Enhancing Dependency Pair Method using Strong Computability in Simply-Typed Term Rewriting

KUSAKARI Keiichirou and SAKAI Masahiko

Graduate School of Information Science, Nagoya University
{kusakari,sakai}@is.nagoya-u.ac.jp

Abstract. We enhance the dependency pair method in order to prove termination using recursive structure analysis in simply-typed term rewriting systems, which is one of the computational models of functional programs. The primary advantage of our method is that one can exclude higher-order variables which are difficult to analyze theoretically, from recursive structure analysis. The key idea of our method is to analyze recursive structure from the viewpoint of strong computability. This property was introduced for proving termination in typed λ-calculus, and is a stronger condition than the property of termination. The difficulty in incorporating this concept into recursive structure analysis is that because it is defined inductively over type structure, it is not closed under the subterm relation. This breaks the correspondence between strong computability and recursive structure. In order to guarantee the correspondence, we propose plain function-passing as a restriction, which is satisfied by many non-artificial functional programs.

Keyword. Termination, Simply-Typed Term Rewriting System, Plain Function-Passing, Dependency Pair, Strong Computability

1 Introduction

One of the important features of functional programming languages is higher-order abstraction achieved using higher-order functions which may take other functions as arguments and may return functions as results. For example, the left-folding function $foldl$ which is a built-in function of the functional programming language SML [38], is defined as follows:

```
fun foldl f y nil = y
  | foldl f y (x::xs) = foldl f (f (x,y)) xs;
```

This function has the type `('a * 'b -> 'b) -> 'b -> 'a list -> 'b`, and hence the first argument `f` has the functional type `'a * 'b -> 'b`, where `'a` and `'b` are type variables. The constructor `::` takes as argument a tuple in `'a * 'a list`.

One of the computational models that provides operational semantics for functional programs and directly handles higher-order functions is simply-typed term-rewriting systems (STRSs) [20]. For example, the usual addition and multiplication over natural numbers are represented as the following STRSs:

$$\begin{array}{lcl} R_+ &=& \{+[(x,0)] \to x,\ \ +[(x,s[y])] \to s[+[(x,y)]]\} \\ R_\times &=& R_+ \cup \{\times[(x,0)] \to 0,\ \ \times[(x,s[y])] \to +[(\times[(x,y)],x)]\} \end{array}$$

Moreover the above $foldl$ function with the simple type $(N \times N \to N) \to N \to L \to N$ is represented as the following STRS:

$$R_{foldl} = \left\{ \begin{array}{rcl} foldl[f,y,nil] &\to& y \\ foldl[f,y,cons[(x,xs)]] &\to& foldl[f,f[(x,y)],xs] \end{array} \right.$$

For the expression "`foldl f y nil`", the term $foldl[f,y,nil]$ in the STRS R_{foldl} is constructed using the the operator "$foldl$" and the argument list "$[f,y,nil]$". A tuple is represented by the special constructor tp, and a tuple $tp[x,xs]$ is usually denoted by a syntactic sugar (x,xs). Note that simple types do not include type variables, hence STRSs do not handle polymorphic functions directly. Moreover STRSs do not use the λ-abstraction, *i.e.*, do not represent anonymous functions. On the other hand, STRSs have a high compatibility with first-order TRSs.

Using the function $foldl$, the sum function, which calculates the total sum for an input list, and the $prod$ function, which calculates the total product, can be represented by the following STRSs:

$$\begin{array}{lcl} R_{sum} & = & R_{+} \cup R_{foldl} \cup \{sum \to foldl[+,0]\} \\ R_{prod} & = & R_{\times} \cup R_{foldl} \cup \{prod \to foldl[\times, s[0]]\} \end{array}$$

Higher-order abstraction can be achieved by using higher-order functions in this way, thereby increasing the expressive power of the language and the reusability of programs.

We can now show the termination of the STRSs R_{sum} and R_{prod}. Intuitively, we believe that they do terminate, but this is quite difficult to verify because their reduction may be affected by unanticipated behaviors of functions held in higher-order variables. Let's consider the STRS R_{foldl}. For most programmers, the rewriting system R_{foldl} that defines $foldl$ terminates. The reasons are that the definitions are assumed to be modular with no recursion through modules, that function f is assumed to be terminating, and that the rightmost argument in the definition of $foldl$, namely $cons[(x, xs)]$ is replaced by a smaller argument, namely xs. However, this reasoning does not hold in general. In fact, if we add the function foo to R_{foldl} as shown:

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

then the second argument of $foldl$ is called through mutual recursion with foo and evaluation does not terminate:

$$foo[(x, y)] \to foldl[foo, y, cons[(x, nil)]] \to foldl[foo, \underline{foo[(x, y)]}, nil]$$

Arts and Giesl proposed a method for proving termination of TRSs called the dependency pair method [3]. It analyzes the recursive structure through function-call dependency relationships in order to prove termination, and it was extended to STRSs [20], and HRSs [29]. Analyzing STRS R_{foo} with the method in [20] yields the following recursive structure.

$$\begin{array}{l}
\{\ foldl^\sharp[f, y, cons[(x, xs)]] \to foldl^\sharp[f, f[(x, y)], xs]\ \} \\
\left\{ \begin{array}{l} foldl^\sharp[f, y, cons[(x, xs)]] \to f[(x, y)] \\ foo^\sharp[(x, y)] \to foldl^\sharp[foo, y, cons[(x, nil)]] \end{array} \right\} \\
\left\{ \begin{array}{l} foldl^\sharp[f, y, cons[(x, xs)]] \to foldl^\sharp[f, f[(x, y)], xs] \\ foldl^\sharp[f, y, cons[(x, xs)]] \to f[(x, y)] \\ foo^\sharp[(x, y)] \to foldl^\sharp[foo, y, cons[(x, nil)]] \end{array} \right\}
\end{array}$$

Each set is called a recursion component. Dependency pair method proves termination by showing non-loopingness in each recursion component. In order to show non-loopingness, we use the notion of (semi-)reduction pairs [19, 20], which is an improvement on weak-reduction order [3], and the notion of the subterm criterion [16], which will be extended in subsection 3.2.

In this paper, we propose a new dependency pair method, called the SC-dependency pair method. The key idea of the method is to analyze recursive structures from the viewpoint of strong computability[1]. This property was introduced for proving termination in typed λ-calculus, and is a stronger condition than the property of termination [13, 33]. Intuitively, the notion of strong computability corresponds to terminating functions. More precisely, for an arbitrary relevant input, computations of the function always succeed. One of the most important pioneering results for proving termination in higher-order rewriting systems, is the higher-order path ordering by Jouannaud and Rubio [17], which was reformulated in [27]. Kusakari showed that it can also be applied to STRSs, and studied an application to the argument filtering method [21], which generates reduction pairs, and was introduced with first-order TRSs by Arts and Giesl [3]. We also incorporated the idea of strong computability into the dependency pair method [32], however the result only proves the sufficient completeness with respect to the call-by-value strategy.

Intuitively, the dependency pair method in [20] analyzes a dynamic recursive structure based on function-call dependency, while the SC-dependency pair method analyzes a static recursive structure based on definition dependency. Hence the SC-dependency pair method does not require the analysis of higher-order variables (such as that of $foldl^\sharp[f, y, cons[(x, xs)]] \to f[(x, y)]$), and the number of recursion

[1]Recently, Blanqui has independently proposed an idea on incorporating the notion of strong computability into dependency pair methods for proving termination of CRS-like systems [5].

components is significantly reduced. For example, analyzing STRS R_{foo} with the SC-dependency pair method yields only the following two recursion components:

$$\{foldl^\sharp[f, y, cons[(x, xs)]] \to foldl^\sharp[f, f[(x, y)], xs]\}$$
$$\{foo^\sharp[(x, y)] \to foo^\sharp[z]\}$$

As a result, the SC-dependency pair method is extremely powerful and efficient. Indeed, our method proves the termination of simply-typed combinatory logic from two facts which can be verified easily: there are no static recursive structure, and each higher-order variable occurs in an argument on the left-hand side (cf. Example 4.25).

The most important and difficult component in the design of the SC-dependency pair method is the compatibility between the structures of "terms" and "types", because strong computability is defined over type structure (cf. Definition 4.1), whereas SC-dependency pairs are defined over term structure (cf. Definition 4.11). From a technical viewpoint, it is the fact that strong computability is not closed under the subterm relation that makes its design difficult. Indeed the SC-dependency pair method cannot be applied to general STRSs (cf. Example 4.4), so we want to define a class that is sufficiently expressive and a desirable property. One such restriction is the notion of plain function-passing (cf. Definition 4.7). Roughly speaking, plain function-passing means that every higher-order variable occurs in an argument position on the left-hand side. Every terminating STRS in this paper is plain function-passing, and many non-artificial functional programs are written as plain function-passing STRSs. This fact demonstrates the versatility of our SC-dependency pair method.

The remainder of this paper is organized as follows. The next section provides preliminaries required later in the paper. In Section 3, in order to clarify various concepts, we provide an abstract framework that unifies several dependency pair methods, and reformulate the dependency pair method in [20]. In Section 4, we give a new dependency pair method, called the SC-dependency pair method, whose soundness is guaranteed by the concept of strong computability. Concluding remarks are presented in Section 6.

2 Preliminaries

In this section, we introduce the basic notations for simply-typed term rewriting systems [20]. We assume that the reader is familiar with notions of term rewriting systems [34].

2.1 Abstract Reduction System

An *abstract reduction system* (ARS) is a pair $\langle A, \to \rangle$ where A is a set and $\to$ is a binary relation on A. The transitive-reflexive closure and the transitive closure of a binary relation $\to$ are denoted by $\xrightarrow{*}$ and $\xrightarrow{+}$, respectively.

An element $a \in A$ is said to be *terminating* or *strongly normalizing* in an ARS $R = \langle A, \to \rangle$, denoted by $SN(R, a)$, if every reduction sequence staring from a is finite. Formally, the predicate SN is defined as $SN(R, a) \iff a \in A_{SN}$, where A_{SN} is the least set such that $a \in A_{SN} \iff \forall b(a \to b \Rightarrow b \in A_{SN})$. An ARS $R = \langle A, \to \rangle$ is said to be *terminating* or *strongly normalizing*, denoted by $SN(R)$, if $SN(R, a)$ holds for any $a \in A$.

2.2 Untyped Term Rewriting Systems

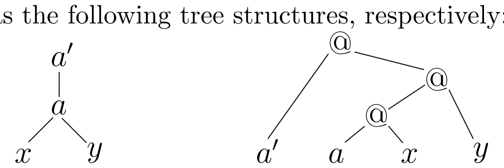
The set $T(\Sigma, \mathcal{V})$ of *(untyped) terms* generated from a set Σ of function symbols and a set $\mathcal{V}$ of variables with $\Sigma \cap \mathcal{V} = \emptyset$ is the smallest set such that $a[t_1, \ldots, t_n] \in T(\Sigma, \mathcal{V})$ whenever $a \in \Sigma \cup \mathcal{V}$ and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

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

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

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

A term u is said to be a *subterm* (resp. an *extended subterm*) of t, denoted by $t \geq_{sub} u$ (resp. $t \geq_{esub} u$), if there exists a leaf-context (resp. context) $C[\,]$ such that $t \equiv C[u]$. We also define $>_{sub} = \geq_{sub} \setminus \equiv$ and $>_{esub} = \geq_{esub} \setminus \equiv$. We denote all subterms (resp. extended subterms) of t by $Sub(t)$ (resp. $ESub(t)$). The subterm of t at position p is denoted by $t|_p$. Note that by using currying technique [18] the term $a'[a[x,y]]$ is written as $@(a',@(@(a,x),y))$. The term $a'[a[x,y]]$ and the curried term $@(a',@(@(a,x),y))$ has the following tree structures, respectively:



The subterms correspond with the subtrees in the left tree, and the extended subterms correspond with the subtrees in the right tree. Actually, $Sub(a'[a[x,y]]) = \{a'[a[x,y]], a[x,y], x, y\}$ and $ESub(a[x,y]) = \{a'[\,], a[\,], a[x]\} \cup Sub(a'[a[x,y]])$.

A *rule* is a pair (l,r) of terms, denoted by $l \to r$, such that $root(l) \in \Sigma$ and $Var(l) \supseteq Var(r)$. The *reduction relation* $\underset{R}{\to}$ of a set R of rules is defined by $s \underset{R}{\to} t$ iff $s \equiv C[l\theta]$ and $t \equiv C[r\theta]$ for some rule $l \to r \in R$, context $C[\,]$ and substitution θ. We often omit the subscript R whenever no confusion arises. An *untyped term rewriting system* (UTRS) is an abstract reduction system $\langle \mathcal{T}(\Sigma,\mathcal{V}), \underset{R}{\to} \rangle$. We often denote an UTRS $\langle \mathcal{T}(\Sigma,\mathcal{V}), \underset{R}{\to} \rangle$ by R.

2.3 Simply-Typed Term Rewriting Systems

A set of *basic types* is denoted by $\mathcal{B}$. The set $\mathcal{S}$ of *simple types* (with product types) is generated from $\mathcal{B}$ by type constructors $\to$ and $\times$, that is, $\mathcal{S} ::= \mathcal{B} \mid (\mathcal{S}_1 \to \mathcal{S}_2) \mid (\mathcal{S}_1 \times \cdots \times \mathcal{S}_n)$. To minimize the number of parentheses, we assume that $\to$ is right-associative and $\to$ has lower precedence than $\times$. We also assume that Σ contains a special constructor tp, called a *tuple*. We write $(t_1,\ldots,t_n)$ instead of $tp[t_1,\ldots,t_n]$. A *typing function* τ is a function from $\mathcal{V} \cup (\Sigma \setminus \{tp\})$ to $\mathcal{S}$. We assume that for any $\alpha \in \mathcal{S}$ there exists a variable $x \in \mathcal{V}$ such that $\tau(x) = \alpha$. Each typing function τ is naturally extended to terms as follows: for any $t \equiv a[t_1,\ldots,t_n] \in T(\Sigma,\mathcal{V})$, if $\tau(t_i) = \alpha_i$ $(i = 1,\ldots,n)$ and either $\tau(a) = \alpha_1 \to \cdots \to \alpha_n \to \alpha$ or $a = tp \wedge \alpha = \alpha_1 \times \cdots \times \alpha_n$, then $\tau(t) = \alpha$. We remark that the symbol tp can be polymorphically interpreted, that is, we can interpret that $\tau(tp) = S_1 \to \cdots \to S_n \to S_1 \times \cdots \times S_n$ for any $S_1,\ldots,S_n \in \mathcal{S}$. A term $t \in T(\Sigma,\mathcal{V})$ is said to be *simply typed* if t has a simple-type, that is, $\tau(t)$ is defined. We denote the set of all simply-typed terms by $T_\tau(\Sigma,\mathcal{V})$. A *product type* is a simple type of the form $\alpha_1 \times \cdots \times \alpha_n$, and a *functional type* or a *higher-order type* is a simple type of the form $\alpha \to \beta$. We denote the set of functional types by $\mathcal{S}_{fun}$, and the set of functional typed terms by $\mathcal{T}_{fun}(\Sigma,\mathcal{V})$. We use $\mathcal{V}_{fun}$ to stand for the set of functionally typed variables (higher-order variables). Now we restrict substitutions to type preserving substitutions. We also index the hole $\square_\alpha$ with every simple type α, and assume that $\tau(t) = \alpha$ whenever we denote $C[t]$ for each context $C[\,]$ with a hole $\square_\alpha$.

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

3 Abstract Frameworks for Dependency Pair Methods and the SN-dependency Pair Method

In order to distinguish between the dependency pair method for STRSs [20] and the new dependency pair method based on strong computability introduced in the next section, we distinguish the two methods as the SN-dependency pair method and the SC-dependency pair method, respectively.

Before introducing the SC-dependency pair method, we provide an abstract framework for dependency pair methods. We introduce the notion of recursion components, and show that if all recursion components of an STRS R are non-looping then R is terminating. We also introduce the notions of (semi-)reduction pairs and the subterm criterion, which prove that recursion components are non-looping. In addition, we explain how dependency pair methods work, by reformulating the SN-dependency pair method. In the next section, we will present the SC-dependency pair method using this abstract framework.

3.1 Dependency Pair and Recursion Component

The most basic notion in all dependency pair methods is that of the dependency pair itself, which expresses some kind of function dependency. Firstly, we recall the SN-dependency pair method in [20]. Note that we naturally extend the definition so that it can handle rules of function type, because the STRSs defined in [20] have the restriction that all rules are of basic types. Before stating the definition, we introduce several concepts.

Definition 3.1 Let R be an STRS and $l \to r \in R$ such that $\tau(l) = \alpha_1 \to \cdots \to \alpha_n \to \alpha$ and $\alpha \notin \mathcal{S}_{\mathit{fun}}$. The set $(l \to r)^{ex}$ of the *expansion forms* of a rule $l \to r$ is defined as $\{l \to r, l[z_1] \to r[z_1], \ldots, l[z_1, \ldots, z_n] \to r[z_1, \ldots, z_n]\}$, where $z_1, \ldots, z_n$ are fresh variables with $\forall i.\tau(z_i) = \alpha_i$. We also define $R^{ex} = \bigcup_{l \to r \in R} (l \to r)^{ex}$.

Definition 3.2 All root symbols of the left-hand sides of rules in an STRS R, denoted by $\mathcal{D}_R$, are called *defined*, whereas all other function symbols, denoted by $\mathcal{C}_R$, are called *constructors*. For each $f \in \mathcal{D}_R$, we define a new function symbol $f^\sharp$, called the *marked-symbol* of f. For each $t \equiv a[t_1, \ldots, t_n]$, we define the *marked term* $t^\sharp$ by $a^\sharp[t_1, \ldots, t_n]$ if $a \in \mathcal{D}_R$; otherwise $t^\sharp \equiv t$.

Definition 3.3 Let R be an STRS. A pair $u^\sharp \to v^\sharp$ of marked terms is an *SN-dependency pair* of R if there exists $u \to r \in R^{ex}$ such that $v \in \mathit{Sub}(r)$, $\mathit{root}(v) \in \mathcal{D}_R \cup \mathcal{V}_{\mathit{fun}}$ and $v \notin \mathit{ESub}(u')$ for all $u' \in \mathit{args}(u)$[2]. We use $DP_{SN}(R)$ to denote the set of all SN-dependency pairs of R.

We recall that for $t \equiv a'[a[x, y]]$ the term $a[x]$ is an extended subterm ($a[x] \in \mathit{ESub}(t)$) but not a subterm ($a[x] \notin \mathit{Sub}(t)$).

Example 3.4 Let $x, y, i, f, g \in \mathcal{V}$, and $s, \mathit{cons}, \mathit{foldl}, \mathit{foo}, \mathit{apply} \in \Sigma$ such that $\tau(x) = \tau(y) = \tau(i) = N$, $\tau(f) = N \times N \to N$, $\tau(g) = \tau(s) = N \to N$, $\tau(\mathit{cons}) = N \times L \to L$, $\tau(\mathit{foldl}) = (N \times N \to N) \to N \to L \to N$, $\tau(\mathit{foo}) = N \to N \times N \to N$, and $\tau(\mathit{apply}) = (N \to N) \to N \to N$. We consider the following STRS R'_{foo}:

$$R'_{\mathit{foo}} = \left\{ \begin{array}{rcl} \mathit{foldl}[f, y, \mathit{nil}] & \to & y \\ \mathit{foldl}[f, y, \mathit{cons}[(x, xs)]] & \to & \mathit{foldl}[f, f[(x, y)], xs] \\ \mathit{foo}[s[i], (x, y)] & \to & \mathit{foldl}[\mathit{foo}[i], y, \mathit{cons}[(x, \mathit{nil})]] \\ \mathit{apply}[g] & \to & g \end{array} \right.$$

The definition of foo in R'_{foo} is a little different from the definition of R_{foo} displayed in the introduction. The change gives R'_{foo} the property of termination. $DP_{SN}(R'_{\mathit{foo}})$ is the following:

$$DP_{SN}(R'_{\mathit{foo}}) = \left\{ \begin{array}{rcl} \mathit{foldl}^\sharp[f, y, \mathit{cons}[(x, xs)]] & \to & \mathit{foldl}^\sharp[f, f[(x, y)], xs] \\ \mathit{foldl}^\sharp[f, y, \mathit{cons}[(x, xs)]] & \to & f[(x, y)] \\ \mathit{foo}^\sharp[s[i], (x, y)] & \to & \mathit{foldl}^\sharp[\mathit{foo}[i], y, \mathit{cons}[(x, \mathit{nil})]] \\ \mathit{foo}^\sharp[s[i], (x, y)] & \to & \mathit{foo}^\sharp[i] \\ \mathit{apply}^\sharp[g, z] & \to & g[z] \end{array} \right.$$

[2] We slightly modify this condition from the original one in [20]: $\mathit{root}(v) \in \mathcal{D}_R$ or $\mathit{root}(v) \in \mathcal{V} \wedge \mathit{args}(v) \neq [\,]$. Such an idea of excluding dependency pairs is due to Dershowitz [7].

Note that $foldl^\sharp[f, y, cons[(x, xs)]] \to f$ is not an SN-dependency pair because f is an extended subterm of the first argument f of $foldl[f, y, cons[(x, xs)]]$.

We now present an abstract framework for dependency chains.

Definition 3.5 Let P be a set of pairs of marked terms, $\gg$ be a binary relation on terms, and T_l and T_r be sets of terms. A sequence $u_0^\sharp \to v_0^\sharp, u_1^\sharp \to v_1^\sharp, \cdots$ of pairs in P is said to be a *$\langle P, \gg\rangle$-chain on* (T_l, T_r) if there exists $\theta_0, \theta_1, \ldots$ such that $u_i\theta_i \in T_l$, $v_i\theta_i \in T_r$ and $(v_i\theta_i)^\sharp \gg^* (u_{i+1}\theta_{i+1})^\sharp$ for any i.

In this abstract framework, the notion of dependency chains is reformulated as follows:

Definition 3.6 A *$DP_{SN}(R)$-chain* of an STRS R is defined as $\langle DP_{SN}(R), \underset{R}{\to}\rangle$-chain on $(\mathcal{T}_{SN}^{args}(R), \mathcal{T}_{SN}^{args}(R))$.

The following theorem represents the essence of dependency pair methods.

Theorem 3.7 Let R be an ARS $\langle T_\tau(\Sigma, \mathcal{V}), \gg\rangle$, P be a set of pairs of marked terms, and T_l and T_r be sets of terms. Suppose that there exists a set T of terms satisfying the following conditions:

(1) $T_r \cap T \neq \emptyset$, and

(2) For any $t \in T_r \cap T$, there exist $u^\sharp \to v^\sharp \in P$ and θ such that $t^\sharp \gg^* (u\theta)^\sharp$, $u\theta \in T_l$ and $v\theta \in T_r \cap T$.

Then there exists an infinite $\langle P, \gg\rangle$-chain on (T_l, T_r).

Proof. Thanks to the property (1), let $t \in T_r \cap T$. From the property (2), there exist $u_0^\sharp \to v_0^\sharp \in P$ and θ_0 such that $t^\sharp \gg^* (u_0\theta)^\sharp$, $u_0\theta \in T_l$ and $v_0\theta \in T_r \cap T$. From the property (2) again, there exist $u_1^\sharp \to v_1^\sharp \in P$ and θ_1 such that $(v_0\theta)^\sharp \gg^* (u_1\theta)^\sharp$, $u_1\theta \in T_l$ and $v_1\theta \in T_r \cap T$. By applying this procedure repeatedly, we have an infinite sequence $u_0^\sharp \to v_0^\sharp, u_1^\sharp \to v_1^\sharp, u_2^\sharp \to v_2^\sharp \cdots$, which is an infinite $\langle P, \gg\rangle$-chain on (T_l, T_r). $\square$

This theorem rebuilds again the fundamental theorem of the SN-dependency pair method stated in [20].

Theorem 3.8 An STRS R is not terminating if and only if there exists an infinite $DP_{SN}(R)$-chain.

Proof. ($\Leftarrow$) Trivial. ($\Rightarrow$) It suffices to show the properties (1) and (2) in Theorem 3.7 with $T = \mathcal{T}_\tau(\Sigma, \mathcal{V}) \setminus \mathcal{T}_{SN}(R)$ and $T_l = T_r = \mathcal{T}_{SN}^{args}(R)$. (1) Any minimal size term in T is in $\mathcal{T}_{SN}^{args}(R)$. (2) This property correspond to Lemma 5.4 in [20]. Although this lemma was given on untyped systems, the proof can still be used because STRSs have the subject property ($s \underset{R}{\to} t \Rightarrow \tau(s) = \tau(t)$). $\square$

To paraphrase this proposition, the non-termination property and the existence of an infinite function-call sequence interpreted by the SN-dependency pair method are logically equivalent. The SN-dependency pair method proves termination by proving that an infinite function-call sequence does not exist.

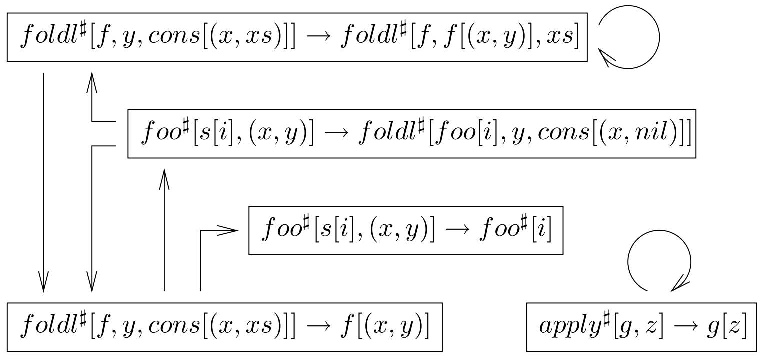
Note that the ($\Leftarrow$)-part does not hold in the SC-dependency pair method introduced in the next section. Hence the SC-dependency pair method has a theoretical limitation (as described in the concluding remarks).

Finally, we present the concepts of dependency graphs and recursion components in an abstract framework, and reformulate them in terms of the SN-dependency pair method.

Definition 3.9 Let P be a set of pairs of marked terms, $\gg$ be a binary relation on terms, and T_l and T_r be sets of terms. A *$\langle P, \gg\rangle$-dependency graph on* (T_l, T_r) is a directed graph, in which nodes are P and there exists an arc from $u_0^\sharp \to v_0^\sharp \in P$ to $u_1^\sharp \to v_1^\sharp \in P$ if $u_0^\sharp \to v_0^\sharp, u_1^\sharp \to v_1^\sharp$ is a $\langle P, \gg\rangle$-chain on (T_l, T_r). A *$\langle P, \gg\rangle$-recursion component on* (T_l, T_r) is the set of nodes in a strongly connected subgraph of $\langle P, \gg\rangle$-dependency graph on (T_l, T_r). A $\langle P, \gg\rangle$-recursion component C on (T_l, T_r) is said to be *non-looping* if there exists no infinite $\langle C, \gg\rangle$-chain on (T_l, T_r) in which every $u^\sharp \to v^\sharp \in C$ occurs infinitely many times.

Definition 3.10 Let R be an STRS. The *$DP_{SN}(R)$-graph* is defined as $\langle DP_{SN}(R), \underset{R}{\to}\rangle$-graph on $(\mathcal{T}_{SN}^{args}(R), \mathcal{T}_{SN}^{args}(R))$. We denote by $RC_{SN}(R)$ the set of $\langle DP_{SN}(R), \to_R\rangle$-recursion components on $(\mathcal{T}_{SN}^{args}(R), \mathcal{T}_{SN}^{args}(R))$.

Example 3.11 The $DP_{SN}(R'_{foo})$-graph is the following:



Hence the recursion components $RC_{SN}(R'_{foo})$ is constructed by the following four components:

$$\left\{\begin{array}{l}
\left\{\ foldl^\sharp[f, y, cons[(x, xs)]] \ \to\ foldl^\sharp[f, f[(x, y)], xs]\ \right\} \\
\left\{\begin{array}{rcl} foldl^\sharp[f, y, cons[(x, xs)]] & \to & f[(x, y)] \\ foo^\sharp[s[i], (x, y)] & \to & foldl^\sharp[foo[i], y, cons[(x, nil)]] \end{array}\right\} \\
\left\{\begin{array}{rcl} foldl^\sharp[f, y, cons[(x, xs)]] & \to & foldl^\sharp[f, f[(x, y)], xs] \\ foldl^\sharp[f, y, cons[(x, xs)]] & \to & f[(x, y)] \\ foo^\sharp[s[i], (x, y)] & \to & foldl^\sharp[foo[i], y, cons[(x, nil)]] \end{array}\right\} \\
\left\{\ apply^\sharp[g, z] \ \to\ g[z]\ \right\}
\end{array}\right.$$

Theorem 3.12 Let P be a finite set of pairs of marked terms, $\gg$ be a binary relation on terms, and T_l and T_r be sets of terms. Then there exists no infinite $\langle P, \gg\rangle$-chain on (T_l, T_r) if and only if any $\langle P, \gg\rangle$-recursion component on (T_l, T_r) is non-looping.

Proof. It is obvious because P is finite. □

Note that the abstract framework above is merely a reformulation of the dependency pair method introduced in [3].

By applying Theorem 3.8 to this theorem, we obtain the following theorem.

Theorem 3.13 Let R be an STRS such that $DP_{SN}(R)$ is finite. Then R is terminating if and only if all recursion components in $RC_{SN}(R)$ are non-looping.

As may be seen from this theorem, the SN-dependency pair method proves termination by proving that all recursion-calls are non-looping.

3.2 Proving that Recursion Components are Non-looping

In this subsection, we introduce the notion of (semi-)reduction pairs and the subterm criterion, which proves that recursion components do not loop. Lastly, we reformulate the SN-dependency pair method in [20], and show that (semi-)reduction pairs and the subterm criterion play important roles in dependency pair methods.

Firstly we introduce the notion of (semi-)reduction pairs described in [19], which is an improvement on weak-reduction order [3].

Definition 3.14 Let $\gtrsim$ be a quasi-order and $>$ be a strict order. The pair $(\gtrsim, >)$ is said to be a *semi-reduction pair* if $\gtrsim$ is closed under leaf-contexts and substitutions, $>$ is well-founded and closed under substitution, and either $\gtrsim\cdot> \subseteq >$ or $>\cdot\gtrsim \subseteq >$ holds. A semi-reduction pair $(\gtrsim, >)$ is said to be a *reduction pair* if $\gtrsim$ is closed under contexts. For a set C of pairs of marked terms, we say that $(\gtrsim, >)$ satisfies the *marked condition* in C if $v\theta \gtrsim (v\theta)^\sharp$ for any θ and $u^\sharp \to v^\sharp \in C$ such that $root(v) \in \mathcal{V}$.

The argument filtering method, which generates a reduction pair from a given reduction order, was introduced in first-order TRSs by Arts and Giesl [3] in order to design reduction pairs. The method was extended to STRSs [20]. Although the path order based on strong computability in [21] generates reduction pairs, the path order based on the simplification order in [20] does not generate reduction pairs and only generates semi-reduction pairs.

Theorem 3.15 Let R be an STRS and C be a set of pairs of marked terms such that $root(u) \notin \mathcal{V}$ for any $u^\sharp \to v^\sharp \in C$. If there exists a reduction pair (resp. semi-reduction pair) $(\gtrsim, >)$ satisfying the following conditions then C is non-looping.

(i) $R \subseteq \gtrsim$ (resp. $R^{ex} \subseteq \gtrsim$),

(ii) $C \subseteq \gtrsim$ and $C \cap > \neq \emptyset$, and

(iii) it satisfies the marked condition in C.

Proof. Assume that there exists an infinite $\langle C, \underset{R}{\to} \rangle$-chain $u_0^\sharp \to v_0^\sharp, u_1^\sharp \to v_1^\sharp, \cdots$, in which every $u^\sharp \to v^\sharp \in C$ occurs infinitely many times, and let $\theta_0, \theta_1, \ldots$ be substitutions such that $(v_i\theta_i)^\sharp \overset{*}{\underset{R}{\to}} (u_{i+1}\theta_{i+1})^\sharp$ for each i. In both cases of a reduction pair and a semi-reduction pair, $(v_i\theta_i)^\sharp \gtrsim (u_{i+1}\theta_{i+1})^\sharp$ follows from the condition (i). Since $\forall i. root(u_i) \notin \mathcal{V}$, $C \subseteq \gtrsim$ and $\gtrsim$ is closed under substitution, we have $(u_i\theta_i)^\sharp \equiv u_i^\sharp\theta_i \gtrsim v_i^\sharp\theta_i$ for each i. From the condition (iii), we have $v_i^\sharp\theta_i \gtrsim (v_i\theta_i)^\sharp$. Hence there exists an infinite decreasing sequence $(u_0\theta_0)^\sharp \gtrsim (v_0\theta_0)^\sharp \gtrsim (u_1\theta_1)^\sharp \gtrsim (v_1\theta_1)^\sharp \gtrsim (u_2\theta_2)^\sharp \gtrsim (v_2\theta_2)^\sharp \gtrsim \cdots$. In a similar way, we have $(u_i\theta_i)^\sharp > (v_i\theta_i)^\sharp$ for infinitely many i. This contradicts the well-foundedness of $>$. $\square$

We next extend the subterm criterion proposed for first-order TRSs [16].

Definition 3.16 Let R be an STRS and C be a set of pairs of marked terms. We say that C satisfies the *subterm criterion* if $root(u), root(v) \notin \mathcal{V}$ for any $u^\sharp \to v^\sharp \in C$, and there exists a function π from $\mathcal{D}_R$ to non-empty sequences of positive integers such that

(α) $u|_{\pi(root(u))} >_{esub} v|_{\pi(root(v))}$ for some $u^\sharp \to v^\sharp \in C$, and

(β) the following conditions hold for any $u^\sharp \to v^\sharp \in C$:

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

Note that the original definition of the codomain of π in [16] allows only positive integers. Our definition, however, stipulates sequences of positive integers. The usefulness of this extension can be seen in Example 4.20.

Theorem 3.17 Let R be an STRS and C be a set of pairs of marked terms. If C satisfies the subterm criterion, then C is non-looping.

Proof. Assume that there exists an infinite $\langle C, \underset{R}{\to} \rangle$-chain on $(\mathcal{T}_{SN}^{args}(R), \mathcal{T}_{SN}^{args}(R))$

$$u_0^\sharp \to v_0^\sharp, u_1^\sharp \to v_1^\sharp, u_2^\sharp \to v_2^\sharp, \cdots$$

in which every $u^\sharp \to v^\sharp \in C$ occurs infinitely many times, and let $\theta_0, \theta_1, \ldots$ be substitutions such that $(v_i\theta_i)^\sharp \overset{*}{\underset{R}{\to}} (u_{i+1}\theta_{i+1})^\sharp$ and $u_i\theta_i, v_i\theta_i \in \mathcal{T}_{SN}^{args}(R)$. Denote $\pi(root(u_i))$ by p_i for each i. Since $(v_i\theta_i)^\sharp \overset{*}{\underset{R}{\to}} (u_{i+1}\theta_{i+1})^\sharp$, we have $root(v_i) = root(u_{i+1})$. From the last two condition in (β) of the subterm criterion, we have $(v_i\theta_i)|_{p_{i+1}} \overset{*}{\underset{R}{\to}} (u_{i+1}\theta_{i+1})|_{p_{i+1}}$ for each i. Hence, from the first condition in (β) of the subterm criterion, we have

$$(u_0\theta_0)|_{p_0} \geq_{esub} (v_0\theta_0)|_{p_1} \overset{*}{\underset{R}{\to}} (u_1\theta_1)|_{p_1} \geq_{esub} (v_1\theta_1)|_{p_2} \overset{*}{\underset{R}{\to}} \cdots.$$

From the condition (α) of the subterm criterion, this sequence contains infinitely many $>_{esub}$. Since $>_{esub}$ is well-founded and $>_{esub} \cdot \underset{R}{\to} \subseteq \underset{R}{\to} \cdot >_{esub}$, there exists an infinite rewriting relation starting from $(u_0\theta_0)|_{p_0}$. Since p_0 is non-empty, there exists a positive integer j such that $j \preceq p_0$. Since $SN(R, (u_0\theta_0)|_j)$ follows from $u_0\theta_0 \in \mathcal{T}_{SN}^{args}(R)$, we have $SN(R, (u_0\theta_0)|_{p_0})$. It is a contradiction. $\square$

By applying Theorem 3.15 and 3.17 to Theorem 3.13, we obtain the following methods for proving the termination of STRSs.

Theorem 3.18 Let R be an STRS such that $DP_{SN}(R)$ is finite. If each $C \in RC_{SN}(R)$ satisfies one of the following properties then R is terminating.

- There exists a reduction pair $(\gtrsim, >)$ such that $R \cup C \subseteq \gtrsim$, $C \cap > \neq \emptyset$ and it satisfies the marked condition in C.
- There exists a semi-reduction pair $(\gtrsim, >)$ such that $R^{ex} \cup C \subseteq \gtrsim$, $C \cap > \neq \emptyset$ and it satisfies the marked condition in C.
- C satisfies the subterm criterion.

This theorem yields a method for proving termination. It is, however, not sufficiently practical. In fact, this theorem does not prove the termination of R_{foldl}, because we do not know any method to design (semi-)reduction pairs that satisfy the properties required by this theorem. In the next section, we will enhance this theorem using strong computability.

4 New Dependency Pair Method based on Strong Computability

In this section, we propose a new dependency pair method based on strong computability, called the SC-dependency pair method. Intuitively, the SN-dependency pair method in [20] analyzes a dynamic recursive structure based on function-call dependency relationships, while the SC-dependency pair method analyzes a static recursive structure based on definition dependency relationships. As a result the SC-dependency pair method ignores terms headed by a higher-order variable, and hence the method is powerful and efficient. On the other hand, the SC-dependency pair method cannot be applied to arbitrary STRSs, because strong computability is not closed under the subterm relation. Hence we provide the notion of plain function-passing STRSs, for which the SC-dependency pair method works well, and by which many non-artificial functional programs can still be represented.

4.1 Strong Computability

First of all, we define strong computability.

Definition 4.1 For each STRS R, we define the set $\times(R)$ of *used product types* as $\alpha \in \times(R)$ iff α is a product type and there exist $l \to r \in R$ and $z \in Var(r)$ such that $\tau(z) = \alpha$. A term t is said to be *strongly computable* in R if $SC(R,t)$ holds, which is inductively defined on simple types as follows:

- in case of $\tau(t) \in \mathcal{B} \cup \times(R)$, $SC(R,t)$ is defined as $SN(R,t)$,
- in case of $\tau(t) = \alpha_1 \times \cdots \times \alpha_n$ and $\tau(t) \notin \times(R)$, $SC(R,t)$ is defined as $SN(R,t)$ and $SC(R,t_i)$ for any t_i such that $t \xrightarrow[R]{*} (t_1, \ldots, t_n)$, and
- in case of $\tau(t) = \alpha \to \beta$, $SC(R,t)$ is defined as
  $(\forall u \in \mathcal{T}_\tau(\Sigma, \mathcal{V}))\ (SC(R,u) \wedge \tau(u) = \alpha \Rightarrow SC(R, t[u]))$.

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

In the case of $\tau(t) = \alpha_1 \times \cdots \times \alpha_n$, the type $\tau(t_i) = \alpha_i$ is smaller than $\tau(t)$, and in the case of $\tau(t) = \alpha \to \beta$, the types $\tau(u) = \alpha$ and $\tau(t[u]) = \beta$ are smaller than $\tau(t)$. Hence the definition above is well-defined.

Strong computability is not closed under the subterm relation: for a term $t \equiv a[t_1, \ldots, t_n]$ with $\tau(a) = \alpha_1 \to \cdots \to \alpha_n \to \alpha$, the type α_i of a subterm t_i may not be smaller than α. Product types, however, are an exceptional case: for a term $t \equiv (t_1, \ldots, t_n)$ with $\tau(t) = \alpha_1 \times \cdots \times \alpha_n$, the type α_i of a subterm t_i is smaller than $\alpha_1 \times \cdots \times \alpha_n$.

In order to combine the dependency pair method with strong computability, the compatibility between structures of terms and types is the most important and difficult task. In order to strengthen the SC-dependency pair method, it is ideal to guarantee the strong computability of subterms as far as possible. Hence product types are treated specially in the definition above. This topic will be further discussed in Example 4.4.

We now present the basic properties of strong computability.

Lemma 4.2 Let t be a term such that $\tau(t) = \alpha_1 \to \cdots \to \alpha_n \to \alpha$, $\alpha \notin \mathcal{S}_{fun}$ and $\neg SC(R,t)$. Then, there exist terms u_i $(1 \leq i \leq n)$ such that $\forall i.(\tau(u_i) = \alpha_i \wedge SC(R,u_i))$ and $\neg SC(R, t[u_1,\ldots,u_n])$.

Proof. We prove the claim by induction on n. The case $n = 0$ is trivial. Suppose that $n > 0$. From $\neg SC(R,t)$, there exists a term u_1 such that $\tau(u_1) = \alpha_1$, $SC(R,u_1)$ and $\neg SC(R,t[u_1])$. From $\tau(t[u_1]) = \alpha_2 \to \cdots \to \alpha_n \to \alpha$ and the induction hypothesis, there exist terms $u_2,\ldots,u_n$ such that $\forall i \geq 2.(\tau(u_i) = \alpha_i \wedge SC(R,u_i))$ and $\neg SC(R,t[u_1][u_2,\ldots,u_n])$. □

Lemma 4.3 For any STRS R, the following properties hold:

(1) Any variable z with $\tau(z) = \alpha$ is strongly computable, for all $\alpha \in \mathcal{S}$.

(2) $SC(R,t) \wedge \tau(t) = \alpha \Rightarrow SN(R,t)$, for all $\alpha \in \mathcal{S}$.

(3) $SC(R,t) \wedge t \xrightarrow[R]{*} t' \Rightarrow SC(R,t')$ for all t and t'.

(4) $\bigwedge_{i=0}^{n} SC(R,t_i) \Rightarrow SC(R, t_0[t_1,\ldots,t_n])$
for all $t_0, t_1, \ldots, t_n$ such that $t_0[t_1,\ldots,t_n] \in \mathcal{T}_\tau(\Sigma,\mathcal{V})$.

Proof.

(1,2) We prove the claim by induction on α.

The case $\alpha \in \mathcal{B} \cup \times(R)$ is trivial.

In case of $\alpha = \alpha_1 \times \cdots \times \alpha_n \notin \times(R)$, $SN(R,t)$ and $SN(R,z)$ are trivial and hence $SC(R,z)$ is also trivial because of $z \xrightarrow[R]{*} t' \Rightarrow z \equiv t'$.

Suppose that $\alpha = \alpha_1 \to \cdots \to \alpha_n \to \beta$ and $\beta \notin \mathcal{S}_{fun}$. From the induction hypothesis, an arbitrary variable z_1 with $\tau(z_1) = \alpha_1$ is strongly computable. Thus $SC(R,t[z_1])$ holds. From the induction hypothesis, $t[z_1]$ is terminating, hence so is t.

Assume that z is not strongly computable. From Lemma 4.2, there exist terms $u_1,\ldots,u_n$ such that $\forall i(\tau(u_i) = \alpha_i \wedge SC(R,u_i))$ and $z[u_1,\ldots,u_n]$ is not strongly computable. From the induction hypothesis, each u_i is terminating. Hence $z[u_1,\ldots,u_n]$ is terminating. Since $z[u_1,\ldots,u_n]$ is not strongly computable and its type β is a non-functional type, β is a product type such that $\beta \notin \times(R)$, and there exists a term t' such that t' is not strongly computable and $z[u_1,\ldots,u_n] \xrightarrow[R]{*} (\ldots,t',\ldots)$. It is a contradiction because $root(l) \notin \mathcal{V}$ for all $l \to r \in R$.

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

(4) We prove the claim by induction on n. The case $n = 0$ is trivial. Suppose that $n > 0$. From the induction hypothesis, $SC(R,t_0[t_1,\ldots,t_{n-1}])$ holds. Hence $SC(R,t_0[t_1,\ldots,t_n])$ follows from the definition of SC. □

4.2 Plain Function-Passing

Strong computability is not closed under the subterm relation, because it is defined inductively over "types", rather than "terms". This breaks the soundness of recursive structure analysis using our SC-dependency pair method, which will be introduced in the next subsection. We have already explained that the SC-dependency pair method analyzes a static recursive structure based on definition dependency. This means that if an STRS has no static recursive structure then it should be terminating. However, there exist some non-terminating STRSs which have no static recursive structure. For example, the STRS $R = \{foo[bar[f]] \to f[bar[f]]\}$ has no static recursive structure but it is non-terminating since $foo[bar[foo]] \xrightarrow[R]{} foo[bar[foo]]$. We now consider a more detailed example:

Example 4.4 Let R'_{bar} and R_{bar} be STRSs defined as follows:

$$\begin{array}{rcl} R'_{bar} & = & \{fst[(f,x)] \to f,\ apply[f,x] \to f[x]\} \\ R_{bar} & = & R'_{bar} \cup \{bar[baz[z]] \to apply[fst[z], baz[z]]\} \end{array}$$

Here $\tau(x) = N$, $\tau(f) = N \to N$, $\tau(z) = (N \to N) \times N$, $\tau(\mathit{fst}) = (N \to N) \times N \to N \to N$, $\tau(\mathit{apply}) = (N \to N) \to N \to N$, $\tau(\mathit{bar}) = N \to N$, and $\tau(\mathit{baz}) = (N \to N) \times N \to N$. Although R'_{bar} is terminating, R_{bar} is not so.

$$\begin{array}{rcl} \mathit{bar}[\mathit{baz}[(\mathit{bar}, x)]] & \to & \mathit{apply}[\mathit{fst}[(\mathit{bar}, x)], \mathit{baz}[(\mathit{bar}, x)]] \\ & \to & \mathit{apply}[\mathit{bar}, \mathit{baz}[(\mathit{bar}, x)]] \\ & \to & \mathit{bar}[\mathit{baz}[(\mathit{bar}, x)]] \end{array}$$

Notice that R'_{bar} and R_{bar} have no static recursive structure. Nevertheless R'_{bar} is terminating, but R_{bar} is not. This fact indicates that we need some restriction described by an appropriate condition in order to make the SC-dependency pair method sound. Moreover it is required to find a condition satisfied by R'_{bar} but not satisfied by R_{bar}. One such restriction is made possible by the concept of plain function-passing. This concept is concerning particular kinds of argument where higher-order variables occur. As a consequence, we want to interpret many positions as certain kinds of argument. In order to define plain function-passing, we present the notion of extended arguments and safe subterms. Note that the higher-order variable f in $\mathit{fst}[(f, x)]$ is an extended argument in R'_{bar}, but not an extended argument in R_{bar}. This delicate clarification becomes possible by using the concept of used product types.

Definition 4.5 For each STRS R and term l, we define the set $e\_args(R, l)$ of *extended arguments* as follows:

- Any argument $u \in args(l)$ is an extended argument $u \in e\_args(R, l)$.
- If an extended argument $u \in e\_args(R, l)$ is a tuple $u \equiv (\ldots, u_i, \ldots)$ such that $\tau(u) \notin \times(R)$, then $u_i \in e\_args(R, l)$.

For each STRS R and term l, we define the set $\mathit{safe}(R, l)$ of *safe subterms* as follows:

$$\mathit{safe}(R, l) = e\_args(l) \cup \{u \in Sub(l) \mid u \not\equiv l, \tau(u) \in \mathcal{B} \cup \times(R)\}$$

Example 4.6 Referencing to Example 4.4. Then used product types are $\times(R'_{bar}) = \emptyset$ and $\times(R_{bar}) = \{(N \to N) \times N\}$. Hence extended arguments in $\mathit{fst}[(f, x)]$ are the following:

$$\begin{array}{rcl} e\_args(R'_{bar}, \mathit{fst}[(f, x)]) & = & \{(f, x), f, x\} \\ e\_args(R_{bar}, \mathit{fst}[(f, x)]) & = & \{(f, x)\} \end{array}$$

Hence safe terms in $\mathit{fst}[(f, x)]$ are the following:

$$\begin{array}{rcl} \mathit{safe}(R'_{bar}, \mathit{fst}[(f, x)]) & = & \{(f, x), f, x\} \\ \mathit{safe}(R_{bar}, \mathit{fst}[(f, x)]) & = & \{(f, x), x\} \end{array}$$

We now present the concept of plain function-passing, according to which the SC-dependency pair method works well.

Definition 4.7 An STRS R is said to be *plain function-passing* if for any leaf-context $C[\,]$ and $l \to C[a[r_1, \ldots, r_m]] \in R$ such that $a \in \mathcal{V}$, there exists a number k ($\leq m$) such that $a[r_1, \ldots, r_k] \in \mathit{safe}(R, l)$. A plain function-passing STRS is abbreviated to PFP-STRS.

Example 4.8 Referencing to Example 4.4 and 4.6. Then we have

$$\begin{array}{rcccl} f & \in & \{(f, x), f, x\} & = & \mathit{safe}(R'_{bar}, \mathit{fst}[(f, x)]), \text{ and} \\ f & \notin & \{(f, x), x\} & = & \mathit{safe}(R_{bar}, \mathit{fst}[(f, x)]). \end{array}$$

Hence STRS R'_{bar} is plain function-passing, but R_{bar} is not so.

Every terminating STRS in this paper is plain function-passing, and many non-artificial functional programs are expressed in plain function-passing STRSs. This fact demonstrates the versatility of the SC-dependency pair method.

Finally, we present a lemma which will be used for analyzing dependency chains with respect to strong computability (cf. Lemma 4.16).

Lemma 4.9 Let R be an STRS, θ be a substitution, $C[\,]$ be a root-context, and $l \to r \in R$ such that $C[l]\theta \in \mathcal{T}^{args}_{SC}(R)$. Then $SC(R, u\theta)$ holds for any $u \in \mathit{safe}(R, l)$.

Proof. We have $l\theta \in \mathcal{T}^{args}_{SC}(R)$ from $C[l]\theta \in \mathcal{T}^{args}_{SC}(R)$.

Suppose that $u \in Sub(l)$ such that $u \not\equiv l$ and $\tau(u) \in \mathcal{B} \cup \times(R)$. Then $u \in Sub(u')$ for some $u' \in args(l)$. Since $l\theta \in \mathcal{T}^{args}_{SC}(R)$ and Lemma 4.3(2), $SN(R, u'\theta)$ holds. Hence $SN(R, u\theta)$ holds, which implies $SC(R, u\theta)$.

Suppose that $u \in e\_args(l)$. We prove the claim by induction on the definition of e_args. In the case of $u \in args(l)$, $SC(R, u\theta)$ follows from $l\theta \in \mathcal{T}^{args}_{SC}(R)$. Suppose that $(\ldots, u, \ldots) \in e\_args(l)$ and $\tau((\ldots, u, \ldots)) \notin \times(R)$. From the induction hypothesis, we have $SC(R, (\ldots, u, \ldots)\theta)$. Since $(\ldots, u, \ldots)\theta = (\ldots, u\theta, \ldots)$ and $\tau((\ldots, u, \ldots)\theta) = \tau((\ldots, u, \ldots)) \notin \times(R)$, we have $SC(R, u\theta)$. $\square$

4.3 SC-dependency pair Method

In this subsection, we present the SC-dependency pair method.

Definition 4.10 Let R be an STRS and $l \to r \in R$ such that $\tau(l) = \alpha_1 \to \cdots \to \alpha_n \to \alpha$ and $\alpha \notin \mathcal{S}_{\mathit{fun}}$. The rule $(l \to r)^{ex\uparrow}$ of the *full expansion form* of $l \to r$ is defined as $l[z_1, \ldots, z_n] \to r[z_1, \ldots, z_n]$, where $z_1, \ldots, z_n$ are fresh variables with $\forall i.\tau(z_i) = \alpha_i$. We also define $R^{ex\uparrow} = \{(l \to r)^{ex\uparrow} \mid l \to r \in R\}$.

Definition 4.11 Let R be an STRS. For each $l \to C[a[r_1, \ldots, r_m]] \in R^{ex\uparrow}$ such that

- $a \in \mathcal{D}_R$,
- $C[\,]$ is a leaf-context, and
- $\forall k \leq m.\ a[r_1, \ldots, r_k] \notin \mathit{safe}(R, l)$,

we define an *SC-dependency pair* of R as $l^\sharp \to a^\sharp[r_1, \ldots, r_m, z_1, \ldots, z_n]$, where $\tau(a^\sharp[r_1, \ldots, r_m, z_1, \ldots, z_n]) \notin \mathcal{S}_{\mathit{fun}}$ and $z_1, \ldots, z_n$ are fresh variables. We denote by $DP_{SC}(R)$ the set of all SC-dependency pairs of R.

Notice that SC-dependency pairs have no terms headed by a higher-order variable nor terms of a functional type.

Example 4.12 Referencing to Example 3.4. Then $DP_{SC}(R'_{foo})$ is the following:

$$DP_{SC}(R'_{foo}) = \left\{ \begin{array}{l} foldl^\sharp[f, y, cons[(x, xs)]] \to foldl^\sharp[f, f[(x, y)], xs] \\ foo^\sharp[s[i], (x, y)] \to foldl^\sharp[foo[i], y, cons[(x, nil)]] \\ foo^\sharp[s[i], (x, y)] \to foo^\sharp[i, z] \end{array} \right.$$

Comparing $DP_{SC}(R'_{foo})$ with $DP_{SN}(R'_{foo})$ in Example 3.4, the number of pairs has been reduced from 5 to 3, and the pairs whose right-hand side is headed by a higher-order variable are not SC-dependency pairs. For instance, the pair $foldl^\sharp[f, z, cons[(x, xs)]] \to f[(z, x)]$ is not in $DP_{SC}(R'_{foo})$, but it is in $DP_{SN}(R'_{foo})$.

Definition 4.13 A DP_{SC}*-chain* (resp. DP_{SC}*-graph*) is a $\langle DP_{SC}(R), \to_R\rangle$-chain (resp. -graph) on $(\mathcal{T}^{args}_{SC}(R), \mathcal{T}^{args}_{SC}(R))$. We denote by $RC_{SC}(R)$ the set of all $\langle DP_{SC}(R), \to_R\rangle$-recursion components on $(\mathcal{T}^{args}_{SC}(R), \mathcal{T}^{args}_{SC}(R))$.

Example 4.14 The set $RC_{SC}(R'_{foo})$ is the following:

$$\boxed{foldl^\sharp[f,y,cons[(x,xs)]] \to foldl^\sharp[f,f[(x,y)],xs]}$$

$$\boxed{foo^\sharp[s[i],(x,y)] \to foldl^\sharp[foo[i],y,cons[(x,nil)]]}$$

$$\boxed{foo^\sharp[s[i],(x,y)] \to foo^\sharp[i,z]}$$

Hence the recursion components $RC_{SC}(R'_{foo})$ is constructed by the following two components:

$$RC_{SC}(R'_{foo}) = \left\{ \begin{array}{l} \{\ foldl^\sharp[f,y,cons[(x,xs)]] \to foldl^\sharp[f,f[(x,y)],xs]\ \} \\ \{\ foo^\sharp[s[i],(x,y)] \to foo^\sharp[i,z]\ \} \end{array} \right.$$

We see that $RC_{SC}(R'_{foo})$ is much smaller and simpler than $RC_{SN}(R'_{foo})$ in Example 3.11.

Next we present the fundamental theorem of the SC-dependency pair method. In order to do this, we present the following two lemmas, which correspond to last two properties in Theorem 3.7 with $T = \mathcal{T}_\tau(\Sigma,\mathcal{V}) \setminus (\mathcal{T}_{fun}(\Sigma,\mathcal{V}) \cup \mathcal{T}_{SC}(R))$ and $T_l = T_r = \mathcal{T}^{args}_{SC}(R)$.

Lemma 4.15 If an STRS R is not terminating then the set $(\mathcal{T}_\tau(\Sigma,\mathcal{V}) \setminus (\mathcal{T}_{fun}(\Sigma,\mathcal{V}) \cup \mathcal{T}_{SC}(R))) \cap \mathcal{T}^{args}_{SC}(R)$ is not empty.

Proof. From the contraposition of Lemma 4.3(2), $\mathcal{T}_\tau(\Sigma,\mathcal{V}) \setminus \mathcal{T}_{SC}(R) \neq \emptyset$ holds.

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

Let t be a minimal term in $\mathcal{T}^{args}_{SC}(R) \setminus \mathcal{T}_{SC}(R)$ with respect to type size. Assume that $t \in \mathcal{T}_{fun}(\Sigma,\mathcal{V})$. Let $\tau(t) = \alpha \to \beta$ and u be an arbitrary strongly computable term with $\tau(u) = \alpha$. Since $t \in \mathcal{T}^{args}_{SC}(R)$ and $u \in \mathcal{T}_{SC}(R)$, we have $t[u] \in \mathcal{T}^{args}_{SC}(R)$. From $\tau(t[u]) = \beta$ and the minimality of $\tau(t) = \alpha \to \beta$, we have $t[u] \notin \mathcal{T}^{args}_{SC}(R) \setminus \mathcal{T}_{SC}(R)$, hence $t[u] \in \mathcal{T}_{SC}(R)$. Then we have $t \in \mathcal{T}_{SC}(R)$, which is a contradiction. Hence $t \notin \mathcal{T}_{fun}(\Sigma,\mathcal{V})$. Therefore we have $t \in (\mathcal{T}_\tau(\Sigma,\mathcal{V}) \setminus (\mathcal{T}_{fun}(\Sigma,\mathcal{V}) \cup \mathcal{T}_{SC}(R))) \cap \mathcal{T}^{args}_{SC}(R)$. $\square$

Lemma 4.16 Let R be a PFP-STRS. For any $t \in (\mathcal{T}_\tau(\Sigma,\mathcal{V}) \setminus (\mathcal{T}_{fun}(\Sigma,\mathcal{V}) \cup \mathcal{T}_{SC}(R))) \cap \mathcal{T}^{args}_{SC}(R)$, there exist $l^\sharp \to v^\sharp \in DP_{SC}(R)$ and θ such that $t^\sharp \xrightarrow{*} (l\theta)^\sharp$ and $l\theta, v\theta \in (\mathcal{T}_\tau(\Sigma,\mathcal{V}) \setminus (\mathcal{T}_{fun}(\Sigma,\mathcal{V}) \cup \mathcal{T}_{SC}(R))) \cap \mathcal{T}^{args}_{SC}(R)$.

Proof. Let $t \in (\mathcal{T}_\tau(\Sigma,\mathcal{V}) \setminus (\mathcal{T}_{fun}(\Sigma,\mathcal{V}) \cup \mathcal{T}_{SC}(R))) \cap \mathcal{T}^{args}_{SC}(R)$. Then $t \in \mathcal{T}^{args}_{SN}(R)$ follows from $t \in \mathcal{T}^{args}_{SC}(R)$ and Lemma 4.3(2).

- Consider the case that $t \notin \mathcal{T}_{SN}(R)$. Since $t \in \mathcal{T}^{args}_{SN}(R) \setminus \mathcal{T}_{fun}(\Sigma,\mathcal{V})$, there exist $l \to r \in R^{ex\uparrow}$ and θ' such that $t^\sharp \xrightarrow{*} (l\theta')^\sharp$, $\neg SN(R, l\theta')$ and $\neg SN(R, r\theta')$. Hence $\neg SC(R, l\theta')$ and $\neg SC(R, r\theta')$ follows from Lemma 4.3(2).
- Consider the case that $t \in \mathcal{T}_{SN}(R)$. Since $t \notin \mathcal{T}_{SC}(R) \cup \mathcal{T}_{fun}(\Sigma,\mathcal{V})$, $\tau(t)$ is a product type not in $\times(R)$, and there exists a term $(u_1,\ldots,u_n)$ such that $t \xrightarrow{*} (u_1,\ldots,u_n)$ and $\neg SC(R, u_i)$ for some i. Since $t \in \mathcal{T}^{args}_{SC}(R)$, $root(t)$ is not tp but a defined symbol. Thus there exist $l \to r \in R^{ex\uparrow}$ and θ' such that $t^\sharp \xrightarrow{*} (l\theta')^\sharp$ and $l\theta' \to r\theta' \xrightarrow{*} (u_1,\ldots,u_n)$. Here $\neg SC(R, l\theta')$ and $\neg SC(R, r\theta')$ holds.

In both cases above, we have $\{v' \in Sub(r) \mid \neg SC(R, v'\theta')\} \neq \emptyset$ since $r \in Sub(r)$ and $\neg SC(R, r\theta')$. Let $v' \equiv a[r_1,\ldots,r_m]$ be a minimal size term in this set. Then $SC(R, r_i\theta')$ holds for every i. From Lemma 4.2, there exist $v_1,\ldots,v_k \in \mathcal{T}_{SC}(R)$ $(k \geq 0)$ such that $\tau(v'\theta'[v_1,\ldots,v_k]) \notin \mathcal{S}_{fun}$ and $\neg SC(v'\theta'[v_1,\ldots,v_k])$. Here $v'\theta'[v_1,\ldots,v_k] \in \mathcal{T}^{args}_{SC}(R)$.

Now let $v \equiv a[r_1,\ldots,r_m,z_1,\ldots,z_k]$ for fresh variables $z_1,\ldots,z_k$ and we define $\theta(x)$ by v_i if $x = z_i$ $(i = 1,\ldots,k)$; otherwise by $\theta'(x)$. Then we have $l\theta = l\theta'$ and $v\theta = v'\theta'[v_1,\ldots,v_k]$. Since $l\theta = l\theta' \in \mathcal{T}^{args}_{SC}(R)$ by $t \in \mathcal{T}^{args}_{SC}(R)$ and Lemma 4.3(3), we have $l\theta \in (\mathcal{T}_\tau(\Sigma,\mathcal{V}) \setminus (\mathcal{T}_{fun}(\Sigma,\mathcal{V}) \cup \mathcal{T}_{SC}(R))) \cap \mathcal{T}^{args}_{SC}(R)$. Because $v\theta \in$

$(\mathcal{T}_\tau(\Sigma,\mathcal{V}) \setminus (\mathcal{T}_{\mathit{fun}}(\Sigma,\mathcal{V}) \cup \mathcal{T}_{SC}(R))) \cap \mathcal{T}^{args}_{SC}(R)$ also holds, it suffices to show that $l^\sharp \to v^\sharp \in DP_{SC}(R)$. We prove this by contradiction. Assume that $l^\sharp \to v^\sharp \notin DP_{SC}(R)$. Let $l \equiv l'[z'_1,\ldots,z'_p]$ and $r \equiv r'[z'_1,\ldots,z'_p]$ such that $l' \to r' \in R$ and $z'_1,\ldots,z'_p$ are fresh variables.

- Assume that $a \in Var(l')$. Since R is plain function-passing, there exists a number k ($\leq m$) such that $a[r_1,\ldots,r_k] \in \mathit{safe}(R,l')$. From Lemma 4.9 and 4.3(4), we have $SC(R, v\theta)$. This is a contradiction.
- Assume that $a = z'_i$ for some i. Then $a \in args(l)$. Since $l\theta \in \mathcal{T}^{args}_{SC}(R)$, $SC(R, a\theta)$ holds. From Lemma 4.3(4) we have $SC(R, v\theta)$. This is a contradiction.
- Assume that $a \in \mathcal{C}_R$. Since $r_1\theta,\ldots,r_m\theta, v_1,\ldots,v_k$ are terminating from Lemma 4.3(2), $SN(R, v\theta)$ holds. Since $v\theta \notin \mathcal{T}_{\mathit{fun}}(\Sigma,\mathcal{V}) \cup \mathcal{T}_{SC}(R)$, a must be tp and there exists a term $(v'_1,\ldots,v'_p)$ such that $v\theta \xrightarrow[R]{*} (v'_1,\ldots,v'_p)$ and $\neg SC(R, v'_i)$ for some i. Since $root(v) = a \in \mathcal{C}_R$ and $v\theta \in \mathcal{T}^{args}_{SC}(R)$, each $SC(R, v'_i)$ follows from Lemma 4.3(3). This is a contradiction.
- Assume that $a \in \mathcal{D}_R$ and $a[r_1,\ldots,r_k] \in \mathit{safe}(R,l')$ for some $k \leq m$. From Lemma 4.9, we have $SC(R, a[r_1,\ldots,r_k]\theta)$. From Lemma 4.3(4), $SC(R, v\theta)$ holds. This is a contradiction. □

By applying Lemmas 4.15 and 4.16 to Theorem 3.7 with $T = \mathcal{T}_\tau(\Sigma,\mathcal{V}) \setminus (\mathcal{T}_{\mathit{fun}}(\Sigma,\mathcal{V}) \cup \mathcal{T}_{SC}(R))$ and $T_l = T_r = \mathcal{T}^{args}_{SC}(R)$, we obtain the fundamental theorem of the SC-dependency pair method.

Theorem 4.17 Let R be a PFP-STRS. If there exists no infinite $DP_{SC}(R)$-chain then R is terminating.

Note that the ($\Leftarrow$)-part does not hold in the theorem. The SC-dependency pair method therefore has a theoretical limitation (as described in the concluding remarks).

We also obtain the following theorem from this theorem and Theorem 3.12.

Theorem 4.18 Let R be a PFP-STRS such that $DP_{SC}(R)$ is finite. If all recursion components in $RC_{SC}(R)$ are non-looping then R is terminating.

Finally, by applying Theorem 3.15 and 3.17 to this theorem, we obtain a powerful and efficient method for proving termination. Note that the marked condition for (semi-)reduction pairs always holds because SC-dependency pairs have no terms headed by a higher-order variable.

Theorem 4.19 Let R be a PFP-STRS such that $DP_{SC}(R)$ is finite. If any $C \in RC_{SC}(R)$ satisfies one of the following properties, then R is terminating.

- There exists a reduction pair $(\gtrsim, >)$ such that $R \cup C \subseteq \gtrsim$ and $C \cap > \neq \emptyset$.
- There exists a semi-reduction pair $(\gtrsim, >)$ such that $R^{ex} \cup C \subseteq \gtrsim$ and $C \cap > \neq \emptyset$.
- C satisfies the subterm criterion.

4.4 Examples

In this subsection, we give some examples in order to demonstrate the power of the SC-dependency pair method. Note that every STRS in this subsection is plain function-passing, because each higher-order variable of the right hand sides occurs in an argument position on the left hand side.

Example 4.20 We show the termination of the PFP-STRS $R_{sum} \cup R_{prod}$, displayed in the introduction. $DP_{SC}(R_{sum} \cup R_{prod})$ is the following:

$$DP_{SC}(R_{sum} \cup R_{prod}) = \left\{ \begin{array}{l} +^\sharp[(x, s[y])] \to +^\sharp[(x, y)] \\ \times^\sharp[(x, s[y])] \to +^\sharp[(\times[(x, y)], x)] \\ \times^\sharp[(x, s[y])] \to \times^\sharp[(x, y)] \\ foldl^\sharp[f, y, cons[(x, xs)]] \to foldl^\sharp[f, f[(x, y)], xs] \\ sum^\sharp[zs] \to foldl^\sharp[+, 0, zs] \\ sum^\sharp[zs] \to +^\sharp[z] \\ prod^\sharp[zs] \to foldl^\sharp[\times, s[0], zs] \\ prod^\sharp[zs] \to \times^\sharp[z] \end{array} \right.$$

The set $RC_{SC}(R_{sum} \cup R_{prod})$ is constructed by the following three components:

$$RC_{SC}(R_{sum} \cup R_{prod}) = \begin{cases} \{\ +^\sharp[(x, \underline{s[y]})] \to +^\sharp[(x, \underline{y})]\ \} \\ \{\ \times^\sharp[(x, \underline{s[y]})] \to \times^\sharp[(x, \underline{y})]\ \} \\ \{\ foldl^\sharp[f, y, \underline{cons[(x, xs)]}] \to foldl^\sharp[f, f[(x, y)], \underline{xs}]\ \} \end{cases}$$

By $\pi(+) = 1 \cdot 2$, the first component satisfies the subterm criterion. By $\pi(\times) = 1 \cdot 2$, the second component satisfies the subterm criterion. By $\pi(foldl) = 3$, the third component satisfies the subterm criterion. Hence the termination of $R_{sum} \cup R_{prod}$ is guaranteed by Theorem 4.19.

Example 4.21 Since all $C \in RC_{SC}(R'_{foo})$, displayed in Example 4.12 and 4.14, satisfies the subterm criterion, the termination of R'_{foo} is guaranteed by Theorem 4.19. Actually, each component satisfies the subterm criterion by setting π to the underlined parts below ($\pi(foldl) = 2$ and $\pi(foo) = 1$):

$$\{foldl^\sharp[f, y, \underline{cons[(x, xs)]}] \to foldl^\sharp[f, f[(x, y)], \underline{xs}]\}$$
$$\{foo^\sharp[\underline{s[i]}, (x, y)] \to foo^\sharp[\underline{i}, z]\}$$

Example 4.22 The right-folding function $foldr$, which is also a built-in function of the functional programing language SML, can be represented as the following PFP-STRS:

$$R_{foldr} = \begin{cases} foldr[f, y, nil] & \to & y \\ foldr[f, y, cons[(x, xs)]] & \to & f[(x, foldr[f, y, xs])] \end{cases}$$

Then $DP_{SC}(R_{foldr}) = \{foldr^\sharp[f, y, cons[(x, xs)]] \to foldr^\sharp[f, y, xs]\}$ and $RC_{SC}(R_{foldr}) = \{DP_{SC}(R_{foldr})\}$. By $\pi(foldr) = 3$, the component satisfies the subterm criterion. Hence the termination is guaranteed by Theorem 4.19.

Example 4.23 Typical higher-order functions map and $filter$ can be represented as the following PFP-STRSs:

$$R_{map} = \begin{cases} map[f, nil] & \to & nil \\ map[f, cons[(x, xs)]] & \to & cons[(f[x], map[f, xs])] \end{cases}$$

$$R_{flt} = \begin{cases} if[true, x, y] & \to & x \\ if[false, x, y] & \to & y \\ filter[p, nil] & \to & nil \\ filter[p, cons[(x, xs)]] & \to & if[p[x], cons[(x, filter[p, xs])], filter[p, xs]] \end{cases}$$

The set $RC_{SC}(R_{map} \cup R_{flt})$ is constructed by the following two components:

$$RC_{SC}(R_{map} \cup R_{flt}) = \begin{cases} \{\ map^\sharp[f, \underline{cons[(x, xs)]}] \to map^\sharp[f, \underline{xs}]\ \} \\ \{\ filter^\sharp[p, \underline{cons[(x, xs)]}] \to filter^\sharp[p, \underline{xs}]\ \} \end{cases}$$

By setting π to the underlined parts ($\pi(map) = \pi(filter) = 2$), any component satisfies the subterm criterion. Hence the termination of $R_{map} \cup R_{flt}$ is guaranteed by Theorem 4.19.

Example 4.24 We show the termination of R'_{bar}, displayed in Example 4.4. Since $DP_{SC}(R'_{bar}) = \emptyset$, we have $RC_{SC}(R'_{bar}) = \emptyset$. Hence the termination of R'_{bar} is guaranteed by Theorem 4.19.

This example is a typical case for which the SC-dependency pair method is particularly effective because it is not necessary for the SC-dependency method to analyze higher-order variables. Simply-typed combinatory logic is also such a case.

Example 4.25 For each $\alpha, \beta, \gamma \in \mathcal{S}$, we define $R_{\alpha,\beta,\gamma}$ as follows:

$$R_{\alpha,\beta,\gamma} = \begin{cases} S_{\alpha,\beta,\gamma}[f, g, x] & \to & f[x, g[x]] \\ K_{\alpha,\beta}[x, y] & \to & x \end{cases}$$

where $\tau(S_{\alpha,\beta,\gamma}) = (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$ and $\tau(K_{\alpha,\beta}) = \alpha \to \beta \to \alpha$. We define STRS R_{CL}, which is a representation of simply-typed combinatory logic, as follows:

$$R_{CL} = \bigcup_{\alpha,\beta,\gamma \in \mathcal{S}} R_{\alpha,\beta,\gamma}$$

Since no defined symbol occurs in any right-hand side, we have $DP_{SC}(R_{CL}) = \emptyset$, hence $RC_{SC}(R_{CL}) = \emptyset$. Thus the termination of R_{CL} follows from Theorem 4.19.

Although several proofs of the termination of simply-typed combinatory logic is known [15], our proof is very elegant.

5 Concluding Remarks

In this paper we reformulated the SN-dependency pair method in [20] and presented the SC-dependency pair method which is based on strong computability. Since there are STRSs on which the SC-dependency pair method fails (cf. Example 4.4), we propose the notion of plain function-passing STRSs on which the SC-dependency pair method always succeeds. This restriction is acceptable since many natural functional programs are plain function-passing.

Since both of the SN-dependency pair and the SC-dependency pair methods are based on the same abstract framework introduced in Section 3, we can see that the only practical difference is in the definition of dependency pairs. Intuitively, the SN-dependency pair method analyzes a dynamic recursive structure based on function-call dependency relationships, while the SC-dependency pair method analyzes a static recursive structure based on definition dependency relationships. The SC-dependency pair method is thus able to ignore terms headed by a higher-order variable. As a result, the SC-dependency pair method is extremely powerful and efficient. Indeed, it proves the termination of simply-typed combinatory logic from two facts which can be verified easily: static recursive structure does not exist, and each higher-order variable occurs in an argument on the left-hand side (cf. Example 4.25). In [30], we also proposed a dependency pair method, which analyzes a static recursive structure. However, this result did not consider strong computability, and hence it required very strong restriction: strongly linear or non-nested.

When verifying termination using the SN-dependency pair method, we cannot ignore terms which are headed by a higher-order variable and thus difficult to handle. One solution is to dynamically analyze the recursive structure (taking into consideration all functions that can be substituted into higher-order variables). This is known as parameterized programming [14] or defunctionalization [28]. The method in [2] is based on such an approach. However, the dynamical analysis approach not only may give rise to a large amount of computation, but also requires an entire analysis whenever function definitions are added. Consequently, This approach is not suitable for incremental proofs, since in this case its capabilities degrades considerably. Moreover, this approach cannot be applied to infinite systems (cf. Example 4.25).

On the other hand, in some cases the SN-dependency pair method is still valuable, because the inverse of Theorem 4.17 does not hold, in contrast to Theorem 3.8. This means that the SC-dependency pair method has a theoretical limitation, which is not required in the SN-dependency pair method. For example, let R_{fix} be the following PFP-STRS:

$$R_{fix} = \{fix[f,x] \to f[fix[f],x]\}$$

Although R_{fix} is terminating, the infinite sequence composed of the dependency pair $fix^{\sharp}[f,x] \to fix^{\sharp}[f,z] \in DP_{SC}(R_{fix})$ is an infinite SC-dependency chain. Hence the SC-dependency pair method cannot prove the termination of R_{fix}. Although we feel that such cases are rare, they require clarification and we will consider this in future works.

Regarding other generic methods that can prove the termination of simply-typed combinatory logic, we are aware of the computational closure proposed by Jouannaud and Rubio [17], and its extension proposed by Blanqui [4]. This concept is based on strong computability and recursive path order [6]. In first-order settings, the recursive path order as a reduction order, and the dependency pair method work in complement to each other, through an argument filtering method which generates a reduction pair from a given reduction order [3]. The argument filtering method for STRSs was proposed in [20]. However, since the argument filtering method eliminates unnecessary subterms and hence destroys type structure, a restriction is necessary in order to use the recursive path order based on strong computability [21]. We intend to design an argument filtering method applicable to this computational closure. Moreover, the notions of accessible subterms in [4] which are an extension of computational closure in [17], and safe subterms (Definition 4.5) originate from the same motivation: to guarantee strong computability of subterms as far as possible. On the other hand, there is no concept corresponding to the used product type (cf. Definition 4.1) in accessible subterms, which is necessary for the SC-dependency pair method (cf. Example 4.4). We intend to clarify the difference between accessible subterms and safe subterms in future work.

Further research will extend the SC-dependency pair method to STRSs with λ-abstractions and/or polymorphic types. Further research will also involve formalizing and implementing methods of proof proposed for first-order TRSs and based on SC-dependency pairs, *e.g.*, usable rules [16, 35][3] and incremental proofs [39]. We also hope to see the results of this research applied to inductive reasoning [22] and the Knuth-Bendix procedure [23] on STRSs.

Acknowledgement

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

This research was partially supported by MEXT.KAKENHI #15500007 and #16300005, by Artificial Intelligence Research Promotion Foundation, and by the 21st Century COE Program (Intelligent Media Integration for Social Infrastructure), Nagoya University.

References

[1] Anderson,H., Khoo,S.C.: Affine-Based Size-Change Termination. In: *Proc. of the 1st Asign Symp. on Programming Languages and Systems*, LNCS 2895 (APLAS2003), pp.122–140, 2003.

[2] Aoto,T., Yamada,T.: Dependency Pairs for Simply Typed Term Rewriting. In: *Proc. of the 16th Int. Conf. on Rewriting Techniques and Applications*, LNCS 3467 (RTA2005), pp.120–134, 2005.

[3] Arts,T., Giesl,J.: Termination of Term Rewriting Using Dependency Pairs. *Theoretical Computer Science*, Vol.236, pp.133–178, 2000.

[4] Blanqui,F.: Termination and Confluence of Higher-Order Rewrite Systems. In: *Proc. of the 11th Int. Conf. on Rewriting Techniques and Applications*, LNCS 1833 (RTA2000), pp.47–61, 2000.

[5] Blanqui,F.: Higher-order Dependency Pairs. In: *Proc. of the 8th Int. Workshop on Termination* (WST06), pp.22–26, 2006.

[6] Dershowitz,N.: Orderings for Term-Rewriting Systems. *Theoretical Computer Science*, vol.17(3), pp.279–301, 1982.

[7] Dershowitz,N.: Termination Dependencies. In: *Proc. of the 6th Int. Workshop on Termination* (WST03), pp.27–30, 2003.

[8] Giesl,J., Arts,T., Ohlebusch,E.: Modular Termination Proofs for Rewriting Using Dependency Pairs. *Journal of Symbolic Computation*, 34(1), pp.21–58, 2002.

[9] Giesl,J., Thiemann,R., Schneider-Kamp,P., Falke,S.: Improving Dependency Pairs. In: *Proc. of the 10th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning*, LNAI 2850 (LPAR2003), pp.165–179, 2003.

[10] Giesl,J., Thiemann,R., Schneider-Kamp,P., Falke,S.: Automated Termination Proofs with AProVE. In: *Proc. of the 15th Int. Conf. on Rewriting Techniques and Applications*, LNCS 3091 (RTA2004), pp.210-220, 2004.

[11] Giesl,J., Thiemann,R., Schneider-Kamp,P.: The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In: *Proc. of the 11th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, LNCS 3452 (LPAR2004), pp.301–331, 2005.

[12] Giesl,J., Thiemann,R., Schneider-Kamp,P.: Proving and Disproving Termination of Higher-Order Functions. In: *Proc. the 5th Int. Workshop on Frontiers of Combining Systems* LNAI 3717 (FroCoS'05), pp.216–231, 2005.

[13] Girard,J.-Y.: Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. *Ph.D. thesis, University of Paris VII*, 1972.

[3] After submission, we have succeeded in enhancing the notion of usable rules onto STRSs [31].

[14] Goguen,J.A.: Higher-Order Functions Considered Unnecessary for Higher-Order Programming. In D.A.Truner, editor, *Research Topics in Functional Programming* Addison-Wesley Longman Publishing, pp.309–351, 1990.

[15] Hindley,J.R., Seldin,J.P.: Introduction to Combinators and λ-Calculus. *Cambridge Univ. Press*, 1986.

[16] Hirokawa,N., Middeldorp,A.: Dependency Pairs Revisited. In: *Proc. of the 15th Int. Conf. on Rewriting Techniques and Applications*, LNCS 3091 (RTA04), pp.249–268, 2004.

[17] Jouannaud,J.-P., Rubio,A.: The Higher-Order Recursive Path Ordering. In: *Proc. of 14th Annual IEEE Symposium on Logic in Computer Science*, IEEE Comp. Sc. Press, pp.402–411.

[18] Kennaway,R., Klop,J.W., Sleep,M.R., de Vries,F.J.: Comparing Curried and Uncurried Rewriting. *Journal of Symbolic Computation*, 21(1), pp.15–39, 1996.

[19] Kusakari,K., Nakamura,M., Toyama,Y.: Argument Filtering Transformation. In *Proc. of Int. Conf. on Principles and Practice of Declarative Programming*, LNCS 1702 (PPDP'99), pp.47–61, 1999.

[20] Kusakari,K.: 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.

[21] Kusakari,K.: Higher-Order Path Orders based on Computability. *IEICE Transactions on Information and Systems*, Vol.E87-D, No.2, pp.352–359, 2004.

[22] Kusakari,K., Sakai,M., Sakabe,T.: Primitive Inductive Theorems Bridge Implicit Induction Methods and Inductive Theorems in Higher-Order Rewriting. *IEICE Transactions on Information and Systems*, Vol.E88-D, No.12, pp.2715–2726, 2005.

[23] Kusakari,K., Chiba,Y.: A Higher-Order Knuth-Bendix Procedure and its Applications. *IEICE Transactions on Information and Systems*, Vol.E90-D, No.4, pp.707–715, 2007.

[24] Lee,C.S., Jones,N.D., Ben-Amram,A.M.: The Size-Change Principle for Program Termination. In: *Proc. of the 28th ACM Sympo. on Principles of Programming Languages* (POPL2001), pp.81–92, 2001.

[25] Marché,C., Urbain,X.: Modular and Incremental Proofs of AC-termination. *Journal of Symbolic Computation*, 38:873–897, 2004.

[26] Middeldorp,A.: Approximating Dependency Graphs using Tree Automata Techniques. In: *Proc. of the Int. Joint Conf. on Automated Reasoning*, LNAI 2083 (IJCAR01), pp.593–610, 2001.

[27] Van Raamsdonk,F.: On Termination of Higher-Order Rewriting. In *Proc. of 12th Int. Conf. on Rewriting Techniques and Applications*, LNCS 2051 (RTA2001), pp.261–275, 2001.

[28] Reynolds,J.: Definitional Interpreters for Higher-Order Programming Languages. *Higher Order Symbolic Computation*, 11(4), pp.363–397, 1998. Reprinted from the Proc. of the 25th ACM National Comference, 1972.

[29] Sakai,M., Watanabe,Y., Sakabe,T.: Dependency Pair Method for Proving Termination of Higher-Order Rewrite Systems. *IEICE Transactions on Information and Systems*, Vol.E84-D, No.8, pp.1025–1032, 2001.

[30] Sakai,M., Kusakari,K.: On Dependency Pair Method for Proving Termination of Higher-Order Rewrite Systems. *IEICE Transactions on Information and Systems*, Vol.E88-D, No.3, pp.583–593, 2005.

[31] Sakurai,T., Kusakari,K., Sakai,M., Sakabe,T., Nishida,N.: Usable Rules and Labeling Product-Typed Terms for Dependency Pair Method in Simply-Typed Term Rewriting Systems. *IEICE Transactions on Information and Systems*, Vol.J90-D, No.4, pp.978–989, 2007.

[32] Sakurai,T., Kusakari,K., Nishida,N., Sakai,M., Sakabe,T.: Proving Sufficient Completeness of Functional Programs based on Recursive Structure Analysis and Strong Computability. In *Proc. of the Forum on Information Technology 2005 (FIT2005), Information Technology Letters*, LA-001, pp.1–4, 2005 (in Japanese).

[33] Tait,T.T.: Intensional Interpretation of Functionals of Finite Type. *Journal of Symbolic Logic* 32, pp.198–212, 1967.

[34] Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, Vol.55, Cambridge University Press, 2003.

[35] Thiemann,R., Giesl,J., Schneider-Kamp,P.: Improved Modular Termination Proofs Using Dependency Pairs. In: *Proc. of the 2nd Int. Joint Conf. on Automated Reasoning*, LNAI 3097 (IJCAR2004), pp.75–90, 2004.

[36] Thiemann,R., Giesl,J.: The Size-Change Principle and Dependency Pairs for Termination of Term Rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 16(4):229-270, 2005.

[37] Toyama,Y.: Termination of S-expression rewriting systems: Lexicographic Path Ordering for Higher-Order Terms. In: *Proc. of the 15th Int. Conf. on Rewriting Techniques and Applications*, LNCS 3091 (RTA2004), pp.40–54, 2004.

[38] Ullman, Jeffrey D.: Elements of ML Programming. Prentice Hall, 1997.

[39] Urbain,X., Modular & Incremental Automated Termination Proofs. *Journal of Automated Reasoning*, 32(4) pp 315–355, 2004.