On Transforming Functions Accessing Global Variables into Logically Constrained Term Rewriting Systems*

Yoshiaki Kanazawa
Graduate School of Informatics
Nagoya University
Nagoya, Japan
yoshiaki@trs.css.i.nagoya-u.ac.jp

Naoki Nishida
Graduate School of Informatics
Nagoya University
Nagoya, Japan
nishida@i.nagoya-u.ac.jp

A usual idea of transforming imperative programs over the integers into rewriting systems is to transform programs into transition systems where some auxiliary function symbols are introduced to represent intermediate states. In transforming a function calling itself or others, we allow an auxiliary symbol corresponding to a function call to have an extra argument where the called function is executed, and values stored in global variables are passed to the called function as its extra arguments, and the caller receives the possibly updated values together with returned values of the called function. Such a mechanism for global variables performs well under sequential execution, but is not enough to adapt to parallel execution. In this paper, using an example, we show another approach to transformations of imperative programs with function calls and global variables into logically constrained term rewriting systems that represent actions of the whole execution environment with call stacks. More precisely, we prepare a function symbol for the whole environment of execution, which stores global variables and a call stack as its arguments. For a function call, we push the frame to the stack and pop it after the execution. Any running frame is executed at the top of the stack, and statements accessing global variables are represented by rewrite rules for the environment symbol.

1 Introduction

Recently, analyses of imperative programs (in C, Java Bytecode, etc.) via transformations into term rewriting systems have been investigated [2, 3, 5, 9]. In particular, *constrained rewriting systems* are popular for these transformations, since logical constraints used for modeling the control flow can be separated from terms expressing intermediate states [2, 3, 5, 7, 10]. To capture the existing approaches for constrained rewriting in one setting, the framework of *logically constrained term rewriting systems* (an LCTRS, for short) has been proposed [6]. Transformations of C programs with characters, arrays of integers, global variables, and so on into LCTRSs has been discussed in [4].

A basic idea of transforming functions defined in imperative programs over the integers is to represent transitions of parameters and local variables as rewrite rules with auxiliary function symbols. The resulting rewriting system can be considered a *transition system* w.r.t. parameters and local variables. Consider the following function `sum1` in Figure 1, which is written in C language. The function `sum1` computes the summation from 0 to a given non-negative integer x. The execution of the body of this function can be considered a transition of values (v_x, v_i, v_z) for x, i, and z, respectively. For example, we have the following transition for `sum1`(3):

$$(3,0,0) \to (3,0,1) \to (3,1,1) \to (3,1,3) \to (3,2,3) \to (3,2,6) \to (3,3,6) \to (3,3,6)$$

*This work was partially supported by Denso Corporation, NSITEXE, Inc., and JSPS KAKENHI Grant Number JP18K11160.

Submitted to:
WPTE 2018

```
int sum1(int x){
  int i = 0, z = 0;
  for( i = 0 ; i < x ; i++ ){
    z += i + 1;
  }
  return z;
}
```

Figure 1: a C program defining a function to compute the summation from 0 to x.

This transition for the execution of the function `sum1` can be modeled by an LCTRS as follows [5, 4]:

$$\mathcal{R}_1 = \left\{ \begin{array}{ll} \mathsf{sum1}(x) \to \mathsf{u}_1(x,0,0) & \\ \mathsf{u}_1(x,i,z) \to \mathsf{u}_1(x,i+1,z+(i+1))) & [\quad i<x \quad] \\ \mathsf{u}_1(x,i,z) \to \mathsf{return}(z) & [\,\neg(i<x)\,] \end{array} \right\}$$

The auxiliary function symbols $\mathsf{u}_1, \mathsf{u}_2$ can be considered locations stored in the program counter.

A function call is added as an extra argument of the auxiliary symbol that corresponds to the statement of the call. In addition to the above `sum1`, consider the following function:

```
int g(int x){
  return x * sum1(x);
}
```

This function is transformed into the following rules:

$$\mathcal{R}_2 = \{\ \mathsf{g}(x) \to \mathsf{u}_2(x,\mathsf{sum1}(x)) \quad \mathsf{u}_2(x,\mathsf{return}(y)) \to \mathsf{return}(x \times y)\ \}$$

The auxiliary function symbol u_2 calls $\mathsf{sum1}$ in the second argument.

To deal with a global variable under sequential execution, it is enough to pass a value stored in the global variable to a function call as an extra argument and to receive from the called function a value of the global variable that may be updated in executing the function call, restoring the value in the global variable. Let us add a global variable counting the total number of function calls to the above program as in Figure 2. When we consider the sequential execution, this program is transformed into the following LCTRS [4]:

$$\mathcal{R}_3 = \left\{ \begin{array}{rl} \mathsf{sum1}(x,num) \to \mathsf{u}_1(x,0,0,num+1) & \\ \mathsf{u}_1(x,i,z,num) \to \mathsf{u}_1(x,i+1,z+(i+1)),num) & [\quad i<x \quad] \\ \mathsf{u}_1(x,i,z,num) \to \mathsf{return}(z,num) & [\,\neg(i<x)\,] \\ \mathsf{g}(x,num) \to \mathsf{u}_2(x,num+1) & \\ \mathsf{u}_2(x,num) \to \mathsf{u}_3(x,\mathsf{sum1}(x,num),num) & \\ \mathsf{u}_3(x,\mathsf{return}(y,num_{new}),num_{old}) \to \mathsf{return}(x \times y,num_{new}) & \end{array} \right\}$$

The above approach to transformations of function calls is very naive but not general. For example, to model parallel execution, a value stored in a global variable do not have to be passed to a particular function or a process because another function or process may access the global variable.

In this paper, using an example, we show another approach to transformations of imperative programs with function calls and global variables into LCTRSs. Our target languages are call-by-value

```
int num = 0;

int sum1(int x){
  num ++;
  int i = 0, z = 0;
  for( i = 0 ; i < x ; i++ ){
    z += i + 1;
  }
  return z;
}

int g(int x){
  num ++;
  return x * sum1(x);
}
```

Figure 2: a C program obtained by adding the definition of g into the program for `sum`.

imperative languages such as C. For this reason, we use a small subclass of C programs over the integers as fundamental imperative programs.

Our idea of the treatment for global variables in calling functions is to prepare a new symbol to represent the whole environment for execution. Values of global variables are stored in arguments of the new symbol, and transitions accessing global variables are represented as transitions of the environment. In reduction sequences of LCTRSs obtained by the original transformation, positions of function calls are not unique, and thus, we may need (possibly infinitely) many rules for a transition related to a global variable. To solve this problem, we prepare a so-called *call stack*, and transform programs into LCTRSs that specify statements as rewrite rules for user-defined functions and the introduced symbol of the environment. In calling a function, a frame of the called function is pushed to the stack, and popped from the stack when the execution halts successfully. This implies that any running frame is called at the top of the stack, i.e., positions of function calls are unique. We transform statements not accessing global variables into rewrite rules for called functions, and transform statements accessing global variables into rewrite rules for the introduced symbol for the environment.

2 Preliminaries

In this section, we recall LCTRSs, following the definitions in [6, 4]. Familiarity with basic notions on term rewriting [1, 8] is assumed.

Let $\mathcal{S}$ be a set of *sorts* and $\mathcal{V}$ a countably infinite set of *variables*, each of which is equipped with a sort. A *signature* Σ is a set, disjoint from $\mathcal{V}$, of *function symbols* f, each of which is equipped with a *sort declaration* $\iota_1 \times \cdots \times \iota_n \Rightarrow \iota$ where $\iota_1, \ldots, \iota_n, \iota \in \mathcal{S}$. For readability, we often write ι instead of $\iota_1 \times \cdots \times \iota_n \Rightarrow \iota$ if $n = 0$. We denote the set of well-sorted *terms* over Σ and $\mathcal{V}$ by $T(\Sigma, \mathcal{V})$. In the rest of this section, we fix $\mathcal{S}$, Σ, and $\mathcal{V}$. The set of variables occurring in s is denoted by $\mathcal{V}ar(s)$. Given a term s and a *position* p (a sequence of positive integers) of s, $s|_p$ denotes the subterm of s at position p, and $s[t]_p$ denotes s with the subterm at position p replaced by t.

A *substitution* γ is a sort-preserving total mapping from $\mathcal{V}$ to $T(\Sigma, \mathcal{V})$, and naturally extended for

a mapping from $T(\Sigma,\mathcal{V})$ to $T(\Sigma,\mathcal{V})$: the result $s\gamma$ of applying a substitution γ to a term s is s with all occurrences of a variable x replaced by $\gamma(x)$. The *domain* $\mathcal{D}om(\gamma)$ of γ is the set of variables x with $\gamma(x) \neq x$. The notation $\{x_1 \mapsto s_1, \ldots, x_k \mapsto s_k\}$ denotes a substitution γ with $\gamma(x_i) = s_i$ for $1 \leq i \leq n$, and $\gamma(y) = y$ for $y \notin \{x_1, \ldots, x_n\}$.

To define LCTRSs, we consider different kinds of symbols and terms: (1) two signatures Σ_{terms} and Σ_{theory} such that $\Sigma = \Sigma_{terms} \cup \Sigma_{theory}$, (2) a mapping $\mathcal{I}$ which assigns to each sort ι occurring in Σ_{theory} a set $\mathcal{I}_\iota$, (3) a mapping $\mathcal{J}$ which assigns to each $f : \iota_1 \times \cdots \times \iota_n \Rightarrow \iota \in \Sigma_{theory}$ a function in $\mathcal{I}_{\iota_1} \times \cdots \times \mathcal{I}_{\iota_n} \Rightarrow \mathcal{I}_\iota$, and (4) a set $\mathcal{V}al_\iota \subseteq \Sigma_{theory}$ of *values*—function symbols $a : \iota$ such that $\mathcal{J}$ gives a bijective mapping from $\mathcal{V}al_\iota$ to $\mathcal{I}_\iota$—for each sort ι occurring in Σ_{theory}. We require that $\Sigma_{terms} \cap \Sigma_{theory} \subseteq \mathcal{V}al = \bigcup_{\iota \in \mathcal{S}} \mathcal{V}al_\iota$. The sorts occurring in Σ_{theory} are called *theory sorts*, and the symbols *theory symbols*. Symbols in $\Sigma_{theory} \setminus \mathcal{V}al$ are *calculation symbols*. A term in $T(\Sigma_{theory},\mathcal{V})$ is called a *logical term*. For ground logical terms, we define the interpretation as $[\![f(s_1,\ldots,s_n)]\!] = \mathcal{J}(f)([\![s_1]\!],\ldots,[\![s_n]\!])$. For every ground logical term s, there is a unique value c such that $[\![s]\!] = [\![c]\!]$. We use infix notation for theory and calculation symbols.

A *constraint* is a logical term φ of some sort *bool* with $\mathcal{I}_{bool} = \mathbb{B} = \{\top, \bot\}$, the set of *booleans*. A constraint φ is *valid* if $[\![\varphi\gamma]\!] = \top$ for all substitutions γ which map $\mathcal{V}ar(\varphi)$ to values, and *satisfiable* if $[\![\varphi\gamma]\!] = \top$ for some such substitution. A substitution γ *respects* φ if $\gamma(x)$ is a value for all $x \in \mathcal{V}ar(\varphi)$ and $[\![\varphi\gamma]\!] = \top$. We typically choose a theory signature with $\Sigma_{theory} \supseteq \Sigma^{core}_{theory}$, where Σ^{core}_{theory} contains $\mathsf{true}, \mathsf{false} : \mathit{bool}$, $\wedge, \vee, \Longrightarrow : \mathit{bool} \times \mathit{bool} \Rightarrow \mathit{bool}$, $\neg : \mathit{bool} \Rightarrow \mathit{bool}$, and, for all theory sorts ι, symbols $=_\iota, \neq_\iota : \iota \times \iota \Rightarrow \mathit{bool}$, and an evaluation function $\mathcal{J}$ that interprets these symbols as expected. We omit the sort subscripts from $=$ and $\neq$ when clear from context.

The standard integer signature Σ^{int}_{theory} is $\Sigma^{core}_{theory} \cup \{+, -, *, \mathsf{exp}, \mathsf{div}, \mathsf{mod} : \mathit{int} \times \mathit{int} \Rightarrow \mathit{int}\} \cup \{\geq, > : \mathit{int} \times \mathit{int} \Rightarrow \mathit{bool}\} \cup \{\mathsf{n} : \mathit{int} \mid n \in \mathbb{Z}\}$ with values true, false, and n for all integers $n \in \mathbb{Z}$. Thus, we use n (in sans-serif font) as the function symbol for $n \in \mathbb{Z}$ (in *math* font). We define $\mathcal{J}$ in the natural way, except: since all $\mathcal{J}(f)$ must be total functions, we set $\mathcal{J}(\mathsf{div})(n,0) = \mathcal{J}(\mathsf{mod})(n,0) = \mathcal{J}(\mathsf{exp})(n,k) = 0$ for all n and all $k < 0$. When constructing LCTRSs from, e.g., *while* programs, we can add explicit error checks for, e.g., "division by zero", to constraints (cf. [4]).

A *constrained rewrite rule* is a triple $\ell \to r\ [\varphi]$ such that ℓ and r are terms of the same sort, φ is a constraint, and ℓ has the form $f(\ell_1,\ldots,\ell_n)$ and contains at least one symbol in $\Sigma_{terms} \setminus \Sigma_{theory}$ (i.e., ℓ is not a logical term). If $\varphi = \mathsf{true}$ with $\mathcal{J}(\mathsf{true}) = \top$, we may write $\ell \to r$. We define $\mathcal{LV}ar(\ell \to r\ [\varphi])$ as $\mathcal{V}ar(\varphi) \cup (\mathcal{V}ar(r) \setminus \mathcal{V}ar(\ell))$. We say that a substitution γ *respects* $\ell \to r\ [\varphi]$ if $\gamma(x) \in \mathcal{V}al$ for all $x \in \mathcal{LV}ar(\ell \to r\ [\varphi])$, and $[\![\varphi\gamma]\!] = \top$. Note that it is allowed to have $\mathcal{V}ar(r) \not\subseteq \mathcal{V}ar(\ell)$, but fresh variables in the right-hand side may only be instantiated with *values*. Given a set $\mathcal{R}$ of constrained rewrite rules, we let $\mathcal{R}_{\mathtt{calc}}$ be the set $\{f(x_1,\ldots,x_n) \to y\ [y = f(x_1,\ldots,x_n)] \mid f : \iota_1 \times \cdots \times \iota_n \Rightarrow \iota \in \Sigma_{theory} \setminus \mathcal{V}al\}$. We usually call the elements of $\mathcal{R}_{\mathtt{calc}}$ constrained rewrite rules (or *calculation rules*) even though their left-hand side is a logical term. The *rewrite relation* $\to_{\mathcal{R}}$ is a binary relation on terms, defined by: $s[\ell\gamma]_p \to_{\mathcal{R}} s[r\gamma]_p$ if $\ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{\mathtt{calc}}$ and γ respects $\ell \to r\ [\varphi]$. A reduction step with $\mathcal{R}_{\mathtt{calc}}$ is called a *calculation*.

Now we define a *logically constrained term rewriting system* (LCTRS) as the abstract rewriting system $(T(\Sigma,\mathcal{V}), \to_{\mathcal{R}})$. An LCTRS is usually given by supplying Σ, $\mathcal{R}$, and an informal description of $\mathcal{I}$ and $\mathcal{J}$ if these are not clear from context. An LCTRS $\mathcal{R}$ is said to be *left-linear* if for every rule in $\mathcal{R}$, the left-hand side is linear. $\mathcal{R}$ is said to be *non-overlapping* if for every term s and rule $\ell \to r\ [\varphi]$ such that s reduces with $\ell \to r\ [\varphi]$ at the root position: (a) there are no other rules $\ell' \to r'\ [\varphi']$ such that s reduces with $\ell' \to r'\ [\varphi']$ at the root position, and (b) if s reduces with any rule at a non-root position q, then q is not a position of ℓ. $\mathcal{R}$ is said to be *orthogonal* if $\mathcal{R}$ is left-linear and non-overlapping. For $f(\ell_1,\ldots,\ell_n) \to r\ [\varphi] \in \mathcal{R}$, we call f a *defined symbol* of $\mathcal{R}$, and non-defined elements of Σ_{terms} and all values are called *constructors* of $\mathcal{R}$. Let $\mathcal{D}_{\mathcal{R}}$ be the set of all defined symbols and $\mathcal{C}_{\mathcal{R}}$ the set of

```
int num = 0;

int sum(int x){
  num ++;
  if( x <= 0 ){
    return 0;
  }else{
    return x + sum(x - 1);
  }
}

int main(){
  int n = 3;
  sum(n);
  return 0;
}
```

Figure 3: a C program obtained by adding the definition of g into the program for `sum`.

constructors. A term in $T(\mathcal{C}_\mathcal{R},\mathcal{V})$ is a *constructor term* of $\mathcal{R}$. We call $\mathcal{R}$ a *constructor system* if the left-hand side of each rule $\ell \to r\ [\varphi] \in \mathcal{R}$ is of the form $f(t_1,\ldots,t_n)$ with $t_1,\ldots,t_n$ constructor terms.

Example 2.1 ([4]) Let $\mathcal{S} = \{int, bool\}$, and $\Sigma = \Sigma_{terms} \cup \Sigma^{int}_{theory}$, where $\Sigma_{terms} = \{\ \mathsf{fact} : int \Rightarrow int\ \} \cup \{\ \mathsf{n} : int \mid n \in \mathbb{Z}\ \}$. Then both *int* and *bool* are theory sorts. We also define set and function interpretations, i.e., $\mathcal{I}_{int} = \mathbb{Z}$, $\mathcal{I}_{bool} = \mathbb{B}$, and $\mathcal{J}$ is defined as above. Examples of logical terms are $0 = 0 + -1$ and $x + 3 \geq y + -42$ that are constraints. $5 + 9$ is also a (ground) logical term, but not a constraint. Expected starting terms are, e.g., $\mathsf{fact}(42)$ or $\mathsf{fact}(\mathsf{fact}(-4))$. To implement an LCTRS calculating the *factorial* function, we use the signature Σ above and the following rules: $\mathcal{R}_{\mathsf{fact}} = \{\ \mathsf{fact}(x) \to 1\ [x \leq 0],\ \mathsf{fact}(x) \to x \times \mathsf{fact}(x-1)\ [\neg(x \leq 0)]\ \}$. Using calculation steps, a term $3-1$ reduces to 2 in one step with the calculation rule $x - y \to z\ [z = x - y]$, and $3 \times (2 \times (1 \times 1))$ reduces to 6 in three steps. Using the constrained rewrite rules in $\mathcal{R}_{\mathsf{fact}}$, $\mathsf{fact}(3)$ reduces in ten steps to 6.

3 A New Approach to Transformations of Imperative Programs

In this section, using an example, we introduce a new approach to transformations of imperative programs with function calls and global variables.

3.1 The Existing Transformation of Functions Accessing Global Variables

In this section, we briefly recall the transformation of imperative programs with functions accessing global variables [4] using the program in Figure 3, which is simpler than that in Section 1. The above

program is transformed into the following LCTRS [4]:

$$\mathcal{R}_4 = \left\{ \begin{array}{rl} \mathsf{sum}(x,num) \to \mathsf{u}_4(x,num+1) & \\ \mathsf{u}_4(x,num) \to \mathsf{return}(0,num) & [\ \ x \le 0\ \ ] \\ \mathsf{u}_4(x,num) \to \mathsf{u}_5(x,num) & [\ \neg(x \le 0)\ ] \\ \mathsf{u}_5(x,num) \to \mathsf{u}_6(x,\mathsf{sum}(x-1,num),num) & \\ \mathsf{u}_6(x,\mathsf{return}(y,num_{new}),num_{old}) \to \mathsf{return}(x+y,num_{new}) & \\ \mathsf{main}(num) \to \mathsf{u}_7(3,num) & \\ \mathsf{u}_7(n,num) \to \mathsf{u}_8(n,\mathsf{sum}(n,num),num) & \\ \mathsf{u}_8(n,\mathsf{return}(y,num_{new}),num_{old}) \to \mathsf{return}(0,num_{new}) & \end{array} \right\}$$

where $\mathsf{sum}, \mathsf{u}_4, \mathsf{u}_5, \mathsf{u}_7, \mathsf{return} : \mathit{int} \times \mathit{int} \Rightarrow \mathit{state}$, $\mathsf{u}_6, \mathsf{u}_8 : \mathit{int} \times \mathit{state} \times \mathit{int} \Rightarrow \mathit{state}$, and $\mathsf{main} : \mathit{int} \Rightarrow \mathit{state}$. To represent the function call `sum(x - 1)`, the auxiliary function symbol u_6 for the function call takes the term $\mathsf{sum}(x-1,num)$ as the second argument. The function symbol sum takes two arguments, while the original function `sum` in the program takes one argument. This is because the global variable `num` is accessed during the execution of `sum`, and we pass the value stored in *num* to sum, passing the variable itself to sum in the constructed rule. The function `sum` increments the global variable `num`, and thus, we include the value stored in *num* in the result of sum via $\mathsf{return}(x+y,num_{new})$. The rule of u_6 is used after the reduction of $\mathsf{sum}(x-1,num)$, receiving the result via the pattern $\mathsf{return}(y,num_{new})$. The updated value stored in `num` is received by num_{new}, and the rule of u_6 updates the global variable `num` by including num_{new} in the result. We do the same for the function call `sum(n)` in the auxiliary function symbol u_8. For the execution of the program, we have the reduction of $\mathcal{R}_4$ illustrated in Figure 4: Note that the global variable `num` is initialized by 0 and we started from $\mathsf{main}(0)$. From the above reduction, we can see that the called function is the only running one under sequential execution, and others are waiting for the called function halting. The approach above to function calls and global variables is enough for sequential execution.

In the LCTRS $\mathcal{R}_4$ above, the function symbol u_6 recursively calls sum in its second argument. For this reason, the running function is located below u_6, and positions where sum is called are not unique. The above approach to transformations of function calls is very naive but not general. For example, to model parallel execution, a value stored in a global variable do not have to be passed to a particular function or a process because another function or process may access the global variable.

3.2 Another Approach to Global Variables

In this section, we show another approach to the treatment of global variables.

To adapt to parallel execution, global variables used like shared memories should be located at fixed places because they may be accessed from two or more functions or processes. To keep values stored in global variables at fixed positions, we do not pass (values of) global variables to called functions in order to avoid locally updating global variables. To this end, we prepare a new function symbol env to represent the whole environment for execution, and make env have values stored in global variables in its arguments. In addition, we make env have an extra argument where functions or processes are executed sequentially. When we execute n (> 1) processes in parallel, we make env have n extra arguments where ith process is executed in the ith extra argument. For example, the process of executing the above program is expressed as follows:

$$\mathsf{env}(0,\mathsf{main}())$$

Note that env has the sort $\mathit{int} \times \mathit{state} \Rightarrow \mathit{univ}$. The first argument of env is the place where the value of the

$$
\begin{aligned}
\mathsf{main}(0) &\to_{\mathcal{R}_4} \mathsf{u}_7(3,0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{sum}(3,0),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_4(3,0+1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_4(3,1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_5(3,1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{sum}(3-1,1),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{sum}(2,1),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_4(2,1+1),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_4(2,2),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_5(2,2),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_6(2,\mathsf{sum}(2-1,2),2),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_6(2,\mathsf{sum}(1,2),2),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_6(2,\mathsf{u}_4(1,2+1),2),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_6(2,\mathsf{u}_4(1,3),2),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_6(2,\mathsf{u}_5(1,3),2),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_6(2,\mathsf{u}_6(1,\mathsf{sum}(1-1,3),3),2),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_6(2,\mathsf{u}_6(1,\mathsf{sum}(0,3),3),2),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_6(2,\mathsf{u}_6(1,\mathsf{u}_4(0,3+1),3),2),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_6(2,\mathsf{u}_6(1,\mathsf{u}_4(0,4),3),2),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_6(2,\mathsf{u}_6(1,\mathsf{return}(0,4),3),2),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_6(2,\mathsf{return}(1+0,4),2),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{u}_6(2,\mathsf{return}(1,4),2),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{return}(2+1,4),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{u}_6(3,\mathsf{return}(3,4),1),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{return}(3+3,4),0)\\
&\to_{\mathcal{R}_4} \mathsf{u}_8(3,\mathsf{return}(6,4),0)\\
&\to_{\mathcal{R}_4} \mathsf{return}(0,4)
\end{aligned}
$$

Figure 4: the reduction of $\mathcal{R}_4$ for the execution of the program for `sum`.

global variable `num` is stored, and the second argument of env is the place where functions are executed, e.g., the main function `main` is called in the above term.

We do not change the transformation of *local statements*—statements without accessing global variables—in function definitions. Let us consider the execution of the program, i.e., `main`. All the statements in `main` are local, and thus, we transform the definition of `main` as follows:

$$
\left\{
\begin{array}{r}
\mathsf{main}() \to \mathsf{u}_7(3)\\
\mathsf{u}_7(n) \to \mathsf{u}_8(n,\mathsf{sum}(n))\\
\mathsf{u}_8(n,\mathsf{return}(y)) \to \mathsf{return}(0)
\end{array}
\right\}
$$

In executing the program (i.e., `main`), the first access to the global variable `num` is the statement `num++`, the increment of `num`, in the definition of sum. The initial term $\mathsf{env}(0,\mathsf{main}())$ can be reduced to $\mathsf{env}(0,\mathsf{u}_8(3,\mathsf{sum}(3))$, and thus, the first execution of the statement `num++` can be expressed by the following rewrite rule for env:

$$\mathsf{env}(\mathit{num},\mathsf{u}_8(n,\mathsf{sum}(x))) \to \mathsf{env}(\mathit{num}+1,\mathsf{u}_8(n,\mathsf{u}_4(x)))$$

The other statements in the definition of `sum` are local and we transform them into the following rules:

$$\left\{\begin{array}{rl} \mathsf{u}_4(x) \to \mathsf{return}(x) & [\ \ x \le 0\ \ ] \\ \mathsf{u}_4(x) \to \mathsf{u}_5(x) & [\ \neg(x \le 0)\ ] \\ \mathsf{u}_5(x) \to \mathsf{u}_6(x, \mathsf{sum}(x-1)) & \\ \mathsf{u}_6(x, \mathsf{return}(y)) \to \mathsf{return}(x+y) & \end{array}\right\}$$

The above rules are not enough to capture all possible executions, e.g. the second execution of `num++`, which is done by the second call of `sum`, is not expressed yet. Thus, we prepare the following rule:

$$\mathsf{env}(\mathit{num}, \mathsf{u}_8(n, \mathsf{u}_6(x', \mathsf{sum}(x)))) \to \mathsf{env}(\mathit{num}+1, \mathsf{u}_8(n, \mathsf{u}_6(x', \mathsf{u}_4(x))))$$

In addition, `sum` is further recursively called, and we need the following rule:

$$\mathsf{env}(\mathit{num}, \mathsf{u}_8(n, \mathsf{u}_6(x'', \mathsf{u}_6(x', \mathsf{sum}(x))))) \to \mathsf{env}(\mathit{num}+1, \mathsf{u}_8(n, \mathsf{u}_6(x'', \mathsf{u}_6(x', \mathsf{u}_4(x)))))$$

Finally, we need another similar rule for `sum(3)`. However, in general, `sum` may receive all the (finitely many) integers, and we need too much rules, all of which express the increment of `num`. In addition, we may need other cases when we added other functions calling `sum`. This troublesome is caused by the fact that positions where sum is called are not unique in the above approach.

3.3 Using a Call Stack for Function Calls

In this section, we show a representation of function calls for transformed LCTRSs by means of the example in Figure 3.

The approach to the treatment of global variables in the previous section needs many similar rules for statements accessing global variables, and we have to add other similar rules when we introduce another function that may call itself or other functions. As described at the end of the previous section, the cause of this problem is that positions where functions are called in terms of the form $\mathsf{env}(\ldots)$ are not unique due to nests of auxiliary function symbols. A solution to fix this problem is to make such positions unique. An execution is represented as a term rooted by env, and global variables are located at fixed positions (i.e., arguments of env). The last argument of env is used for execution of user-defined functions. In the last argument, we fix positions where functions are called by using a so-called *call stack*. To this end, we prepare a binary function symbol $\mathsf{stack} : \mathit{state} \times \mathit{process} \Rightarrow \mathit{process}$ and a constant $\bot : \mathit{process}$ (the empty stack). To adapt to stacks, we change the sort of env. For example, we give $\mathit{int} \times \mathit{process} \Rightarrow \mathit{univ}$ to env, and the initial term for the execution of the program is the following one:

$$\mathsf{env}(0, \mathsf{stack}(\mathsf{main}(), \bot))$$

In this approach, the environment has a stack s to execute functions by means of the form $\mathsf{env}(\ldots, s)$. In calling a function f as $\mathsf{f}(\vec{t})$, we push $\mathsf{f}(\vec{t})$ as a frame for the function call to the stack s, and after halting the execution successfully, we pop the frame of the form $\mathsf{return}(\ldots)$ from the stack s.

The statements of calling functions in the above example are transformed into the following rules:

$$\left\{\begin{array}{rl} \mathsf{stack}(\mathsf{u}_5(x), s) \to \mathsf{stack}(\mathsf{sum}(x-1), \mathsf{stack}(\mathsf{u}_6(x), s)) & [\ \neg(x \le 0)\ ] \\ \mathsf{stack}(\mathsf{return}(y), \mathsf{stack}(\mathsf{u}_6(x), s)) \to \mathsf{stack}(\mathsf{return}(x+y), s) & \\ \mathsf{stack}(\mathsf{u}_7(n), s) \to \mathsf{stack}(\mathsf{sum}(n), \mathsf{stack}(\mathsf{u}_8(n), s)) & \\ \mathsf{stack}(\mathsf{return}(y), \mathsf{stack}(\mathsf{u}_8(n))) \to \mathsf{stack}(\mathsf{return}(0), s) & \end{array}\right\}$$

The first and third rules push frames to the stack, and the second and fourth pop frames. For a term $\mathsf{env}(x_1,\ldots,x_n,\mathsf{stack}(\ldots))$, the reduction of user-defined functions is performed at the position $n+1$ of the term, where $x_1,\ldots,x_n$ are global variables. For this reason, statements accessing global variables can be represented by the following form:

$$\mathsf{env}(x_1,\ldots,x_n,\mathsf{stack}(\mathsf{f}(\ldots),s)) \to \mathsf{env}(t_1,\ldots,t_n,\mathsf{stack}(\mathsf{g}(\ldots),s))\ [\varphi]$$

Note that s in the above rule is a variable. The statement `num++` in the above example is transformed into the following rule:

$$\mathsf{env}(\mathit{num},\mathsf{stack}(\mathsf{sum}(x),s)) \to \mathsf{env}(\mathit{num}+1,\mathsf{stack}(\mathsf{u}_4(x),s))$$

In summary, the above example is transformed into the following LCTRS:

$$\mathcal{R}_5 = \left\{ \begin{array}{rll}
\mathsf{env}(\mathit{num},\mathsf{stack}(\mathsf{sum}(x),s)) \to \mathsf{env}(\mathit{num}+1,\mathsf{stack}(\mathsf{u}_4(x),s)) & & \\
\mathsf{u}_4(x) \to \mathsf{return}(0) & [\quad x \le 0 \quad] \\
\mathsf{u}_4(x) \to \mathsf{u}_5(x) & [\,\neg(x \le 0)\,] \\
\mathsf{stack}(\mathsf{u}_5(x),s) \to \mathsf{stack}(\mathsf{sum}(x-1),\mathsf{stack}(\mathsf{u}_6(x),s)) & & \\
\mathsf{stack}(\mathsf{return}(y),\mathsf{stack}(\mathsf{u}_6(x),s)) \to \mathsf{stack}(\mathsf{return}(x+y),s) & & \\
\mathsf{main}() \to \mathsf{u}_7(3) & & \\
\mathsf{stack}(\mathsf{u}_7(n),s) \to \mathsf{stack}(\mathsf{sum}(n),\mathsf{stack}(\mathsf{u}_8(n),s)) & & \\
\mathsf{stack}(\mathsf{return}(y),\mathsf{stack}(\mathsf{u}_8(n))) \to \mathsf{stack}(\mathsf{return}(0),s) & &
\end{array} \right\}$$

For the execution of the program, we have the reduction of $\mathcal{R}_5$ illustrated in Figure 5.

The function symbol stack is a defined symbol of $\mathcal{R}_5$, while it looks a constructor for stacks. If we would like the resulting LCTRS to be a constructor system, rules performing "push" and "pop" for stacks may be generated as rules for env. More precisely, instead of $\mathsf{stack}(\mathsf{f}(\vec{x}),s) \to \mathsf{stack}(\mathsf{g}(\vec{t}),\mathsf{stack}(\mathsf{f}'(\vec{x}),s))$, we generate

$$\mathsf{env}(\vec{y},\mathsf{stack}(\mathsf{f}(\vec{x}),s)) \to \mathsf{env}(\vec{y},\mathsf{stack}(\mathsf{g}(\vec{t}),\mathsf{stack}(\mathsf{f}'(\vec{x}),s)))$$

and instead of $\mathsf{stack}(\mathsf{return}(z),\mathsf{stack}(\mathsf{f}'(\vec{x}),s)) \to \mathsf{stack}(\mathsf{f}''(\vec{x}),s)$, we generate

$$\mathsf{env}(\vec{y},\mathsf{stack}(\mathsf{return}(z),\mathsf{stack}(\mathsf{f}'(\vec{x}),s))) \to \mathsf{env}(\vec{y},\mathsf{stack}(\mathsf{f}''(\vec{x}),s))$$

For example, $\mathcal{R}_5$ can be modified to the following constructor LCTRS:

$$\left\{ \begin{array}{rll}
\mathsf{env}(\mathit{num},\mathsf{stack}(\mathsf{sum}(x),s)) \to \mathsf{env}(\mathit{num}+1,\mathsf{stack}(\mathsf{u}_4(x),s)) & & \\
\mathsf{u}_4(x) \to \mathsf{return}(0) & [\quad x \le 0 \quad] \\
\mathsf{u}_4(x) \to \mathsf{u}_5(x) & [\,\neg(x \le 0))\,] \\
\mathsf{env}(\mathit{num},\mathsf{stack}(\mathsf{u}_5(x),s)) \to \mathsf{env}(\mathit{num},\mathsf{stack}(\mathsf{sum}(x-1),\mathsf{stack}(\mathsf{u}_6(x),s)) & & \\
\mathsf{env}(\mathit{num},\mathsf{stack}(\mathsf{return}(y),\mathsf{stack}(\mathsf{u}_6(x),s))) \to \mathsf{env}(\mathit{num},\mathsf{stack}(\mathsf{return}(x+y),s)) & & \\
\mathsf{main}() \to \mathsf{u}_7(3) & & \\
\mathsf{env}(\mathit{num},\mathsf{stack}(\mathsf{u}_7(n),s)) \to \mathsf{env}(\mathit{num},\mathsf{stack}(\mathsf{sum}(n),\mathsf{stack}(\mathsf{u}_8(n),s))) & & \\
\mathsf{env}(\mathit{num},\mathsf{stack}(\mathsf{return}(y),\mathsf{stack}(\mathsf{u}_8(n)))) \to \mathsf{env}(\mathit{num},\mathsf{stack}(\mathsf{return}(0),s)) & &
\end{array} \right\}$$

The reduction of this LCTRS from $\mathsf{env}(0,\mathsf{stack}(\mathsf{main}(),\bot))$ is the same as that in Figure 5.

$$
\begin{aligned}
&\mathsf{env}(0,\mathsf{stack}(\mathsf{main}(),\bot))\\
&\to_{\mathcal{R}_5} \mathsf{env}(0,\mathsf{stack}(\mathsf{u}_7(3),\bot))\\
&\to_{\mathcal{R}_5} \mathsf{env}(0,\mathsf{stack}(\mathsf{sum}(3),\mathsf{stack}(\mathsf{u}_8(3),\bot)))\\
&\to_{\mathcal{R}_5} \mathsf{env}(0+1,\mathsf{stack}(\mathsf{u}_4(3),\mathsf{stack}(\mathsf{u}_8(3),\bot)))\\
&\to_{\mathcal{R}_5} \mathsf{env}(1,\mathsf{stack}(\mathsf{u}_4(3),\mathsf{stack}(\mathsf{u}_8(3),\bot)))\\
&\to_{\mathcal{R}_5} \mathsf{env}(1,\mathsf{stack}(\mathsf{u}_5(3),\mathsf{stack}(\mathsf{u}_8(3),\bot)))\\
&\to_{\mathcal{R}_5} \mathsf{env}(1,\mathsf{stack}(\mathsf{sum}(3-1),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(1,\mathsf{stack}(\mathsf{sum}(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(1+1,\mathsf{stack}(\mathsf{u}_4(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(2,\mathsf{stack}(\mathsf{u}_4(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(2,\mathsf{stack}(\mathsf{u}_5(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(2,\mathsf{stack}(\mathsf{sum}(2-1),\mathsf{stack}(\mathsf{u}_6(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot)))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(2,\mathsf{stack}(\mathsf{sum}(1),\mathsf{stack}(\mathsf{u}_6(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot)))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(2+1,\mathsf{stack}(\mathsf{u}_4(1),\mathsf{stack}(\mathsf{u}_6(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot)))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(3,\mathsf{stack}(\mathsf{u}_4(1),\mathsf{stack}(\mathsf{u}_6(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot)))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(3,\mathsf{stack}(\mathsf{u}_5(1),\mathsf{stack}(\mathsf{u}_6(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot)))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(3,\mathsf{stack}(\mathsf{sum}(1-1),\mathsf{stack}(\mathsf{u}_6(1),\mathsf{stack}(\mathsf{u}_6(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot))))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(3,\mathsf{stack}(\mathsf{sum}(0),\mathsf{stack}(\mathsf{u}_6(1),\mathsf{stack}(\mathsf{u}_6(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot))))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(3+1,\mathsf{stack}(\mathsf{u}_4(0),\mathsf{stack}(\mathsf{u}_6(1),\mathsf{stack}(\mathsf{u}_6(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot))))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(4,\mathsf{stack}(\mathsf{u}_4(0),\mathsf{stack}(\mathsf{u}_6(1),\mathsf{stack}(\mathsf{u}_6(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot))))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(4,\mathsf{stack}(\mathsf{u}_5(0),\mathsf{stack}(\mathsf{u}_6(1),\mathsf{stack}(\mathsf{u}_6(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot))))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(4,\mathsf{stack}(\mathsf{return}(0),\mathsf{stack}(\mathsf{u}_6(1),\mathsf{stack}(\mathsf{u}_6(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot))))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(4,\mathsf{stack}(\mathsf{return}(1+0),\mathsf{stack}(\mathsf{u}_6(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot)))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(4,\mathsf{stack}(\mathsf{return}(1),\mathsf{stack}(\mathsf{u}_6(2),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot)))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(4,\mathsf{stack}(\mathsf{return}(2+1),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(4,\mathsf{stack}(\mathsf{return}(3),\mathsf{stack}(\mathsf{u}_6(3),\mathsf{stack}(\mathsf{u}_8(3),\bot))))\\
&\to_{\mathcal{R}_5} \mathsf{env}(4,\mathsf{stack}(\mathsf{return}(3+3),\mathsf{stack}(\mathsf{u}_8(3),\bot)))\\
&\to_{\mathcal{R}_5} \mathsf{env}(4,\mathsf{stack}(\mathsf{return}(6),\mathsf{stack}(\mathsf{u}_8(3),\bot)))\\
&\to_{\mathcal{R}_5} \mathsf{env}(4,\mathsf{stack}(\mathsf{return}(0),\bot))
\end{aligned}
$$

Figure 5: the reduction of $\mathcal{R}_5$ for the execution of the program for `sum`.

4 Conclusion

In this paper, we illustrated a new approach to transformations of imperative programs with function calls and global variables into LCTRSs using an example. We will formalize the transformation based on the approach, proving correctness of the transformation.

A direction of future work is to apply this approach to a sequential program and its parallelized version in order to prove their equivalence. To simplify the discussion, we considered a program executed as a single process, i.e., executed *sequentially*, and the introduced symbol env has an argument that is used for the single process (see $\mathcal{R}_5$ again). To adapt to *parallel* execution where the number of executed processes is fixed, it suffices to add arguments for all executed processes into the symbol env. We will formalize this idea and prove the correctness of the transformation for parallel execution.

References

[1] Franz Baader & Tobias Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press, doi:10.1145/505863.505888.

[2] Stephan Falke & Deepak Kapur (2009): *A Term Rewriting Approach to the Automated Termination Analysis of Imperative Programs*. In Renate A. Schmidt, editor: *Proceedings of the 22nd International Conference on Automated Deduction*, *Lecture Notes in Computer Science* 5663, Springer, pp. 277–293, doi:10.1007/978-3-642-02959-2_22.

[3] Stephan Falke, Deepak Kapur & Carsten Sinz (2011): *Termination Analysis of C Programs Using Compiler Intermediate Languages*. In Manfred Schmidt-Schauß, editor: *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications*, *LIPIcs* 10, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 41–50, doi:10.4230/LIPIcs.RTA.2011.41. Available at `https://doi.org/10.4230/LIPIcs.RTA.2011.41`.

[4] Carsten Fuhs, Cynthia Kop & Naoki Nishida (2017): *Verifying Procedural Programs via Constrained Rewriting Induction*. *ACM Transactions on Computational Logic* 18(2), pp. 14:1–14:50, doi:10.1145/3060143.

[5] Yuki Furuichi, Naoki Nishida, Masahiko Sakai, Keiichirou Kusakari & Toshiki Sakabe (2008): *Approach to Procedural-program Verification Based on Implicit Induction of Constrained Term Rewriting Systems*. *IPSJ Transactions on Programming* 1(2), pp. 100–121. In Japanese (a translated summary is available from `http://www.trs.css.i.nagoya-u.ac.jp/crisys/`).

[6] Cynthia Kop & Naoki Nishida (2013): *Term Rewriting with Logical Constraints*. In Pascal Fontaine, Christophe Ringeissen & Renate A. Schmidt, editors: *Proceedings of the 9th International Symposium on Frontiers of Combining Systems*, *Lecture Notes in Computer Science* 8152, Springer, pp. 343–358, doi:10.1007/978-3-642-40885-4_24.

[7] Naoki Nakabayashi, Naoki Nishida, Keiichirou Kusakari, Toshiki Sakabe & Masahiko Sakai (2011): *Lemma Generation Method in Rewriting Induction for Constrained Term Rewriting Systems*. *Computer Software* 28(1), pp. 173–189. In Japanese (a translated summary is available from `http://www.trs.css.i.nagoya-u.ac.jp/crisys/`).

[8] Enno Ohlebusch (2002): *Advanced Topics in Term Rewriting*. Springer, doi:10.1007/978-1-4757-3661-8.

[9] Carsten Otto, Marc Brockschmidt, Christian von Essen & Jürgen Giesl (2010): *Automated termination analysis of Java bytecode by term rewriting*. In Christopher Lynch, editor: *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, *LIPIcs* 6, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 259–276.

[10] Tsubasa Sakata, Naoki Nishida, Toshiki Sakabe, Masahiko Sakai & Keiichirou Kusakari (2009): *Rewriting Induction for Constrained Term Rewriting Systems*. *IPSJ Transactions on Programming* 2(2), pp. 80–96. In Japanese (a translated summary is available from `http://www.trs.css.i.nagoya-u.ac.jp/crisys/`).