n}}}m} (fnp \Rightarrow t\theta u_{\theta}) v \triangleright \triangleright_{\hat{\hat{\hat{n}}}m} \in T_{SC}\). From (SC4) and Lemma 5.11 (3), \(\theta p\) is strongly computable. Thanks to the variable convention, we can define the substitution \(\theta' = \theta p\theta\), which is strongly computable. Then we have \((fnq \Rightarrow t\theta u_{\theta} \theta p\theta \triangleright \triangleright_{\hat{\hat{\hat{n}}}m} t\theta' p w)\). From (SC4), we have \((fnq \Rightarrow t\theta u_{\theta} \theta p\theta \triangleright \triangleright_{\hat{\hat{\hat{n}}}m} (fnq \Rightarrow t\theta' u_{\theta} \theta p \theta \triangleright \triangleright_{\hat{\hat{\hat{n}}}m} \in T_{SC}\). From (SC2), we have \((fnq \Rightarrow t\theta' u_{\theta} \theta p \theta \triangleright \triangleright_{\hat{\hat{\hat{n}}}m} \in T_{SC}\).

Since \((fnp \Rightarrow fnq \Rightarrow t\theta\) and \(v\) are strongly computable, strong computability of \((fnp \Rightarrow fnq \Rightarrow t\theta\) \(v\) follows from (SC2), and hence strong computability of \((fnq \Rightarrow t\theta\) follows from (SC4). As similarity, each \((fnq \Rightarrow u_{\theta}\) is strongly computable. Hence we have \((fnq \Rightarrow t\theta' u_{\theta} \theta p \theta \triangleright \triangleright_{\hat{\hat{\hat{n}}}m} \in T_{SC}\) from the induction hypothesis, It is a contradiction. □

We have shown the basic properties for strong computability. 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 \(t\theta \in T_{SC}\) for any \(t'\theta \in hdec(t)\) then \(t\theta \in T_{SC}\).

**Proof.** We proceed by induction on \(|t|\). Since the case \(t \in \text{hdec}(t)\) is trivial, it suffices to show the case that \(t\) has the form of \(H'[\langle fnp \Rightarrow r_1 | \cdots | l_m \Rightarrow r_m \rangle \pi]\) with \(m > 1\), where \(H'[\] is a single head binding context. Suppose that \(H'[\] \(\equiv H'[\] \(\theta\).

Assume that \(t\theta \notin T_{SC}\). From (SC3), there exist \(\pi \in T_{SC}\) such that \(t\theta \pi \in T_{SC} \cap T_{SC}\). Since \(hdec(t) = \bigcup_{m \geq 1} \text{hdec}(H'[\langle fnp \Rightarrow r_1 \rangle \pi])\), we have \(?i, H'[\langle fnp \Rightarrow r_1 \rangle \pi] \in T_{SC}\) from the induction hypothesis. From (SC2) and (SC1), \(H'[\langle fnp \Rightarrow r_1 \rangle \pi] \pi\) is terminating. Hence it is obvious that \(t\theta \pi \equiv H'[\langle fnp \Rightarrow r_1 \rangle \pi] \pi\) \(\pi\) is also terminating. From (SC4) and (SC5), there exists a sequence \(t\theta \pi \frac{2 \hat{\hat{n}},\hat{n}}{n} w \notin T_{SC}\) with \(\text{root}(w) \neq \pi\). Since \(H'[\] is a single head binding context, there exists \(i\) such that \(H'[\langle fnp \Rightarrow r_1 \rangle \pi] \pi\) \(\frac{2 \hat{\hat{n}},\hat{n}}{n} w \notin T_{SC}\). It is a contradiction with (SC4).

**Lemma 5.14** Let \(fnp \Rightarrow \pi \in T_{SC}\) (\(i = 1, \ldots, n\)). Then \(fnp \Rightarrow \pi \in T_{SC}\) and \(fnp \Rightarrow \pi \in T_{SC}\) for any \(a \in C \cup \mathcal{V}\).

**Proof.** First we show that \(fnp \Rightarrow (\pi) \in T_{SC}\) by induction on \(m\). The case of \(m = 0\) follows from Lemma 5.11 (2). Suppose that \(fnp \Rightarrow (\pi) \equiv fnp \Rightarrow fnq \Rightarrow (\pi)\). From the induction hypothesis, we have \(fnq \Rightarrow (\pi) \in T_{SC}\). Assume that \(fnp \Rightarrow fnq \Rightarrow (\pi) \in T_{SC}\). Then, from (SC1) and Lemma 5.11 (1), there exist \(u, w \in T_{SC}\) such that \((fnp \Rightarrow fnq \Rightarrow (\pi)) u \triangleright \triangleright_{\hat{\hat{\hat{n}}}m} (fnp \Rightarrow fnq \Rightarrow (\pi)) u \triangleright \triangleright_{\hat{\hat{\hat{n}}}m} \in T_{SC}\). From (SC4) and (SC2), we have \(fnq \Rightarrow (\pi) \in T_{SC}\). Since \((fnp \Rightarrow fnq \Rightarrow (\pi)) u \triangleright \triangleright_{\hat{\hat{\hat{n}}}m} fnq \Rightarrow (\pi)\), it follows from (SC2) and (SC4) that \(fnq \Rightarrow (t\theta) \in T_{SC}\) for each \(i\). Hence, from the induction hypothesis, we have \(fnq \Rightarrow (\pi) \in T_{SC}\). It is a contradiction.

Next we show that \(fnp \Rightarrow \pi \in T_{SC}\). The case \(a \in C\) can be proved as similar to the above case. In case \(a \in \mathcal{V}\), we have \(fnp \Rightarrow \pi \in T_{SC}\) 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 \(\text{hd}(t) \theta \in T_{SC}\) and \(t' \theta \subseteq T_{SC}\) for any \(t' \in \text{args}(t)\) then \(t\theta \in T_{SC}\).

**Proof.** We proceed by induction on \(|t|\).

In case that \(t\) has the form of \(fnp \Rightarrow t'\) with \(\text{root}(t') \neq \pi\), the desired property follows from \(t \equiv \text{hd}(t)\).

In case that \(t\) has the form of \(s \triangleright \pi\) with \(n > 0\). Since \(\text{hd}(s) \equiv \text{hd}(t)\) and \(\text{args}(s) \subseteq \text{args}(t)\), we have \(s \theta \in T_{SC}\) from the induction hypothesis. Hence \(t \theta \equiv s \theta \triangleright \pi \in T_{SC}\) follows from (SC2).

In the remaining case, \(t\) has the form of \(fnp \Rightarrow (fnq \Rightarrow r) \triangleright \pi\) with \(n > 0\). Then we have \(\text{hd}(t) \equiv \text{hd}(fnp \Rightarrow fnq \Rightarrow r)\) and \(\text{args}(t) = \text{args}(fnp \Rightarrow fnq \Rightarrow r)\).
Suppose that for any \( fn \) \( \trianglerightrightarrow \) \( u_i \mid i = 1, \ldots, n \). From the induction hypothesis, we have \( fn \) \( \trianglerightrightarrow \) \( fn \) \( q \trianglerightrightarrow \) \( r \in T_{SC} \). Hence, by considering Lemma 5.12 with the empty substitution, we obtain \( fn \) \( \trianglerightrightarrow \) \( (fn \) \( q \trianglerightrightarrow \) \( r \) \( \trianglerightrightarrow \) \( ) \) \( w_n \in T_{SC} \).

**Lemma 5.16**: Suppose that for any \( fn \) \( \trianglerightrightarrow \) \( a \) \( w_n \in Sub_{bp}(t) \) with \( a \) \( \in FV(t) \), there exists \( k \leq n \) such that \( fn \) \( \trianglerightrightarrow \) \( (a \) \( w_n \) \( \theta \in T_{SC} \). If \( t \theta \in T_{SC} \) then there exist \( fn \) \( \trianglerightrightarrow \) \( d \) \( w_n \in Sub_{bp}(t) \) such that \( d \in D \), \( fn \) \( \trianglerightrightarrow \) \( v \theta \in T_{SC} \) for any \( i \), and \( fn \) \( \trianglerightrightarrow \) \( d \) \( v \theta \in T_{SC} \) for any \( k \leq m \).

**Proof**. We proceed by induction on \( |t| \).

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

Suppose that \( hdec(c(t)) = \{ t \} \). From Lemma 5.15, there exists \( u \in \{ hd(c(t)) \} \cup \text{args}(t) \) such that \( u \theta \in T_{SC} \). In case of \( u \equiv \text{hd}(t) \) and \( |hd(c(t))| < |t| \), the desired property follows from the induction hypothesis as similar. Hence it suffices to show the case \( hd(t) = t \) under the assumption \( \theta \in T_{SC} \). Thanks to Lemma 5.14 and \( \theta \in T_{SC} \), we can denote \( t \equiv fn \) \( p \trianglerightrightarrow \) \( a \) \( w_n \) \( with \) \( a \) \( \in D \). Then \( a \in D \cup FV(t) \) and each \( fn \) \( p \trianglerightrightarrow \) \( v \theta \) is strongly computable.

Assume that there exists \( k \leq n \) such that \( fn \) \( p \trianglerightrightarrow \) \( (a \) \( w_n \) \( \theta \in T_{SC} \). From Lemma 5.12, we have \( \theta \in T_{SC} \). It is a contradiction. Moreover \( a \in D \) follows from the assumption for free variables.

**Lemma 5.17**: If \( fn \) \( p \trianglerightrightarrow \) \( d \) \( \in T_{SC} \) and \( d \in D \) then there exist \( w \in T_{SC} \) and a strongly computable substitution \( \theta \) such that \( d \) \( w \theta \in T_{nfun} \cap T_{SC} \cap T_{SC} \).

**Proof**. We proceed by induction on \( m \). The case \( m = 0 \) directly follows from (SC3) with the empty substitution. Suppose that \( fn \) \( p \trianglerightrightarrow \) \( d \) \( \trianglerightrightarrow \) \( fn \) \( \trianglerightrightarrow \) \( d \) \( \trianglerightrightarrow \) \( ) \) \( \in T_{nfun} \cap T_{SC} \cap T_{SC} \).

In case of \( fn \) \( \sim \) \( d \) \( \in T_{SC} \), the desired property follows from the induction hypothesis, because \( fn \) \( \sim \) \( d \) \( \in T_{SC} \) follows from Lemma 5.11 (4).

Suppose that \( fn \) \( \sim \) \( d \) \( \in T_{SC} \). From (SC1) and Lemma 5.11 (1), there exist \( u, w \in T_{SC} \) such that \( fn \) \( p \trianglerightrightarrow \) \( fn \) \( \sim \) \( d \) \( \trianglerightrightarrow \) \( fn \) \( \sim \) \( d \) \( \trianglerightrightarrow \) \( u \) \( w \) \( \sim \) \( n \) \( \in \) \( fn \) \( \sim \) \( \trianglerightrightarrow \) \( v \) \( \theta \) \( w \) \( \sim \) \( n \) \( \in \) \( T_{SC} \). Then \( fn \) \( \sim \) \( d \) \( \trianglerightrightarrow \) \( v \theta \) \( w \) \( \sim \) \( n \) \( \in \) \( T_{SC} \). From (SC4) and (SC2), we have \( fn \) \( \sim \) \( d \) \( \theta \) \( \in T_{SC} \). From \( fn \) \( p \trianglerightrightarrow \) \( \theta \) \( \in T_{SC} \), we have \( fn \) \( p \trianglerightrightarrow \) \( \eta \) \( \in T_{SC} \) for each \( i \). From (SC2) and (SC4), we have \( fn \) \( p \trianglerightrightarrow \) \( \eta \) \( \theta \) \( \trianglerightrightarrow \) \( v \) \( \theta \) \( \in T_{SC} \). Hence \( fn \) \( \sim \) \( d \) \( \theta \) \( \in T_{SC} \). From the induction hypothesis, there exist a strongly computable substitution \( \theta' \) such that \( d \) \( \theta ' \) \( \in T_{nfun} \cap T_{SC} \cap T_{SC} \). Here, strong computability of \( \theta' \) follows from Lemma 5.11 (3). Thanks to the variable convention, the substitution \( \theta = \theta ' \cap \theta '' \) satisfies the desired property.

**Lemma 5.18**: If ATRFP \( R \) is not terminating then \( \{ d \) \( \in T_{nfun} \cap T_{SC} \cap T_{SC} \mid d \in D \} \neq \emptyset \).

**Proof**. From Proposition 2.2, we have \( T_{SC} \cap T_{SC} \neq \emptyset \). From (SC1), we have \( T_{SC} \cap T_{SC} \neq \emptyset \). Let \( t \) be a minimal size term in \( T_{SC} \cap T_{SC} \).

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 \( d \in D \) and \( fn \) \( p \trianglerightrightarrow \) \( d \) \( \in Sub_{bp}(t) \) such that \( fn \) \( \trianglerightrightarrow \) \( d \) \( \in T_{SC} \). 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 \( d \in D \) and \( d \) \( \in T_{nfun} \cap T_{SC} \cap T_{SC} \), there exist \( w' \) \( \in T_{SC} \cap T_{SC} \cap T_{SC} \).

**Proof**. From (SC1) and Lemma 5.11 (1), there exists \( a \) \( \in FV(t) \) such that \( fn \) \( \trianglerightrightarrow \) \( v \theta \) \( \theta \) \( \in T_{SC} \). Hence there exist \( g \in D \) and \( fn \) \( \trianglerightrightarrow \) \( g \) \( \theta \) \( \in Sub_{bp}(t) \) such that \( fn \) \( \trianglerightrightarrow \) \( g \) \( \theta \) \( \in T_{SC} \). Since \( R \) is accessible, it follows from the axiom of accessible function, the variable convention and Lemma 5.11 (5) that \( r \) and \( \theta' \) satisfy the condition of Lemma 5.16. Hence there exist \( g \in D \) and \( fn \) \( \trianglerightrightarrow \) \( g \) \( \theta \) \( \in T_{SC} \). Next, we bridge such counterexamples. Then an infinite static dependency chain emerges.

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 \( t \in T_{nfun} \cap T_{SC} \cap T_{SC} \). 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\(^1\) 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 \((\succeq, \succ)\) of relations on terms is a **reduction pair** if \(\succeq\) and \(\succ\) satisfy the following properties:

- \(\succ\) is closed under term substitutions,
- \(\succeq\) is closed under contexts, type substitutions and term substitutions, and
- \(\succeq \cdot \succ \cdot \succeq\) is well-founded.

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

(i) \(u|_{\pi(\text{root}(u))} \succ_{\text{sub}} v|_{\pi(\text{root}(v))}\) for some \(u^\pi \rightarrow v^\pi \in C\), and

(ii) the following conditions hold for any \(u^\pi \rightarrow v^\pi \in C\):

- \(u|_{\pi(\text{root}(u))} \succeq_{\text{sub}} v|_{\pi(\text{root}(v))}\),
- \((u)_p \not\in V \cup \{\text{fn}\}\) for all \(p \prec \pi(\text{root}(u))\), and
- \(q \not\in \varepsilon \Rightarrow (v)_q \in C \cup \{\text{tp}\}\) for all \(q \prec \pi(\text{root}(v))\).

**Theorem 6.3** Let \(R\) be a finite ATRFP. Then, \(C \in \text{SRC}(R)\) is non-looping if \(C\) satisfies one of the following properties:

- There is a reduction pair \((\succeq, \succ)\) such that \(R \subseteq \succeq\), \(\text{Act}(C) \subseteq \succeq \cdot \succ\), and \(\text{Act}(u^\pi \rightarrow v^\pi) \subseteq \succ\) for some \(u^\pi \rightarrow v^\pi \in C\).

- \(C\) 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 a final example, we will prove the termination of ATRFP \(R_{\text{ack}}\). We take \(\pi(\text{prec}) = 1\). Then the only static recursion component satisfies the subterm criterion in the underlined position below.

\[
\{\text{prec}^2 \ (\text{suc} \ x) \ z \ f \rightarrow \text{prec}^2 \ x \neq z \ f\}
\]

Hence from Theorem 6.3, the static recursion component is non-looping. Therefore the termination of \(R_{\text{ack}}\) follows from Corollary 5.9.

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

\[
\begin{align*}
\{\text{foldl}^2 \ f \ e \ (\text{cons} \ (x, xs)) \rightarrow \text{foldl}^2 \ f \ (f \ (e, x)) \ xs \\
\text{len}^2 \ xs \rightarrow \text{foldl}^2 \ (\text{fn} \ (x, y) \Rightarrow \text{suc} \ x) \ 0 \ xs
\end{align*}
\]

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

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

Hence from Theorem 6.3, the static recursion component is non-looping. Therefore the termination of \(R_{\text{len}}\) 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.

\[
\begin{align*}
\{\text{filter} \ p \ \text{nil} \rightarrow \text{nil} \\
\text{filter} \ p \ (\text{cons} \ (x, xs)) \rightarrow \\
\quad \text{if} \ p \ x \ \text{then} \ \text{cons} \ (x, \text{filter} \ p \ xs) \ \text{else} \ \text{filter} \ p \ xs \\
\text{plist} \ xs \rightarrow \text{filter} \ (\text{fn} \ (x, y) \Rightarrow \text{suc} \ x) \ 0 \ xs
\end{align*}
\]

Here, the expression

\[
\text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_3
\]

is a syntax sugar for the following term:

\[
(\text{fn} \ \text{true} \Rightarrow e_2 \ | \ \text{false} \Rightarrow e_3) \ e_1
\]

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

\[
\begin{align*}
\{\text{filter}^2 \ p \ (\text{cons} \ (x, xs)) \rightarrow \text{filter}^2 \ p \ xs \\
\text{plist}^2 \ xs \rightarrow \text{filter}^2 \ (\text{fn} \ 0 \Rightarrow \text{false} \ | \ \text{suc} \ x \Rightarrow \text{true}) \ xs
\end{align*}
\]

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

\[
\{\text{filter}^2 \ p \ (\text{cons} \ (x, xs)) \rightarrow \text{filter}^2 \ p \ xs\}
\]

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

In Section 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:

\[
\begin{align*}
(S f^{\alpha \to \beta \to \gamma})^{\gamma} & \to f \,(g \, x) \\
(K \, x^{\alpha})^{\beta} & \to x \\
(I \, x^{\alpha})^{\alpha} & \to x \\
(B \, x^{\alpha \to \beta \gamma} \, y^{\gamma \to \alpha} \, z^{\gamma})^{\beta} & \to x \,(y \, z) \\
(B' \, x^{\alpha \to \beta \gamma} \, y^{\beta \to \gamma} \, z^{\alpha})^{\gamma} & \to y \,(x \, z) \\
(C \, x^{\alpha \to \beta \gamma} \, y^{\beta} \, z^{\alpha})^{\gamma} & \to x \, z \, y \\
(J \, x^{\alpha \to \beta \gamma} \, y^{\alpha \beta} \, z^{\beta \alpha})^{\beta} & \to x \, y \,(x \, w \, z) \\
(W \, x^{\alpha \to \alpha \beta} \, y^{\alpha})^{\beta} & \to x \, y \, y
\end{align*}
\]

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


