PAPER

Static Dependency Pair Method Based on Strong Computability for Higher-Order Rewrite Systems

Keiichirou KUSAKARI[†a)], ***Member***, **Yasuo ISOGAI**[†b)], ***Nonmember***, **Masahiko SAKAI**[†c)], ***Member***, ***and*** **Frédéric BLANQUI**[††d)], ***Nonmember***

SUMMARY Higher-order rewrite systems (HRSs) and simply-typed term rewriting systems (STRSs) are computational models of functional programs. We recently proposed an extremely powerful method, the static dependency pair method, which is based on the notion of strong computability, in order to prove termination in STRSs. In this paper, we extend the method to HRSs. Since HRSs include λ-abstraction but STRSs do not, we restructure the static dependency pair method to allow λ-abstraction, and show that the static dependency pair method also works well on HRSs without new restrictions.

key words: *higher-order rewrite system, termination, static dependency pair, plain function-passing, strong computability, subterm criterion*

1. Introduction

A term rewriting system (TRS) is a computational model that provides operational semantics for functional programs [22]. A TRS cannot, however, directly handle higher-order functions, which are widely used in functional programming languages. Simply-typed term rewriting systems (STRSs) [12] and higher-order rewrite systems (HRSs) [17] have been introduced to extend TRSs. These rewriting systems can directly handle higher-order functions. For example, a typical higher-order function `foldl` can be represented by the following HRS $R_{\mathtt{foldl}}$:

$$\left\{\begin{array}{l}\mathtt{foldl}(\lambda xy.F(x,y), X, \mathtt{nil}) \to X\\ \mathtt{foldl}(\lambda xy.F(x,y), X, \mathtt{cons}(Y,L))\\ \qquad\qquad \to \mathtt{foldl}(\lambda xy.F(x,y), F(X,Y), L)\end{array}\right.$$

HRSs can represent anonymous functions because HRSs have a λ-abstraction syntax, which STRSs do not. For instance, an anonymous function $\lambda xy.\mathtt{add}(x,\mathtt{mul}(y,y))$ is used in the HRS $R_{\mathtt{sqsum}}$, which is the union of $R_{\mathtt{foldl}}$ and the following rules:

$$\left\{\begin{array}{l}\mathtt{add}(0,Y) \to Y\\ \mathtt{add}(\mathtt{s}(X),Y) \to \mathtt{s}(\mathtt{add}(X,Y))\\ \mathtt{mul}(0,Y) \to 0\\ \mathtt{mul}(\mathtt{s}(X),Y) \to \mathtt{add}(\mathtt{mul}(X,Y),Y)\\ \mathtt{sqsum}(L) \to \mathtt{foldl}(\lambda xy.\mathtt{add}(x,\mathtt{mul}(y,y)),0,L)\end{array}\right.$$

Here, the function `sqsum` returns the square sum $x_1^2 + x_2^2 + \cdots + x_n^2$ from an input list $[x_1, x_2, \ldots, x_n]$.

As a method for proving termination of TRSs, Arts and Giesl proposed the dependency pair method for TRSs based on recursive structure analysis [1], which was then extended to STRSs [12], and to HRSs [18].

In higher-order settings, there are two kinds of analysis for recursive structures. One is dynamic analysis, and the other is static analysis. The extensions in [12] and [18] analyze dynamic recursive structures based on function-call dependency relationships, but not on relationships that may be extracted syntactically from function definitions. When a program runs, some functions can be substituted for higher-order variables. Dynamic recursive structure analysis considers dependencies through higher-order variables. Static recursive structure analysis on the other hand, does not consider such dependencies.

For example, consider the HRS $R_{\mathtt{sqsum}}$. The dynamic dependency pair method in [18] extracts the following 9 pairs, called dynamic dependency pairs:

$$\left\{\begin{array}{ll}\mathtt{foldl}^\sharp(\lambda xy.F(x,y), X, \mathtt{cons}(Y,L))\\ \qquad \to \mathtt{foldl}^\sharp(\lambda xy.F(x,y), F(X,Y), L) & (a)\\ \mathtt{foldl}^\sharp(\lambda xy.F(x,y), X, \mathtt{cons}(Y,L)) \to F(c_x,c_y) & (b)\\ \mathtt{foldl}^\sharp(\lambda xy.F(x,y), X, \mathtt{cons}(Y,L)) \to F(X,Y) & (c)\\ \mathtt{add}^\sharp(\mathtt{s}(X),Y) \to \mathtt{add}^\sharp(X,Y) & (d)\\ \mathtt{mul}^\sharp(\mathtt{s}(X),Y) \to \mathtt{add}^\sharp(\mathtt{mul}(X,Y),Y) & (e)\\ \mathtt{mul}^\sharp(\mathtt{s}(X),Y) \to \mathtt{mul}^\sharp(X,Y) & (f)\\ \mathtt{sqsum}^\sharp(L) \to \mathtt{foldl}^\sharp(\lambda xy.\mathtt{add}(x,\mathtt{mul}(y,y)),0,L) & (g)\\ \mathtt{sqsum}^\sharp(L) \to \mathtt{add}^\sharp(c_x,\mathtt{mul}(c_y,c_y)) & (h)\\ \mathtt{sqsum}^\sharp(L) \to \mathtt{mul}^\sharp(c_y,c_y) & (i)\end{array}\right.$$

Here c_x, c_y are fresh constants corresponding to the bound variables x and y. The dynamic dependency pair method returns the following 15 components, called dynamic recursion components:

$$\left\{\begin{array}{l}\{(a)\},\{(b)\},\{(c)\},\{(d)\},\{(f)\},\{(a),(b)\},\\ \{(a),(c)\},\{(b),(c)\},\{(b),(g)\},\{(c),(g)\},\\ \{(a),(b),(c)\},\{(a),(b),(g)\},\{(a),(c),(g)\},\\ \{(b),(c),(g)\},\{(a),(b),(c),(g)\}\end{array}\right\}$$

It is intuitive that this recursive structure analysis may be unnatural and intractable. The problem is caused by function-call dependency relationships through the higher-order variable F.

Manuscript received October 28, 2008.
Manuscript revised May 19, 2009.

[†]The authors are with the Graduate School of Information Science, Nagoya Univ., Nagoya-shi, 464–8601 Japan.
[††]The author is with INRIA & LORIA, France.
a) E-mail: kusakari@is.nagoya-u.ac.jp
b) E-mail: isogai@trs.cm.is.nagoya-u.ac.jp
c) E-mail: sakai@is.nagoya-u.ac.jp
d) E-mail: frederic.blanqui@inria.fr

DOI: 10.1587/transinf.E92.D.2007

The static dependency pair method, which is based on definition dependency relationships, can solve the unnatural and intractable problem above. Since the static dependency pair method can ignore terms headed by a higher-order variable which are difficult to handle, in this meaning the static dependency pair method is more natural and more powerful than the dynamic dependency pair method. In fact, the static dependency pair method presented in this paper shows that $R_{\mathtt{sqsum}}$ only has the following 3 static recursion components:

$$\begin{cases} \mathtt{foldl}^\sharp(\lambda xy.F(x,y), X, \mathtt{cons}(Y,L)) \\ \qquad\qquad \to \mathtt{foldl}^\sharp(\lambda xy.F(x,y), F(X,Y), L) \end{cases}$$
$$\begin{cases} \mathtt{add}^\sharp(\mathtt{s}(X), Y) \to \mathtt{add}^\sharp(X, Y) \end{cases}$$
$$\begin{cases} \mathtt{mul}^\sharp(\mathtt{s}(X), Y) \to \mathtt{mul}^\sharp(X, Y) \end{cases}$$

The first result for the static dependency pair method was given by Sakai and Kusakari [19]. However, this result demanded that target HRSs be either 'strongly linear' or 'non-nested', which is a very strong restriction. By reconstructing a dependency pair method based on the notion of strong computability, Kusakari and Sakai proposed the static dependency pair method for STRSs and showed that the method is sound for plain function-passing STRSs [13]. Note that strong computability was introduced for proving termination in typed λ-calculus, which is a stronger condition than the property of termination [7], [21]. 'Plain function-passing' means that every higher-order variable occurs in an argument position on the left-hand side. Since many non-artificial functional programs are plain function-passing, this method has a general versatility. In this paper, we extend the static dependency pair method and the notion of plain function-passing to HRSs. Since the difference between STRSs and HRSs is the existence of anonymous functions (i.e. λ-abstraction), extension is necessary. We show that our static dependency pair method works well on plain function-passing HRSs without new restrictions.

When proving termination by dependency pair methods, non-loopingness should be shown for each recursion component. The notion of the subterm criterion [8] is frequently utilized, as is that of a reduction pair [11], which is an abstraction of the weak-reduction order [1]. The subterm criterion was slightly improved by extending the subterms permitted by the criterion [13]. Since the subterm criterion and reduction pairs are effective in termination proofs, we also reformulate these notions for HRSs. An effective and efficient method of proving termination in plain function-passing HRSs is obtained as a result. These results can be used to prove the termination of $R_{\mathtt{sqsum}}$, which cannot be achieved with the dynamic dependency pair method in [18]. It can easily be seen that each static recursion component satisfies the subterm criterion in the underlined positions:

$$\begin{cases} \mathtt{foldl}^\sharp(\lambda xy.F(x,y), X, \underline{\mathtt{cons}(Y,L)}) \\ \qquad\qquad \to \mathtt{foldl}^\sharp(\lambda xy.F(x,y), F(X,Y), \underline{L}) \end{cases}$$
$$\begin{cases} \mathtt{add}^\sharp(\underline{\mathtt{s}(X)}, Y) \to \mathtt{add}^\sharp(\underline{X}, Y) \end{cases}$$
$$\begin{cases} \mathtt{mul}^\sharp(\underline{\mathtt{s}(X)}, Y) \to \mathtt{mul}^\sharp(\underline{X}, Y) \end{cases}$$

The termination of $R_{\mathtt{sqsum}}$ can thus be shown easily.

The remainder of this paper is organized as follows. The next section provides preliminaries required later in the paper. In Sect. 3, we introduce the notion of strong computability, which provides a theoretical rationale for the static dependency pair method. In Sect. 4, we describe the notion of plain function-passing. In Sect. 5, we present the static dependency pair method for plain function-passing HRSs, the soundness of which is guaranteed by the notion of strong computability. In Sect. 6, we introduce the notions of the reduction pair and the subterm criterion in order to prove the non-loopingness of static recursion components. Concluding remarks are presented in Sect. 7.

2. Preliminaries

In this section, we give preliminaries needed later on. We assume that the reader is familiar with notions for TRSs and HRSs [22].

The set $\mathcal{S}$ of *simple types* is generated from the set $\mathcal{B}$ of *basic types* by the type constructor $\to$. A *functional type* or a *higher-order type* is a simple type of the form $\alpha \to \beta$. We denote by $\mathcal{V}_\alpha$ the set of variables of type α, and denote by Σ_α the set of function symbols of type α. We define $\mathcal{V} = \bigcup_{\alpha\in\mathcal{S}} \mathcal{V}_\alpha$ and $\Sigma = \bigcup_{\alpha\in\mathcal{S}} \Sigma_\alpha$. We assume that the sets of variables and function symbols are disjoint. The set $\mathcal{T}^{pre}_\alpha$ of *simply-typed preterms* with simple type α is generated from sets $\mathcal{V} \cup \Sigma$ by λ-abstraction and λ-application. We denote by $t{\downarrow}$ the η-long β-normal form of a simply-typed preterm t. The set $\mathcal{T}_\alpha$ of *simply-typed terms with a simple type* α is defined as $\{t{\downarrow} \mid t \in \mathcal{T}^{pre}_\alpha\}$. We denote $type(t) = \alpha$ if $t \in T_\alpha$. We also define the set $\mathcal{T}$ of *simply-typed terms* by $\bigcup_{\alpha\in\mathcal{S}} \mathcal{T}_\alpha$, and the set $\mathcal{T}_\mathcal{B}$ of *basic typed terms* by $\bigcup_{\alpha\in\mathcal{B}} \mathcal{T}_\alpha$. We write t^α to stand for $t \in \mathcal{T}_\alpha$. Any term in η-long β-normal form is of the form $\lambda x_1 \cdots x_m.a\ t_1\ \cdots\ t_n$, where a is a variable or a function symbol. We remark that $\lambda x_1 \cdots x_m.a\ t_1\ \cdots\ t_n$ is denoted with $\lambda x_1 \cdots x_m.a(t_1, \ldots, t_n)$ or $\lambda\overline{x_m}.a(\overline{t_n})$ in short. The α-equality of terms is denoted by $\equiv$. For a simply-typed term $t \equiv \lambda\overline{x_m}.a(\overline{t_n})$, the symbol a, denoted by $top(t)$, is said to be the *top symbol* of t, and the set $\{t_1, \ldots, t_n\}$, denoted by $args(t)$, is said to be *arguments* of t. The set of free variables in t denoted by $FV(t)$. We assume for convenience that bound variables in a term are all different, and are disjoint from free variables. We define the set $Sub(t)$ of *subterms* of t by $\{t\} \cup Sub(s)$ if $t \equiv \lambda x.s$; $\{t\} \cup \bigcup_{i=1}^n Sub(t_i)$ if $t \equiv a(t_1, \ldots, t_n)$. We use $t \geq_{sub} s$ to represent $s \in Sub(t)$, and define $t >_{sub} s$ by $t \geq_{sub} s$ and $t \not\equiv s$. The set of *positions* of a term t is the set $Pos(t)$ of strings over positive integers, which is inductively defined as $Pos(\lambda x.t) = \{\varepsilon\} \cup \{1p \mid p \in Pos(t)\}$ and $Pos(a(t_1, \ldots, t_n)) = \{\varepsilon\} \cup \bigcup_{i=1}^n \{ip \mid p \in Pos(t_i)\}$. The *prefix order* $\prec$ on positions is defined by $p \prec q$ iff $pw = q$ for some w ($\neq \varepsilon$). The subterm of t at position p is denoted by $t|_p$.

A term containing a special constant $\Box_\alpha$ of type α is called a *context*, denoted by $C[\,]$. We use $C[t]$ for the term obtained from $C[\,]$ by replacing $\Box_\alpha$ with t^α. A substitution θ is a mapping from variables to terms such that $\theta(X)$ has a

same type of X for each variable X. We define $Dom(\theta) = \{X \mid X \not\equiv \theta(X)\}$. A substitution is naturally extended to a mapping from terms to terms.

A *rewrite rule* is a pair (l, r) of terms, denoted by $l \to r$, such that $top(l) \in \Sigma$, $type(l) = type(r) \in \mathcal{B}$ and $FV(l) \supseteq FV(r)$[†]. A higher-order rewrite system (HRS) is a set of rules. The *reduction relation* $\underset{R}{\to}$ of an HRS R is defined by $s \underset{R}{\to} t$ iff $s \equiv C[l\theta\downarrow]$ and $t \equiv C[r\theta\downarrow]$ for some rule $l \to r \in R$, context $C[\,]$ and substitution θ. The transitive-reflexive closure of $\underset{R}{\to}$ is denoted by $\underset{R}{\overset{*}{\to}}$.

Proposition 2.1 [15] If $s \underset{R}{\overset{*}{\to}} t$ then $s\theta\downarrow \underset{R}{\overset{*}{\to}} t\theta\downarrow$.

A term t is said to be *terminating* or *strongly normalizing* in an HRS R, denoted by $SN(R, t)$, if there is no infinite sequence of R steps starting from t. We simply denote $SN(R)$ if $SN(R, t)$ holds for any term t. We also define $\mathcal{T}_{SN}(R) = \{t \mid SN(R, t)\}$, $\mathcal{T}_{\neg SN}(R) = \mathcal{T} \setminus \mathcal{T}_{SN}(R)$, and $\mathcal{T}_{SN}^{args}(R) = \{t \mid \forall u \in args(t).SN(R, u)\}$.

All top symbols of the left-hand sides of rules in an HRS R, denoted by $\mathcal{D}_R$, are called *defined*, whereas all other function symbols, denoted by $\mathcal{C}_R$, are *constructors*. We define the *marked term* $t^\sharp$ by $a^\sharp(t_1, \ldots, t_n)$ if t has a form $a(t_1, \ldots, t_n)$ with $a \in \mathcal{D}_R$; otherwise $t^\sharp \equiv t$. Here $a^\sharp$ is called a *marked symbol*.

3. Strong Computability

In this section, we define the notion of strong computability, introduced for proving termination in typed λ-calculus, which is a stronger condition than the property of termination [7], [21]. This notion provides a theoretical rationale for the static dependency pair method.

Definition 3.1 (Strong Computability) A term t is said to be *strongly computable* in an HRS R if $SC(R, t)$ holds, which is inductively defined on simple types as follows:

- in case of $type(t) \in \mathcal{B}$, $SC(R, t)$ is defined as $SN(R, t)$,
- in case of $type(t) = \alpha \to \beta$, $SC(R, t)$ is defined as $\forall u \in \mathcal{T}_\alpha.(SC(R, u) \Rightarrow SC(R, (tu)\downarrow))$.

We also define $\mathcal{T}_{SC}(R) = \{t \mid SC(R, t)\}$, $\mathcal{T}_{\neg SC}(R) = \mathcal{T} \setminus \mathcal{T}_{SC}(R)$, and $\mathcal{T}_{SC}^{args}(R) = \{t \mid \forall u \in args(t).SC(R, u)\}$.

Here we give the basic properties for strong computability, needed later on.

Lemma 3.2 For any HRS R, the following properties hold:

(1) For any $(t_0\, t_1\, \cdots\, t_n)\downarrow \in \mathcal{T}$, if $SC(R, t_i)$ holds for all t_i, then $SC(R, (t_0\, t_1\, \cdots\, t_n)\downarrow)$.
(2) For any $t^{\alpha_1 \to \cdots \to \alpha_n \to \alpha}$, if $\neg SC(R, t)$, then there exist strongly computable terms $u_i^{\alpha_i}$ $(1 \le i \le n)$ such that $\neg SC(R, (t\, u_1\, \cdots\, u_n)\downarrow)$.
(3) $SC(R, s)$ and $s \underset{R}{\overset{*}{\to}} t$ implies $SC(R, t)$, for all s, t.
(4) The η-long β-normal form $z\downarrow$ of any variable z^α is strongly computable, for all types α.
(5) $SC(R, t^\alpha)$ implies $SN(R, t^\alpha)$, for all types α.

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

(3) We prove the claim by induction on $type(t)$. The case $type(t) \in \mathcal{B}$ is trivial. Suppose that $type(s) = type(t) = \alpha \to \beta$. Let $s \equiv \lambda x.s'$, $t \equiv \lambda x.t'$, and u^α be an arbitrary strongly computable term. Since $type(l) \in \mathcal{B}$ for every $l \to r \in R$, we have $s' \underset{R}{\overset{*}{\to}} t'$. From Proposition 2.1, we have $(su)\downarrow \equiv s'\{x := u\} \underset{R}{\overset{*}{\to}} t'\{x := u\} \equiv (tu)\downarrow$. Since $(su)\downarrow$ is strongly computable, $SC(R, (tu)\downarrow)$ follows from the induction hypothesis. Hence t is strongly computable.
(4,5) We prove claims by simultaneous induction on α. The case $\alpha \in \mathcal{B}$ is trivial. Suppose that $\alpha = \alpha_1 \to \cdots \to \alpha_n \to \beta$ and $\beta \in \mathcal{B}$.
Induction step of (4): Assume that $z\downarrow$ is not strongly computable for some $z \in \mathcal{V}_\alpha$. From (2), there exist strongly computable terms $u_1^{\alpha_1}, \ldots, u_n^{\alpha_n}$ and $(z(u_1, \ldots, u_n))\downarrow \equiv z(u_1, \ldots, u_n)$ is not strongly computable. From the induction hypothesis (5), each u_i is terminating, hence so is $z(u_1, \ldots, u_n)$. Since $z(u_1, \ldots, u_n)$ is of basic types, $z(u_1, \ldots, u_n)$ is strongly computable. This is a contradiction.
Induction step of (5): From the induction hypothesis (4), $y\downarrow$ is strongly computable for any $y \in \mathcal{V}_{\alpha_1}$, hence so is $(ty)\downarrow$. From the induction hypothesis (5), $(ty)\downarrow$ is terminating, hence so is t. □

4. Plain Function-Passing

The static dependency pair method defined in the next section cannot be applied to HRSs in general. For example, consider the HRS $R = \{\mathtt{foo}(\mathtt{bar}(\lambda x.F(x))) \to F(\mathtt{bar}(\lambda x.F(x)))\}$. Since the defined symbol `foo` does not occur on the right hand side, no static recursive structure exists. However, R is not terminating: $\mathtt{foo}(\mathtt{bar}(\lambda x.\mathtt{foo}(x))) \underset{R}{\to} \mathtt{foo}(\mathtt{bar}(\lambda x.\mathtt{foo}(x))) \underset{R}{\to} \cdots$. The static dependency pair method therefore requires a suitable restriction. In [19], we introduced the notions of 'strongly linear' and 'non-nested' HRSs. However, these restrictions are too tight. For STRSs we presented the notion of plain function-passing, which covers practical level programs [13]. Intuitively, plain function-passing means that higher-order free variables on the left-hand side are passed to the right-hand side directly. In this section, we extend the notion of plain function-passing to HRSs.

Definition 4.1 Let R be an HRS and $l \to r \in R$. We define the set $safe(l)$ of *safe subterms* of l as the following:

$$args(l) \cup \bigcup_{l' \in args(l)} \{u \in safe_{\mathcal{B}}(l', FV(l)) \mid FV(l) \supseteq FV(u)\},$$

where $safe_{\mathcal{B}}(\lambda\overline{x_m}.a(\overline{t_n}), X)$ is defined as $\{a(\overline{t_n})\}$ if $a \in X$; otherwise $\{a(\overline{t_n})\} \cup \bigcup_{i=1}^{n} safe_{\mathcal{B}}(t_i, X)$.

[†]In order to guarantee the decidability of higher-order pattern-matching, Nipkow restricts rewrite rules by the notion of pattern [17]. Such a restriction, however, is not necessary to our study.

We note that $safe(l) \subseteq Sub(l)$ and any $t \in safe_{\mathcal{B}}(l', FV(l))$ is of basic types.

Example 4.2 Consider HRS $R_{\mathtt{foldl}}$ displayed in the introduction. Suppose that

$$l \equiv \mathtt{foldl}(\lambda xy.F(x,y), Y, \mathtt{cons}(X, L)).$$

For each argument $u \in args(l)$, $safe_{\mathcal{B}}(u, FV(l))$ is the following:

$$\begin{aligned} safe_{\mathcal{B}}(\lambda xy.F(x,y), FV(l)) &= \{F(x,y)\} \\ safe_{\mathcal{B}}(Y, FV(l)) &= \{Y\} \\ safe_{\mathcal{B}}(\mathtt{cons}(X,L), FV(l)) &= \{\mathtt{cons}(X,L), X, L\} \end{aligned}$$

Since $FV(F(x,y)) \not\subseteq FV(l)$, safe subterms $safe(l)$ is the following:

$$\begin{aligned} safe(l) &= args(l) \cup \{Y, \mathtt{cons}(X,L), X, L\} \\ &= \{\lambda xy.F(x,y), Y, \mathtt{cons}(X,L), X, L\} \end{aligned}$$

We prepare a technical lemma to show the soundness of the static dependency pair method.

Lemma 4.3 Let R be an HRS, $l \to r \in R$ and θ be a substitution. Then $l\theta\downarrow \in \mathcal{T}^{args}_{SC}(R)$ implies $SC(R, s\theta\downarrow)$ for any $s \in safe(l)$.

Proof. The case $s \in args(l)$ is trivial because $s\theta\downarrow \in args(l\theta\downarrow)$ follows from $top(l) \in \Sigma$. Suppose that $s \in safe_{\mathcal{B}}(l', FV(l))$ and $FV(s) \subseteq FV(l)$ for some $l' \in args(l)$. Then we have $SN(R, l'\theta\downarrow)$ from Lemma 3.2(5). Since $type(s) \in \mathcal{B}$ from the definition of $safe_{\mathcal{B}}$, it suffices to show $SN(R, s\theta\downarrow)$. We prove by induction on definition of $safe_{\mathcal{B}}$ that $s \in safe_{\mathcal{B}}(t, FV(l))$ and $SN(R, t\theta\downarrow)$ implies $SN(R, s\theta\downarrow)$, for all $t \equiv \lambda x_1 \cdots x_m.a(t_1, \ldots, t_n) \in Sub(l')$.

The case $t \equiv \lambda x_1 \cdots x_m.s$ is trivial because $t\theta\downarrow \equiv \lambda x_1 \cdots x_m.(s\theta\downarrow)$. Suppose that $s \in safe_{\mathcal{B}}(t_j, FV(l))$ for some j. Without loss of generality, we can assume that $a \notin Dom(\theta)$ because $a \notin FV(l)$. Then $t\theta\downarrow \equiv \lambda\overline{x_m}.a(\overline{t_n\theta\downarrow})$. Hence, $SN(R, t_j\theta\downarrow)$ holds. From the induction hypothesis, we have $SN(R, s\theta\downarrow)$. □

Definition 4.4 (Plain Function-Passing) An HRS R is said to be *plain function-passing* (PFP) if for any $l \to r \in R$ and $Z(r_1, \ldots, r_n) \in Sub(r)$ such that $Z \in FV(r)$, there exists k ($\leq n$) such that $Z(r_1, \ldots, r_k)\downarrow \in safe(l)$. We often abbreviate plain function-passing HRS to PFP-HRS.

Example 4.5 Referencing to Example 4.2. Since $F\downarrow \equiv \lambda xy.F(x,y) \in safe(l)$, HRS $R_{\mathtt{foldl}}$ is PFP.

Example 4.6 Let R be the following non-terminating HRS:

$$\left\{ \ \mathtt{foo}(\mathtt{bar}(\lambda x.F(x))) \to F(\mathtt{bar}(\lambda x.F(x))) \right.$$

Then R is not PFP because:

$$F\downarrow \notin \{\mathtt{bar}(\lambda x.F(x))\} = safe(\mathtt{foo}(\mathtt{bar}(\lambda x.F(x)))).$$

Example 4.7 Let R be the following terminating HRS:

$$\begin{cases} \mathtt{mapfun}(\mathtt{nil}_{\mathrm{F}}, X) \to \mathtt{nil} \\ \mathtt{mapfun}(\mathtt{cons}_{\mathrm{F}}(\lambda x.F(x), L), X) \\ \qquad \to \mathtt{cons}(F(X), \mathtt{mapfun}(L, X)) \end{cases}$$

Then R is not PFP because:

$$\begin{aligned} F\downarrow &\notin \{\mathtt{cons}_{\mathrm{F}}(\lambda x.F(x), L), L, X\} \\ &= safe(\mathtt{mapfun}(\mathtt{cons}_{\mathrm{F}}(\lambda x.F(x), L), X)) \end{aligned}$$

In any PFP-HRS R, for any subterm $Z(r_1, \ldots, r_n)$ headed by a higher-order variable in the right hand side of a rule $l \to r$, there exists a prefix $Z(r_1, \ldots, r_k)$ such that $Z(r_1, \ldots, r_k)\downarrow \in safe(l)$. Thanks to Lemmas 3.2(1) and 4.3, this property guarantees that $Z(r_1, \ldots, r_n)\theta\downarrow$ is strongly computable whenever $l\theta\downarrow \in \mathcal{T}^{args}_{SC}(R)$ and $r_i\theta\downarrow \in \mathcal{T}_{SC}(R)$ ($i = 1, \ldots, n$). This beneficial property eliminates a dependency analysis through higher-order variables from static recursive structure analysis (cf. Lemma 5.11), and contributes in obtaining the soundness of the static dependency pair method (cf. Theorem 5.12).

In the definition of PFP, the case $n = 0$ must be considered. That is, any first-order variable in $Var(r)$ should belong to $safe(l)$. Otherwise Lemma 4.3 does not hold. For example, consider the HRS $R = \{\mathtt{foo}(F(X)) \to X\}$ and the substitution $\theta = \{F := \lambda x.0\}$. Then X does not occur in $\mathtt{foo}(0) \equiv \mathtt{foo}(F(X))\theta\downarrow$, and we must exclude R from plain function-passing.

Note that every first-order rewrite system is plain function-passing.

A termination condition for higher-order rewrite rules having a specific form of plain function-passing was investigated under Jouannaud and Okada's general schema [9], [10]. The restriction that higher-order variables occur as arguments is weakened by using the notion of computability closure [3]–[5]. We leave a similar extension of the present work with computability closure for the future.

5. Static Dependency Pair Method

In this section we present the static dependency pair method for PFP-HRSs. The recursive structures derived by the static dependency pair method accord with a programmer's intuition. Since many existing programs are written so as to terminate, this method is of benefit in proving that they do indeed terminate.

First, we describe candidate terms, improving on the notion of candidate terms in [18]. Candidate terms are a variant of subterms, and bound variables never become free in candidate terms. This feature is useful for showing the soundness of our method (cf. Lemma 5.11).

Definition 5.1 (Candidate Term) The set of *candidate terms* of $t \equiv \lambda\overline{x_m}.\ a(\overline{t_n})$, denoted by $Cand(t)$, is defined as follows:

$$Cand(t) = \{t\} \cup \bigcup_{i=1}^{n} Cand(\lambda x_1 \cdots x_m.t_i)$$

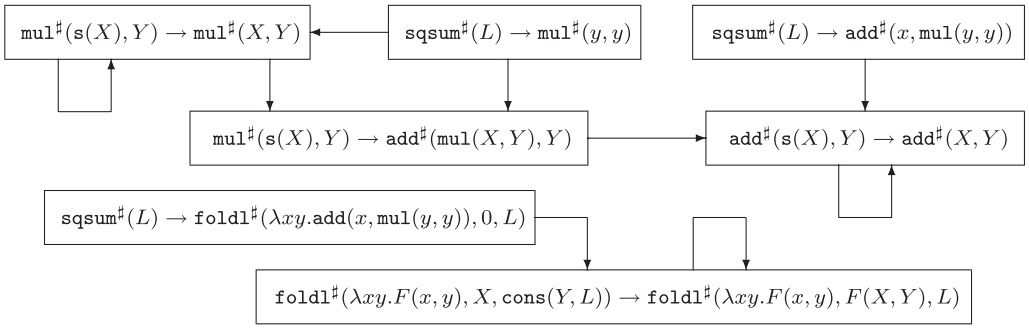

Fig. 1 Static dependency graph of $R_{\mathtt{sqsum}}$.

We consider the case of $\mathtt{foo},\mathtt{bar} \in \mathcal{D}_R$ and $t \equiv \lambda x.\mathtt{foo}(\mathtt{bar},x)$. Then we have

$$Cand(t) = \{\lambda x.\mathtt{foo}(\mathtt{bar},x), \lambda x.\mathtt{bar}, \lambda x.x\}.$$

Note that the definition in [18] gave $Cand(t) = \{\mathtt{foo}(\mathtt{bar},c_x),\mathtt{bar}\}$, where c_x is a fresh constant corresponding to the bound variable x.

Next, we introduce the notion of static dependency pairs by using candidate terms. This notion forms the basis for the static dependency pair method.

Definition 5.2 (Static Dependency Pair) Let R be an HRS. A pair $\langle l^\sharp, a^\sharp(r_1,\dots,r_n)\rangle$, denoted by $l^\sharp \to a^\sharp(r_1,\dots,r_n)$, is said to be a *static dependency pair* in R if there exists $l \to r \in R$ such that

- $\lambda x_1 \cdots x_m.a(r_1,\dots,r_n) \in Cand(r)$,
- $a \in \mathcal{D}_R$, and
- $a(r_1,\dots,r_k)\downarrow \notin safe(l)$ for all $k\ (\le n)$.

We denote by $SDP(R)$ the set of static dependency pairs in R.

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

Example 5.3 For the HRS $R_{\mathtt{sqsum}}$ displayed in the introduction, the set $SDP(R_{\mathtt{sqsum}})$ consists of the following seven pairs:

$$\begin{cases} \mathtt{foldl}^\sharp(\lambda xy.F(x,y),X,\mathtt{cons}(Y,L)) \\ \qquad\qquad \to \mathtt{foldl}^\sharp(\lambda xy.F(x,y),F(X,Y),L) \\ \mathtt{add}^\sharp(\mathtt{s}(X),Y) \to \mathtt{add}^\sharp(X,Y) \\ \mathtt{mul}^\sharp(\mathtt{s}(X),Y) \to \mathtt{add}^\sharp(\mathtt{mul}(X,Y),Y) \\ \mathtt{mul}^\sharp(\mathtt{s}(X),Y) \to \mathtt{mul}^\sharp(X,Y) \\ \mathtt{sqsum}^\sharp(L) \to \mathtt{foldl}^\sharp(\lambda xy.\mathtt{add}(x,\mathtt{mul}(y,y)),0,L) \\ \mathtt{sqsum}^\sharp(L) \to \mathtt{add}^\sharp(x,\mathtt{mul}(y,y)) \\ \mathtt{sqsum}^\sharp(L) \to \mathtt{mul}^\sharp(y,y) \end{cases}$$

Notice that we use the extra variables x,y in the sixth and seventh dependency pairs.

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 chain and a static dependency graph.

Definition 5.4 (Static Dependency Chain) Let R be an HRS. A sequence $u_0^\sharp \to v_0^\sharp, u_1^\sharp \to v_1^\sharp, \cdots$ of static dependency pairs in R is said to be a *static dependency chain in* R if there exist $\theta_0,\theta_1,\dots$ such that $v_i^\sharp\theta_i\downarrow \xrightarrow[R]{*} u_{i+1}^\sharp\theta_{i+1}\downarrow$ and $u_i\theta_i\downarrow, v_i\theta_i\downarrow \in \mathcal{T}_{SC}^{args}(R)$ for any i.

Definition 5.5 (Static Dependency Graph) The *static dependency graph of* R is a directed graph, in which nodes are $SDP(R)$ and there exists an arc from $u^\sharp \to v^\sharp$ to $u'^\sharp \to v'^\sharp$ if $u^\sharp \to v^\sharp, u'^\sharp \to v'^\sharp$ is a static dependency chain.

Example 5.6 The static dependency graph of the HRS $R_{\mathtt{sqsum}}$ (cf. Example 5.3) is shown in Fig. 1.

Unfortunately, the connectability of the static dependency pairs is undecidable. Hence, we need suitable approximation techniques. In TRSs, such techniques were studied [16]. One of simple approximated dependency graphs is the graph in which an arc from $u^\sharp \to v^\sharp$ to $u'^\sharp \to v'^\sharp$ exists if $v^\sharp$ and $u'^\sharp$ have the same top symbol. Note that for the HRS $R_{\mathtt{sqsum}}$ this approximation gives the precise static dependency graph shown in Fig. 1.

We now introduce the notions of static recursion components and non-loopingness. As usual, the termination of HRS can be proved by proving the non-loopingness of each recursion component. These proofs are similar to the other dependency pair methods.

Definition 5.7 (Static Recursion Component) Let R be an HRS. A *static recursion component* in R is a set of nodes in a strongly connected subgraph of the static dependency graph of R. Using $SRC(R)$ we denote the set of static recursion components in R.

Example 5.8 The static dependency graph of $R_{\mathtt{sqsum}}$ (Fig. 1) has three strongly connected subgraphs. Thus, the set $SRC(R_{\mathtt{sqsum}})$ consists of the following three components:

$$\begin{cases} \mathtt{foldl}^\sharp(\lambda xy.F(x,y), X, \mathtt{cons}(Y,L)) \\ \qquad\qquad \to \mathtt{foldl}^\sharp(\lambda xy.F(x,y), F(X,Y), L) \end{cases}$$
$$\begin{cases} \mathtt{add}^\sharp(\mathtt{s}(X), Y) \to \mathtt{add}^\sharp(X,Y) \end{cases}$$
$$\begin{cases} \mathtt{mul}^\sharp(\mathtt{s}(X), Y) \to \mathtt{mul}^\sharp(X,Y) \end{cases}$$

Definition 5.9 (Non-Looping) A static recursion component C in an HRS R is said to be *non-looping* if there exists no infinite static dependency chain in which only pairs in C occur and every $u^\sharp \to v^\sharp \in C$ occurs infinitely many times.

In the remainder of this section, we show the soundness of the static dependency pair method on PFP-HRSs. That is, we show that if any static recursion component of PFP-HRS R is non-looping, then R is terminating. We need two lemmas.

Lemma 5.10 Let R be a non-terminating HRS. Then $\mathcal{T}_\mathcal{B} \cap \mathcal{T}_{\neg SC}(R) \cap \mathcal{T}_{SC}^{args}(R) \neq \emptyset$.

Proof. Since R is not terminating, $\mathcal{T}_{\neg SC}(R) \neq \emptyset$ follows from Lemma 3.2(5). Let $t \equiv \lambda x_1 \cdots x_m.a(t_1, \ldots, t_n)$ be a minimal size term in $\mathcal{T}_{\neg SC}(R)$. From Lemma 3.2(2), there exist $u_1, \ldots, u_m \in \mathcal{T}_{SC}(R)$ such that $\neg SC(R, t')$ where $t' \equiv (t\ u_1\ \cdots\ u_m)\downarrow$. Suppose that $\sigma = \{x_j := u_j \mid 1 \leq j \leq m\}$. Then $t' \equiv (a\sigma\ t_1\sigma\ \cdots\ t_n\sigma)\downarrow$. Since the size of $t'_i \equiv \lambda x_1 \cdots x_m.t_i$ is less than the size of t, we have $SC(R, t'_i)$ by the minimality of t. Since $t_i\sigma\downarrow \equiv (t'_i\ u_1\ \cdots\ u_m)\downarrow$, we have $SC(R, t_i\sigma\downarrow)$ by Lemma 3.2(1). Assume that $a \in \{x_1, \ldots, x_m\}$. Since $a\sigma\downarrow \equiv u_j \in \mathcal{T}_{SC}(R)$, $SC(R, t')$ follows from Lemma 3.2(1). This is a contradiction. Hence, we have $a \notin \{x_1, \ldots, x_m\}$. Therefore we have $t' \equiv a(t_1\sigma\downarrow, \ldots, t_n\sigma\downarrow) \in \mathcal{T}_\mathcal{B} \cap \mathcal{T}_{\neg SC}(R) \cap \mathcal{T}_{SC}^{args}(R)$. □

Lemma 5.11 Let R be a PFP-HRS. For any $t \in \mathcal{T}_\mathcal{B} \cap \mathcal{T}_{\neg SC}(R) \cap \mathcal{T}_{SC}^{args}(R)$, there exist $l^\sharp \to v^\sharp \in SDP(R)$ and a substitution θ such that $t^\sharp \xrightarrow[R]{*} (l\theta\downarrow)^\sharp$ and $l\theta\downarrow, v\theta\downarrow \in \mathcal{T}_\mathcal{B} \cap \mathcal{T}_{\neg SC}(R) \cap \mathcal{T}_{SC}^{args}(R)$.

Proof. From $t \in \mathcal{T}_{SC}^{args}(R)$ and Lemma 3.2(5), we have $t \in \mathcal{T}_{SN}^{args}(R)$. From $t \in \mathcal{T}_\mathcal{B} \cap \mathcal{T}_{\neg SC}(R)$, we have $\neg SN(R, t)$. Hence, there exist $l \to r \in R$ and a substitution θ' such that $t^\sharp \xrightarrow[R]{*} (l\theta'\downarrow)^\sharp$, $l\theta'\downarrow, r\theta'\downarrow \in \mathcal{T}_{\neg SN}(R)$, and $Dom(\theta') \subseteq FV(l)$. Since $type(l) = type(r) \in \mathcal{B}$, we have $l\theta'\downarrow, r\theta'\downarrow \in \mathcal{T}_{\neg SC}(R)$. Moreover, $l\theta'\downarrow \in \mathcal{T}_{SC}^{args}(R)$ follows from Lemma 3.2(3). Since $r \in Cand(r)$ and $\neg SC(R, r\theta'\downarrow)$, we have $\{r' \in Cand(r) \mid \neg SC(R, r'\theta'\downarrow)\} \neq \emptyset$. Let $v' \equiv \lambda x_1 \cdots x_m.a(r_1, \ldots, r_n)$ be a minimal size term in this set.

From Lemma 3.2(2), there exist strongly computable terms $u_1, \ldots, u_m$ such that $(v'\theta'\ u_1\ \cdots\ u_m)\downarrow$ is not strongly computable. Let v and θ be $v \equiv a(r_1, \ldots, r_n)$ and $\theta = \theta' \cup \{x_i := u_i \mid 1 \leq i \leq m\}$. Since $v\theta\downarrow \equiv (v'\theta'\ u_1\ \cdots\ u_m)\downarrow$, we have $v\theta\downarrow \in \mathcal{T}_\mathcal{B} \cap \mathcal{T}_{\neg SC}(R)$. Since $l\theta\downarrow \equiv l\theta'\downarrow$ from $x_i \notin FV(l)$, we have $l\theta\downarrow \in \mathcal{T}_\mathcal{B} \cap \mathcal{T}_{\neg SC}(R) \cap \mathcal{T}_{SC}^{args}(R)$. Since $\lambda x_1 \cdots x_m.r_i \in Cand(r)$, $SC(R, (\lambda x_1 \cdots x_m.r_i)\theta'\downarrow)$ follows from the minimality of v'. Hence, each $r_i\theta\downarrow \equiv ((\lambda x_1 \cdots x_m.r_i)\theta'\ u_1\ \cdots\ u_m)\downarrow$ is strongly computable from Lemma 3.2(1).

We prove the remaining claims that $v\theta\downarrow \in \mathcal{T}_{SC}^{args}(R)$ and $l^\sharp \to v^\sharp \in SDP(R)$.

- Assume that $a \in \{x_i \mid 1 \leq i \leq m\}$. Then $SC(R, v\theta\downarrow)$ follows from $SC(R, a\theta\downarrow)$ and Lemma 3.2(1). This is a contradiction.
- Assume that $a \in FV(r)$. Since R is PFP, there exists k ($\leq n$) such that $a(r_1, \ldots, r_k)\downarrow \in safe(l)$. From Lemma 4.3, $SC(R, a(r_1, \ldots, r_k)\theta\downarrow)$ holds. From Lemma 3.2(1), $SC(R, v\theta\downarrow)$ holds. This is a contradiction.
- Assume that $a \in C_R$. Then $\forall i.SN(R, r_i\theta\downarrow)$ follows from Lemma 3.2(5). From $a \in C_R$, we have $SN(R, v\theta\downarrow)$. From $v \in \mathcal{T}_\mathcal{B}$, we have $SC(R, v\theta\downarrow)$. This is a contradiction.
- Assume that $a \in \mathcal{D}_R$ and there exists k ($\leq n$) such that $a(r_1, \ldots, r_k)\downarrow \in safe(l)$. From Lemma 4.3, $SC(R, a(r_1, \ldots, r_k)\theta\downarrow)$ holds. From Lemma 3.2(1), $SC(R, v\theta\downarrow)$ holds. This is a contradiction.

As shown above, we have $a \in \mathcal{D}_R$ and $a(r_1, \ldots, r_k)\downarrow \notin safe(l)$ for all k ($\leq n$). Hence $l^\sharp \to v^\sharp \in SDP(R)$. Moreover, $v\theta\downarrow \in \mathcal{T}_{SC}^{args}(R)$ holds because $v\theta\downarrow \equiv a(r_1\theta\downarrow, \ldots, r_n\theta\downarrow)$ and $SC(R, r_i\theta\downarrow)$ for any i. □

By using the two lemmas above, we can show the soundness of the static dependency pair method.

Theorem 5.12 Let R be a PFP-HRS. If there exists no infinite static dependency chain then R is terminating.

Proof. Assume that $\neg SN(R)$. From Lemma 5.10, there exists $t \in \mathcal{T}_\mathcal{B} \cap \mathcal{T}_{\neg SC}(R) \cap \mathcal{T}_{SC}^{args}(R)$. By applying Lemma 5.11 repeatedly, we obtain an infinite static dependency chain, which leads to a contradiction. □

Corollary 5.13 Let R be a PFP-HRS such that there exists no infinite path[†] in the static dependency graph. If all static recursion components are non-looping, then R is terminating.

Note that no infinite path condition in this corollary is always satisfied for finite PFP-HRSs, since nodes are finite in the static dependency graph.

6. Non-loopingness

In Sect. 5 we showed that a PFP-HRS terminates if every static recursion component is non-looping. In order to show non-loopingness, the notion of the subterm criterion [8], [13] is frequently utilized, as is that of a reduction pair [11], which is an abstraction of the weak-reduction order[††] [1]. These techniques are also effective in termination proofs for HRSs. We begin with reduction pairs.

Definition 6.1 (Reduction Pair) Let $\gtrsim$ be a quasi-order

[†]Each node cannot appear more than once in a *path*.

[††]A quasi-order $\gtrsim$ is said to be a *weak reduction order* if the pair $(\gtrsim, \gtrsim)$ of $\gtrsim$ and its strict part $\gtrsim$ is a reduction pair.

and $>$ be a strict order. The pair $(\gtrsim, >)$ is said to be a *reduction pair* if the following properties hold:

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

Lemma 6.2 Let R be an HRS and $C \in SRC(R)$. If there exists a reduction pair $(\gtrsim, >)$ such that $R \subseteq \gtrsim$, $C \subseteq \gtrsim \cup >$, and $C \cap > \neq \emptyset$, then C is non-looping.

Proof. Obvious. □

Next we introduce the subterm criterion for HRSs. In [8], Hirokawa and Middeldorp proved that the subterm criterion guarantees the non-loopingness in TRSs. The key of the proof is that the relation $\underset{R}{\rightarrow} \cup >_{sub}$ is well-founded on terminating terms. Since the property also holds in higher-order rewriting, we directly ported the criterion to STRSs [13]. We also slightly improved the subterm criterion by extending the codomain of a function π from positive integers to sequences of positive integers [13]. In the following, we extend the improved subterm criterion onto HRSs, that is to handle λ-abstraction.

Definition 6.3 (Subterm Criterion) Let R be an HRS and $C \in SRC(R)$. We say that C satisfies the *subterm criterion* if there exists a function π from $\mathcal{D}_R$ to non-empty sequences of positive integers such that

(α) $u|_{\pi(top(u))} >_{sub} v|_{\pi(top(v))}$ for some $u^\sharp \to v^\sharp \in C$, and
(β) the following conditions hold for any $u^\sharp \to v^\sharp \in C$:

- $u|_{\pi(top(u))} \geq_{sub} v|_{\pi(top(v))}$,
- $\forall p \prec \pi(top(u)).top(u|_p) \notin FV(u)$, and
- $\forall q \prec \pi(top(v)).q = \varepsilon \vee top(v|_q) \notin FV(v) \cup \mathcal{D}_R$.

Lemma 6.4 Let R be an HRS and $C \in SRC(R)$. If C satisfies the subterm criterion then C is non-looping.

Proof. Assume that pairs in C generate an infinite chain $u_0^\sharp \to v_0^\sharp, u_1^\sharp \to v_1^\sharp, u_2^\sharp \to v_2^\sharp, \cdots$ in which every $u^\sharp \to v^\sharp \in C$ occurs infinitely many times, and let $\theta_0, \theta_1, \ldots$ be substitutions such that $v_i^\sharp\theta_i\downarrow \underset{R}{\overset{*}{\rightarrow}} u_{i+1}^\sharp\theta_{i+1}\downarrow$ and $u_i\theta_i\downarrow, v_i\theta_i\downarrow \in \mathcal{T}_{SC}^{args}(R)$ for each i. From Lemma 3.2(5), $u_i\theta_i\downarrow, v_i\theta_i\downarrow \in \mathcal{T}_{SN}^{args}(R)$. Denote $\pi(top(u_i))$ by p_i for each i. Since $v_i^\sharp\theta_i\downarrow \underset{R}{\overset{*}{\rightarrow}} u_{i+1}^\sharp\theta_{i+1}\downarrow$, we have $top(v_i) = top(u_{i+1})$. Hence, from the condition (β) of the subterm criterion, we have

$$(u_0\theta_0\downarrow)|_{p_0} \geq_{sub} (v_0\theta_0\downarrow)|_{p_1} \underset{R}{\overset{*}{\rightarrow}} (u_1\theta_1\downarrow)|_{p_1} \geq_{sub} \cdots.$$

From the condition (α) of the subterm criterion, the sequence above contains infinitely many $>_{sub}$. Hence there exists an infinite sequence starting with $(u_0\theta_0\downarrow)|_j$ with respect to $\underset{R}{\rightarrow} \cup >_{sub}$, where j is the positive integer such that $j \preceq p_0$. This is a contradiction with $u_0\theta_0\downarrow \in \mathcal{T}_{SN}^{args}(R)$. □

Finally, we present a powerful method for proving termination of PFP-HRSs.

Theorem 6.5 Let R be a PFP-HRS such that there exists no infinite path in the static dependency graph. If any static recursion component $C \in SRC(R)$ satisfies one of the following properties, then R is terminating.

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

Proof. From Corollary 5.13 and Lemma 6.2, 6.4. □

As seen in the theorem, proving non-loopingness by the subterm criterion depends only on a recursion component, unlike proving one by a reduction pair. Thus the approach by the subterm criterion is more efficient than the approach by reduction pairs.

Example 6.6 We show the termination of PFP-HRS $R_{\texttt{sqsum}}$ displayed in the introduction. Let $\pi(\texttt{foldl}) = 3$, $\pi(\texttt{add}) = 1$, and $\pi(\texttt{mul}) = 1$. Then all $C \in SRC(R_{\texttt{sqsum}})$ (cf. Example 5.8) satisfy the subterm criterion in the underlined positions below:

$$\begin{cases} \texttt{foldl}^\sharp(\lambda xy.F(x,y), X, \underline{\texttt{cons}(Y,L)}) \\ \qquad\qquad \to \texttt{foldl}^\sharp(\lambda xy.F(x,y), F(X,Y), \underline{L}) \end{cases}$$
$$\begin{cases} \texttt{add}^\sharp(\underline{\texttt{s}(X)}, Y) \to \texttt{add}^\sharp(\underline{X}, Y) \end{cases}$$
$$\begin{cases} \texttt{mul}^\sharp(\underline{\texttt{s}(X)}, Y) \to \texttt{mul}^\sharp(\underline{X}, Y) \end{cases}$$

Hence the termination can be shown by Theorem 6.5.

7. Concluding Remarks

In this paper, we extended the static dependency pair method based on strong computability for STRSs [13] to that for HRSs. The following topics remain for future work.

- *Argument filtering method for HRSs*: Since it is generally difficult to design reduction pairs, the argument filtering method was proposed for the dependency pair method of TRSs [1], and extended to STRSs [12]. However, there is no known argument filtering method for HRSs. The argument filtering method in [12] can only be applied to left-firmness systems, in which every variable of the left-hand sides occurs at a leaf position. It may be possible to adapt the argument filtering method for HRSs without the left-firmness restriction because the counterexample shown in [12] is no longer a counterexample for HRSs.
- *Notion of usable rules for HRSs*: The notion of usable rules was introduced for TRSs by Hirokawa and Middeldorp [8], and by Thiemann, Giesl, and Schneider-Kamp [23] to reduce constraints when trying to prove non-loopingness by means of reduction pairs. These proofs are based on Urbain's proof of an incremental approach to the dependency pair method [24]. It will be of benefit to develop the notion of usable rules for HRSs.
- *Extending upon the class of plain function-passing*:

We have only shown the soundness of the static dependency pair method for the class of plain function-passing systems. The notions of pattern computable closure [4] and safe function-passing [14] are promising techniques by which this may be extended.

Acknowledgements

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

References

[1] T. Arts and J. Giesl, "Termination of term rewriting using dependency pairs," Theor. Comput. Sci., vol.236, pp.133–178, 2000.
[2] F. Blanqui, "Termination and confluence of higher-order rewrite systems," Proc. 11th Int. Conf. on Rewriting Techniques and Applications, LNCS 1833 (RTA2000), pp.47–61, 2000.
[3] F. Blanqui, J.-P. Jouannaud, and M. Okada, "Inductive-data-type systems," Theor. Comput. Sci., vol.272, pp.41–68, 2002.
[4] F. Blanqui, "Higher-order dependency pairs," Proc. 8th Int. Workshop on Termination (WST2006), pp.22–26, 2006.
[5] F. Blanqui, "Computability closure: Ten years later," Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday, LNCS 4600 (Rewriting, Computation and Proof), pp.68–88, 2007.
[6] N. Dershowitz, "Orderings for term-rewriting systems," Theor. Comput. Sci., vol.17, no.3, pp.270–301, 1982.
[7] J.-Y. Girard, Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. Ph.D. thesis, University of Paris VII, 1972.
[8] N. Hirokawa and A. Middeldorp, "Dependency pairs revisited," Proc. 15th Int. Conf. on Rewriting Techniques and Applications, LNCS 3091 (RTA04), pp.249–268, 2004.
[9] J.-P. Jouannaud and M. Okada, "A computation model for executable higher-order algebraic specification languages," Proc. 6th IEEE Symposium on Logic in Computer Science, pp.350–361, 1991.
[10] J.-P. Jouannaud and M. Okada, "Abstract data type systems," Theor. Comput. Sci., vol.173, no.2, pp.349–391, 1997.
[11] K. Kusakari, M. Nakamura, and Y. Toyama, "Argument filtering transformation," Proc. Int. Conf. on Principles and Practice of Declarative Programming, LNCS 1702 (PPDP'99), pp.47–61, 1999.
[12] K. Kusakari, "On proving termination of term rewriting systems with higher-order variables," IPSJ Trans. Programming, vol.42, no.SIG 7 (PRO 11), pp.35–45, 2001.
[13] K. Kusakari and M. Sakai, "Enhancing dependency pair method using strong computability in simply-typed term rewriting systems," Applicable Algebra in Engineering, Communication and Computing, vol.18, no.5, pp.407–431, 2007.
[14] K. Kusakari and M. Sakai, "Static dependency pair method for simply-typed term rewriting and related techniques," IEICE Trans. Inf. & Syst., vol.E92-D, no.2, pp.235–247, Feb. 2009.
[15] R. Mayr and N. Nipkow, "Higher-order rewrite systems and their confluence," Theor. Comput. Sci., vol.192, no.2, pp.3–29, 1998.
[16] A. Middeldorp, "Approximations for strategies and termination," Proc. 2nd Int. Workshop on Reduction Strategies in Rewriting and Programming, vol.70(6) of Electronic Notes in Theoretical Computer Science, 2002.
[17] N. Nipkow, "Higher-order critical pairs," Proc. 6th Annual IEEE Symposium on Logic in Computer Science, pp.342–349, 1991.
[18] M. Sakai, Y. Watanabe, and T. Sakabe, "An extension of the dependency pair method for proving termination of higher-order rewrite systems," IEICE Trans. Inf. & Syst., vol.E84-D, no.8, pp.1025–1032, Aug. 2001.
[19] M. Sakai and K. Kusakari, "On dependency pair method for proving termination of higher-order rewrite systems," IEICE Trans. Inf. & Syst., vol.E88-D, no.3, pp.583–593, March 2005.
[20] T. Sakurai, K. Kusakari, M. Sakai, T. Sakabe, and N. Nishida, "Usable rules and labeling product-typed terms for dependency pair method in simply-typed term rewriting systems," IEICE Trans. Inf. & Syst. (Japanese Edition), vol.J90-D, no.4, pp.978–989, April 2007.
[21] T.T. Tait, "Intensional interpretation of functionals of finite type," J. Symbolic Logic, vol.32, pp.198–212, 1967.
[22] Terese, Term Rewriting Systems, Cambridge Tracts in Theoretical Computer Science, vol.55, Cambridge University Press, 2003.
[23] R. Thiemann, J. Giesl, and P. Schneider-Kamp, "Improved modular termination proofs using dependency pairs," Proc. 2nd Int. Joint Conf. on Automated Reasoning, LNAI 3097 (IJCAR2004), pp.75–90, 2004.
[24] X. Urbain, "Modular & incremental automated termination proofs," J. Automated Reasoning, vol.32, no.4, pp.315–355, 2004.

Keiichirou Kusakari received B.E. from Tokyo Institute of Technology in 1994, received M.E. and the Ph.D. degree from Japan Advanced Institute of Science and Technology in 1996 and 2000. From 2000, he was a research associate at Tohoku University. He transferred to Nagoya University's Graduate School of Information Science in 2003 as an assistant professor and became an associate professor in 2006. His research interests include term rewriting systems, program theory, and automated theorem proving. He is a member of IPSJ and JSSST.

Yasuo Isogai received the B.E. and M.E. degrees from Nagoya University in 2006 and 2008, respectively. He engaged in research on term rewriting systems. He is going to work at Hitachi Ltd. from April 2008.

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

Frédéric Blanqui received his PhD degree in September 2001 at the University of Paris 11 (Orsay, France). He did a postdoc at Cambridge University (UK) from October 2001 to August 2002, and at Ecole Polytechnique (Palaiseau, France) from September 2002 to August 2003. Since October 2003, he is permanent full-time INRIA researcher at LORIA (Nancy, France). He is interested in rewriting theory, type theory, termination, functional programming and proof assistants. He received the Kleene Award for the best student paper at LICS'01, and the French SPECIF 2001 Award for his PhD.