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.

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 $R_{\text{foldl}}$:

\[
\begin{align*}
\text{foldl}(\lambda xy.F(x,y),X,\text{nil}) & \to X \\
\text{foldl}(\lambda xy.F(x,y),X,\text{cons}(Y,L)) & \to \text{foldl}(\lambda xy.F(x,y),F(X,Y),L)
\end{align*}
\]

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

\[
\begin{align*}
\text{add}(0,Y) & \to Y \\
\text{add}(s(X),Y) & \to s(\text{add}(X,Y)) \\
\text{mul}(0,Y) & \to 0 \\
\text{mul}(s(X),Y) & \to \text{mul}(\text{mul}(X,Y),Y) \\
\text{sqsum}(L) & \to \text{foldl}(\lambda xy.\text{add}(x,\text{mul}(y)),0,L)
\end{align*}
\]

Here, the function $\text{sqsum}$ returns the square sum $x_1^2 + x_2^2 + \ldots + x_n^2$ 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 $R_{\text{sqsum}}$. The dynamic dependency pair method in [18] extracts the following 9 pairs, called dynamic dependency pairs:

\[
\begin{align*}
\text{foldl}^2(\lambda xy.F(x,y),X,\text{cons}(Y,L)) & \to \text{foldl}^2(\lambda xy.F(x,y),F(X,Y),L) \quad (a) \\
\text{foldl}^2(\lambda xy.F(x,y),X,\text{cons}(Y,L)) & \to F(c_x,c_y) \quad (b) \\
\text{foldl}^2(\lambda xy.F(x,y),X,\text{cons}(Y,L)) & \to F(X,Y) \quad (c) \\
\text{add}^2(s(X,Y)) & \to \text{add}^2(X,Y) \quad (d) \\
\text{mul}^2(s(X,Y)) & \to \text{add}^2(\text{mul}(X,Y),Y) \quad (e) \\
\text{mul}^2(s(X,Y)) & \to \text{mul}^2(X,Y) \quad (f) \\
\text{sqsum}^2(L) & \to \text{foldl}^2(\lambda xy.\text{add}(x,\text{mul}(y)),0,L) \quad (g) \\
\text{sqsum}^2(L) & \to \text{add}^2(c_x,\text{mul}(c_y,c_y)) \quad (h) \\
\text{sqsum}^2(L) & \to \text{mul}^2(c_x,c_y) \quad (i)
\end{align*}
\]

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:

\[
\{[(a)], [(b)], [(c)], [(d)], [(f)], [(a),(b)], [(a),(c)], [(b),(c)], [(b),(g)], [(c),(g)], [(a),(b),(c)], [(a),(b),(g)], [(a),(c),(g)], [(b),(c),(g)]\}
\]

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

Copyright © 2009 The Institute of Electronics, Information and Communication Engineers
The static dependency pair method, which is based on the definition of 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 proposed in this paper shows that $R_{sqsum}$ only has the following 3 static recursion components:

$$\begin{align*}
\operatorname{foldl}^2(\lambda xy. F(x, y), X, \text{cons}(Y, L)) \\
\rightarrow \operatorname{foldl}^2(\lambda xy. F(x, y), F(X, Y), L) \\
\operatorname{add}^s(s(X), Y) \rightarrow \operatorname{add}^d(X, Y) \\
\operatorname{mul}^s(s(X), Y) \rightarrow \operatorname{mul}^d(X, Y)
\end{align*}$$

The first result for the static dependency pair method was given by Sakai and Kusakari [19]. However, this result was obtained as a result. These results can be achieved with the dynamic dependency pair method in [18].

The termination of $R_{sqsum}$ 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 $S$ of simple types is generated from the set $B$ of basic types by the type constructor $\rightarrow$. A functional type or a higher-order type is a simple type of the form $\alpha \rightarrow \beta$.

We denote by $V_\alpha$ the set of variables of type $\alpha$, and denote by $\Sigma_\alpha$ the set of function symbols of type $\alpha$. We define $V = \bigcup_{\alpha \in S} V_\alpha$ and $\Sigma = \bigcup_{\alpha \in S} \Sigma_\alpha$. We assume that the sets of variables and function symbols are disjoint. The set $T_{\alpha}^{pre}$ of simply-typed preterms with simple type $\alpha$ is generated from sets $V \cup \Sigma$ by $\lambda$-abstraction and $\lambda$-application. We denote by $t_1$ the $\eta$-long $\beta$-normal form of a simply-typed preterm $t$. The set $T_\alpha$ of simply-typed terms with a simple type $\alpha$ is defined as $\{ t_1 \mid t \in T_{\alpha}^{pre} \}$. We denote type($t$) = $\alpha$ if $t \in T_\alpha$.

We also define the set $T$ of simply-typed terms by $\bigcup_{\alpha \in S} T_\alpha$, and the set $T_B$ of basic typed terms by $\bigcup_{\alpha \in S} T_\alpha$. We write $t^*$ to stand for $t \in T_B$. Any term in $\eta$-long $\beta$-normal form is of the form $\lambda x_1 \cdots x_n. a t_1 \cdots t_n$, where $a$ is a variable or a function symbol. We remark that $\lambda x_1 \cdots x_n. a t_1 \cdots t_n$ is denoted with $\lambda x_1 \cdots x_n. a(t_1, \ldots, t_n)$ or $\lambda t_{\alpha_1}, \ldots, t_{\alpha_n}$ in short. The $\alpha$-equality of terms is denoted by $\equiv$. For a simply-typed term $t \equiv \lambda x_1 \cdots x_n. a(t_1, \ldots, t_n)$, 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 $\text{args}(t)$, 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 $\text{Sub}(t)$ of subterms of $t$ by $\{ t \} \cup \text{Sub}(s)$ if $t \equiv \lambda x.s; \{ t \} \cup \bigcup_{i=1}^n \text{Sub}(t_i)$ if $t \equiv a(t_1, \ldots, t_n)$. We use $t \supseteq_{\text{sub}} s$ to represent $s \in \text{Sub}(t)$, and define $t \supseteq_{\text{sub}} s$ by $t \supseteq_{\text{sub}} s$ and $t \neq s$. The set of positions of a term $t$ is the set $\text{Pos}(t)$ of strings over positive integers, which is inductively defined as $\text{Pos}(\lambda t) = \{ e \} \cup \{ 1p \mid p \in \text{Pos}(t) \} \cup \text{Pos}(a(t_1, \ldots, t_n)) = \{ e \} \cup \bigcup_{i=1}^n \{ ip \mid p \in \text{Pos}(t_i) \}$.

The prefix order $<_{\text{pos}}$ is defined by $p < q$ if $p \neq q$ for some $w$ ($\neq e$). The subterm of $t$ at position $p$ is denoted by $t_p$.

A term containing a special constant $\Box_\alpha$ of type $\alpha$ is called a context, denoted by $C[\,]$. We use $C[t]$ for the term obtained from $C[\,]$ by replacing $\Box_\alpha$ with $t^*$. A substitution $\theta$ is a mapping from variables to terms such that $\theta(X)$ has a
same type of $X$ for each variable $X$. We define $\text{Dom}(\theta) = \{X \mid X \notin \theta(X)\}$. A substitution is naturally extended to a mapping from terms to terms.

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

**Proposition 2.1** [15] If $s \rightarrow t$ then $s\theta\downarrow \rightarrow^* t\theta\downarrow$.

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 $T_{SN}(R) = \{t \mid SN(R, t)\}$, $T_{SN} = T \setminus T_{SN}(R)$, and $T_{\text{arg}}^{\text{SN}}(R) = \{t \mid \forall u \in \text{args}(t).SN(R, u)\}$.

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

### 3. Strong Computability

In this section, we define the notion of strong computability, introduced for proving termination in typed $\lambda$-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 $\text{type}(t) \in B$, $SC(R, t)$ is defined as $SN(R, t)$,
- in case of $\text{type}(t) = \alpha \rightarrow \beta$, $SC(R, t)$ is defined as $\forall u \in T_\alpha. (SC(R, u) \Rightarrow SC(R, (tu)\downarrow))$.

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

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 $(t_0, t_1, \ldots, t_n)\downarrow \in T$, if $SC(R, t_i)$ holds for all $t_i$, then $SC(R, (t_0, t_1, \ldots, t_n)\downarrow)$.
2. For any $t_i^{\alpha_1 \rightarrow \cdots \rightarrow \alpha_n}$, if $\neg SC(R, t)$, then there exist strongly computable terms $u_i^{\alpha_1}$ (1 ≤ $i$ ≤ $n$) such that $\neg SC(R, (tu_i^{\alpha_1} \ldots u_n^{\alpha_n})\downarrow)$.
3. $SC(R, s)$ and $s \rightarrow t$ implies $SC(R, t)$, for all $s, t$.
4. The $\eta$-long $\beta$-normal form $\eta\downarrow$ of any variable $\bar{z}$ is strongly computable, for all types $\alpha$.
5. $SC(R, \bar{r})$ implies $SN(R, \bar{r})$, for all types $\alpha$.

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

(3) We prove the claim by induction on $\text{type}(t)$. The case $\text{type}(t) \in B$ is trivial. Suppose that $\text{type}(s) = \alpha \rightarrow \beta$. Let $s \equiv \lambda x.s'\downarrow$, $t \equiv \lambda x.t'\downarrow$, and $u^\alpha$ be an arbitrary strongly computable term. Since $\text{type}(l) \in B$ for every $l \rightarrow r \in R$, we have $s' \rightarrow^* t'$. From Proposition 2.1, we have $(su)\downarrow \equiv s'[x := u] \rightarrow^* t'[x := u] \equiv (tu)\downarrow$. 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 $\alpha$. The case $\alpha \in B$ is trivial. Suppose that $\alpha = \alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow \beta$ and $\beta \in B$.

Induction step of (4): Assume that $z_\downarrow \downarrow$ 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), $y_\downarrow \downarrow$ is strongly computable for any $y \in \mathcal{V}_{\alpha_1}$, hence so is $(ty)\downarrow$. From the induction hypothesis (5), $(ty)\downarrow \downarrow$ 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 $R = \{\text{foo}(\text{bar}(\lambda x.\text{foo}(x))) \rightarrow \text{foo}(\text{bar}(\lambda x.\text{foo}(x)))\}$. Since the defined symbol foo does not occur on the right hand side, no static recursive structure exists. However, $R$ is not terminating: $\text{foo}(\text{bar}(\lambda x.\text{foo}(x))) \rightarrow^* \text{foo}(\text{bar}(\lambda x.\text{foo}(x))) \rightarrow^* \cdots$. 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 $l \rightarrow r \in R$. We define the set $\text{safe}(l)$ of safe subterms of $l$ as the following:

$$\text{args}(l) \cup \bigcup_{F \in \text{arg}(l)} \{u \in \text{safe}(F, \mathcal{F}(l)) \mid \mathcal{F}(l) \supseteq \mathcal{F}(u)\}$$

where $\text{safe}(\lambda x.a(\bar{t})\downarrow, X)$ is defined as $\{a(\bar{t})\} \cup \bigcup_{i=1}^{\alpha} \text{safe}(t_i, X)$.

1In 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 $t \in safe_2(l', FV(l))$ is of basic types.

**Example 4.2** Consider HRS $R_{ fold1}$ displayed in the introduction. Suppose that

\[
l \equiv fold1(Axy.F(x,y), Y, cons(X, L)).
\]

For each argument $u \in args(l)$, $safe_2(u, FV(l))$ is the following:

\[
safe_2(\lambda x.F(x,y), FV(l)) = \{F(x,y)\}
\]

\[
safe_2(cons(X, L), FV(l)) = \{cons(X, L), X, L\}
\]

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

\[
safe(l) = args(l) \cup \{Y, cons(X, L), X, L\}
\]

\[
= \{\lambda x.F(x,y), Y, cons(X, L), X, L\}
\]

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

**Lemma 4.3** Let $R$ be an HRS, $l \rightarrow r \in R$ and $\theta$ be a substitution. Then $\theta l_\downarrow \in T_{args}^\prime(R)$ implies $SC(R, \theta l_\downarrow)$ for any $s \in safe(l)$.

**Proof.** The case $s \in args(l)$ is trivial because $\theta l_\downarrow \in args(\theta l_\downarrow)$ follows from $top(l) \in \Sigma$. Suppose that $s \in safe_2(\theta l_\downarrow, FV(l))$ and $FV(s) \not\subseteq FV(l)$ for some $l' \in args(l)$. Then we have $S N(R, l' \theta l_\downarrow)$ from Lemma 3.2(5). Since type($s \in B$ from the definition of safe_2, it suffices to show $S N(R, \theta l_\downarrow)$. We prove by induction on definition of safe_2 that $s \in safe_2(t, FV(l))$ and $S N(R, \theta l_\downarrow)$ implies $S N(R, \theta s l_\downarrow)$ for all $t \equiv \lambda x_1 \ldots x_n.a(t_1, \ldots, t_n) \in Sub(l').$ The case $t \equiv \lambda x_1 \ldots x_n.a$ is trivial because $\theta l_\downarrow \equiv \lambda x_1 \ldots x_n.a(\theta s l_\downarrow)$. Suppose that $s \in safe_2(l', FV(l'))$ for some $j$. Without loss of generality, we can assume that $a \not\in Dom(\theta)$ because $a \not\in FV(l)$. Then $\theta s l_\downarrow \equiv \lambda x_1 \ldots x_n.a(t_1, \theta l_\downarrow)$. Hence, $S N(R, \theta s l_\downarrow)$ holds. From the induction hypothesis, we have $S N(R, \theta s l_\downarrow)$. $\square$

**Definition 4.4 (Plain Function-Passing)** An HRS $R$ is said to be plain function-passing (PFP) if for any $l \rightarrow r \in R$ 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 $F \equiv \lambda x.F(x,y) \in safe(l)$, HRS $R_{ fold1}$ is PFP.

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

\[
\{ \text{foo}(\text{bar}(\lambda x.F(x))) \rightarrow F(\text{bar}(\lambda x.F(x))) \}
\]

Then $R$ is not PFP because:

\[
F \not\subseteq \{\text{bar}(\lambda x.F(x))\} = safe(\text{foo}(\text{bar}(\lambda x.F(x)))�
\]

**Example 4.7** Let $R$ be the following terminating HRS:

\[
\begin{align*}
\text{mapfun}(\text{nil}_l, X) & \rightarrow \text{nil} \\
\text{mapfun}(\text{cons}((\lambda x.F(x), L), X)) & \rightarrow \text{cons}(F(X), \text{mapfun}(L, X))
\end{align*}
\]

Then $R$ is not PFP because:

\[
F \not\subseteq \{\text{cons}((\lambda x.F(x), L), X) \}
\]

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 $l \rightarrow r$, there exists a prefix $Z(r_1, \ldots, r_i)$ such that $Z(r_1, \ldots, r_i) \downarrow \in safe(l)$. Thanks to Lemmas 3.2(1) and 4.3, this property guarantees that $Z(r_1, \ldots, r_n) \downarrow$ is strongly computable whenever $\theta l_\downarrow \in T_{args}^\prime(R)$ and $\theta r_\downarrow \in T_{args}^\prime(R)$ ($i = 1, \ldots, n)$. 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($l$) should belong to safe($l$). Otherwise Lemma 4.3 does not hold. For example, consider the HRS $R = \{\text{foo}(F(X)) \rightarrow X\}$ and the substitution $\theta = \{F := \lambda x.0\}$. Then $X$ does not occur in $\text{foo}(0) \equiv \text{foo}(F(X))\theta l_\downarrow$, 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 $t \equiv \lambda x_1 \ldots x_n.a(t_1, \ldots, t_n)$, denoted by $Cand(t)$, is defined as follows:

\[
Cand(t) = \{t\} \cup \bigcup_{i=1}^{n} Cand(\lambda x_1 \ldots x_m.t_i)
\]
We consider the case of \( \text{foo}, \text{bar} \in D_R \) and \( t \equiv \lambda x. \text{foo}(\text{bar}, x) \). Then we have

\[
\text{Cand}(t) = \{ \lambda x. \text{foo}(\text{bar}, x), \lambda x. \text{bar}, \lambda x. x \}.
\]

Note that the definition in [18] gave \( \text{Cand}(t) = \{ \text{foo}(\text{bar}, c_i), \text{bar} \} \), where \( c_i \) 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 \( \langle \lambda^t, a^r(r_1, \ldots, r_n) \rangle \), denoted by \( l \rightarrow r \) such that

- \( \lambda x_1 \cdots x_m. a(r_1, \ldots, r_n) \in \text{Cand}(r) \),
- \( a \in D_R \), and
- \( a(r_1, \ldots, r_k) \notin \text{safe}(l) \) for all \( k \leq n \).

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 \( R_{\text{sqsum}} \) displayed in the introduction, the set \( SDP(R_{\text{sqsum}}) \) consists of the following seven pairs:

\[
\begin{align*}
\text{fold}_{\text{sq}}(\lambda x. F(x, y), X, \text{cons}(Y, L)) & \rightarrow \text{fold}_{\text{sq}}(\lambda x. F(x, y), F(X, Y), L) \\
\text{add}_{\text{sq}}(s(X, Y)) & \rightarrow \text{add}_{\text{sq}}(X, Y) \\
\text{mul}_{\text{sq}}(s(X, Y)) & \rightarrow \text{add}_{\text{sq}}(\text{mul}(X, Y), Y) \\
\text{mul}_{\text{sq}}(s(X, Y)) & \rightarrow \text{mul}_{\text{sq}}(X, Y) \\
\text{sqsum}_{\text{sq}}(L) & \rightarrow \text{fold}_{\text{sq}}(\lambda x. \text{add}(x, \text{mul}(y, y)), 0, L) \\
\text{sqsum}_{\text{sq}}(L) & \rightarrow \text{add}_{\text{sq}}(x, \text{mul}(y, y)) \\
\text{sqsum}_{\text{sq}}(L) & \rightarrow \text{mul}_{\text{sq}}(y, y)
\end{align*}
\]

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 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 \( u^0 \rightarrow v^1 \rightarrow u^1 \rightarrow v^2 \rightarrow \cdots \) of static dependency pairs in \( R \) is said to be a static dependency chain in \( R \) if there exist \( \theta_0, \theta_1, \ldots \) such that \( v_i \theta_i \downarrow \overset{\text{deg}}{\rightarrow} u_{i+1} \theta_{i+1} \downarrow \) and \( u_i \theta_i \downarrow, v_i \theta_i \downarrow \in T_{SC}^\text{args}(R) \) 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 \( u^i \rightarrow v^i \) to \( u^i \rightarrow v^i \) if \( u^i \rightarrow v^i, u^i \rightarrow v^i \) is a static dependency chain.

**Example 5.6** The static dependency graph of the HRS \( R_{\text{sqsum}} \) (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 \( u^i \rightarrow v^i \) to \( u^i \rightarrow v^i \) exists if \( u^i \rightarrow v^i, u^i \rightarrow v^i \) have the same top symbol. Note that for the HRS \( R_{\text{sqsum}} \) 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 \( R_{\text{sqsum}} \) (Fig. 1) has three strongly connected subgraphs. Thus, the set \( SRC(R_{\text{sqsum}}) \) consists of the following three components:
Lemma 5.10 Let \( R \) be a non-terminating HRS. Then \( TBG \cap T_{-SC}(R) \cap T_{SC}^{\text{arx}}(R) \neq \emptyset \).

Proof. Since \( R \) is not terminating, \( T_{-SC}(R) \neq \emptyset \) follows from Lemma 3.2(5). Let \( t = \lambda x_1 \cdots x_m a(t_1, \ldots, t_n) \) be a minimal size term in \( T_{-SC}(R) \). From Lemma 3.2(2), there exist \( u_1, \ldots, u_m \in T_{SC}(R) \) such that \( \neg SC(R, t') \) where \( t' = (t u_1 \cdots u_m) \). Suppose that \( \sigma = \{ x_i : := u_i \mid 1 \leq j \leq m \} \). Then \( t' = (\sigma a t_1 \sigma \cdots t_n \sigma) \). Since the size of \( t' \) is less than the size of \( t \), we have \( SC(R, t') \) by the minimality of \( t \). Since \( t \sigma \sigma \in \) \( T_{SG} \cap T_{-SC}(R) \) by Lemma 3.2(1). Assume that \( a \in \{ x_1, \ldots, x_m \} \). Since \( \alpha a \sigma \sigma \subseteq u_1 \in T_{SC}(R) \), \( SC(R, t') \) follows from Lemma 3.2(1). This is a contradiction. Hence, we have \( a \notin \{ x_1, \ldots, x_m \} \). Therefore we have \( t' = a(t_1 \sigma \sigma, \ldots, t_n \sigma \sigma) \in T_{SG} \cap T_{-SC}(R) \cap T_{SC}^{\text{arx}}(R) \). \( \square \)

Lemma 5.11 Let \( R \) be a PFP-HRS. For any \( t \in T_{SB} \cap T_{-SC}(R) \cap T_{SC}^{\text{arx}}(R) \), there exist \( \bar{t} \to \bar{v} \in SDP(R) \) and a substitution \( \theta \) such that \( \bar{t} \overset{\theta}{\rightarrow} (l0 \bar{v})^\dagger \) and \( l0 \bar{v} \in T_{SB} \cap T_{-SC}(R) \cap T_{SC}^{\text{arx}}(R) \).

Proof. From \( t \in T_{SC}^{\text{arx}}(R) \) and Lemma 3.2(5), we have \( t \in T_{SC}^{\text{arx}}(R) \). From \( t \in T_{SB} \cap T_{-SC}(R) \), we have \( \neg SC(R, t) \). Hence, there exist \( l \to r \in R \) and a substitution \( \theta' \) such that \( \bar{t} \overset{\theta'}{\rightarrow} (l0 \bar{r} \theta')^\dagger \) and \( Dom(\theta') \subseteq FV(l) \). Since type(\( l = \text{type}(r) \in B \)), we have \( l0 \bar{r} \theta' \in T_{-SC}(R) \). Moreover, \( l0 \bar{r} \theta' \in T_{SC}^{\text{arx}}(R) \) follows from Lemma 3.2(3). Since \( r \in \text{Cand}(r) \) and \( \neg SC(R, r \theta') \), we have \( \{ r' \in \text{Cand}(r) \mid \neg SC(R, r' \theta') \} \neq \emptyset \). Let \( \nu' \overset{\lambda x_1 \cdots x_m a(t_1, \ldots, t_n)}{\rightarrow} \) 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 \( (\nu' \theta' u_1 \cdots u_m) \) is not strongly computable. Let \( \nu \) and \( \theta \) be \( \nu = a(r_1, \ldots, r_n) \) and \( \theta = \theta' \cup \{ x_i : := u_i \mid 1 \leq i \leq m \} \). Since \( \nu \theta \overset{\nu' \theta'}{\rightarrow} (\nu' \theta' u_1 \cdots u_m) \), we have \( \nu \theta \in T_{SB} \cap T_{-SC}(R) \). Since \( l0 \nu \theta \rightarrow l0 \nu \theta' \), we have \( l0 \nu \theta \in T_{SB} \cap T_{-SC}(R) \cap T_{SC}^{\text{arx}}(R) \).

In Sect. 5 we showed that a PFP-HRS terminates if every static recursion component is non-terminating. In order to show non-terminlessness, 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\(^\dagger\). These techniques are also effective in termination proofs for HRSs. We begin with reduction pairs.

Definition 6.1 (Reduction Pair) Let \( \geq \) be a quasi-order\(^\dagger\) each node cannot appear more than once in a path.

\(^\dagger\)A quasi-order \( \geq \) is said to be a weak reduction order if the pair \((\geq, \geq)\) of \( \geq \) and its strict part \( \geq \) is a reduction pair.
and $>$ be a strict order. The pair $(\succ, >)$ is said to be a reduction pair if the following properties hold:

- $>$ is well-founded and closed under substitution,
- $\succ$ is closed under contexts and substitutions, and
- $\succ \cdot \prec \subseteq \succ \cup \succ \cdot \prec \subseteq \succ$.

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

**Proof.** Obvious. \hfill $\square$

Next we introduce the subterm criterion for HRSs. In [8], Hirokawa and Middeldorp proved that the subterm criteria guarantees the non-loopingness in TRSs. The key of the proof is that the relation $\rightarrow \cup \succ \cdot \sub$ 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 $\pi$ from positive integers to sequences of positive integers [13]. In the following, we extend the improved subterm criterion onto HRSs, that is to handle $\lambda$-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 $\pi$ from $D_R$ to non-empty sequences of positive integers such that

$$(\alpha) \ u_{\pi(top(u))} \succ \ sub \ v_{\pi(top(v))} \text{ for some } u^\pi \rightarrow v^\pi \in C, \text{ and }$$

$$(\beta) \text{ the following conditions hold for any } u^\pi \rightarrow v^\pi \in C:$$

- $u_{\pi(top(u))} \succ \ sub \ v_{\pi(top(v))},$
- $\forall p < \pi(top(u)).top(u_p) \notin FV(u), \text{ and }$
- $\forall q < \pi(top(v)), q = e \lor top(v^\pi_q) \notin FV(v) \cup D_R.$

**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 $u_0 \rightarrow v_0, u_1 \rightarrow v_1, u_2 \rightarrow v_2, \ldots$ in which every $u^\pi \rightarrow v^\pi \in C$ occurs infinitely many times, and let $\theta_0, \theta_1, \ldots$ be substitutions such that $v_i^\pi \cdot \theta_i \rightarrow u_i^\pi \cdot \theta_{i+1} \cdot \theta_i$ and $v_i^\pi \cdot \theta_i \in T^{\arg}_{SN}(R)$ for each $i$. From Lemma 3.2(5), $u_i \cdot \theta_i, v_i \cdot \theta_i \in T^{\arg}_{SN}(R)$. Denote $\pi(top(u))$ by $p_i$ for each $i$. Since $v_i^\pi \cdot \theta_i \rightarrow u_i^\pi \cdot \theta_{i+1} \cdot \theta_i$, we have $top(v_i) = top(u_{i+1})$. Hence, from the condition (\beta) of the subterm criterion, we have

$$\left( u_0 \cdot \theta_0 \cdot \theta_i \right)_{\theta_i} \succ \ sub \ \left( v_0 \cdot \theta_0 \cdot \theta_i \right)_{\theta_i} \rightarrow \left( u_1 \cdot \theta_1 \cdot \theta_i \right)_{\theta_i} \succ \ sub \ \ldots$$

From the condition (\alpha) of the subterm criterion, the sequence above contains infinitely many $\succ$. Hence there exists an infinite sequence starting with $(u_0 \cdot \theta_0)_{\theta_j}$ with respect to $\succ \cup \succ \cdot \sub$, where $j$ is the positive integer such that $j \leq p_0$. This is a contradiction with $u_0 \cdot \theta_0 \in T^{\arg}_{SN}(R). \hfill \square$

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 $(\succ, >)$ such that $R \subseteq \succ, C \subseteq \succ \cup >$, and $C \cap > \neq \emptyset$.

**Proof.** From Corollary 5.13 and Lemma 6.2, 6.4. \hfill $\square$

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 $R_{sqsum}$ displayed in the introduction. Let $\pi(fold^L_1) = 3$, $\pi(add) = 1$, and $\pi(mul) = 1$. Then all $C \in SRC(R_{sqsum})$ (cf. Example 5.8) satisfy the subterm criterion in the underlined positions below:

$$\left\{ \begin{array}{l}
\text{fold}^L_2((x, y), X, cons(Y, L))
\rightarrow \text{fold}^L_2((x, FV(x), F(X, Y), L))
\text{add}^L_2(s(X), Y) \rightarrow \text{add}^L_2(X, Y)
\text{mul}^L_2(s(X), Y) \rightarrow \text{mul}^L_2(X, Y)
\end{array} \right.$$

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

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.