A Nesting-Preserving Transformation of SIMP Programs into Logically Constrained Term Rewrite Systems*

Naoki Nishida
Nagoya University
Nagoya Japan
`nishida@i.nagoya-u.ac.jp`

Misaki Kojima
Nagoya University
Nagoya, Japan
`k-misaki@trs.css.i.nagoya-u.ac.jp`

Ayuka Matsumi
Nagoya University
Nagoya, Japan
`matsumi@trs.css.i.nagoya-u.ac.jp`

In the last decade, several transformations of an imperative program into a logically constrained term rewrite system (LCTRS, for short) have been investigated and extended. They do not preserve the nesting of statements, generating rewrite rules like transition systems, while function calls are represented by the nesting of function symbols. Structural features of the original program must often be useful in analyzing the transformed LCTRS, but, to use such features, we have to know how to transform the program into the LCTRS, to keep the correspondence between statements in the program and the introduced auxiliary function symbols in the LCTRS, or to transform the LCTRS into a control flow graph to recover loop information. In this paper, we propose a nesting-preserving transformation of a SIMP program (a C integer program) into an LCTRS. The transformation is mostly based on previous work and introduces the nesting of function symbols that correspond to statements in the original program. To be more precise, we propose a construction of a tree homomorphism which is used as a post-process of the transformation in previous work, i.e., which is applied to the LCTRS obtained from the program. As a correctness of the nesting-preserving transformation, we show that the tree homomorphism is sound and complete for the reduction of the LCTRS.

1 Introduction

In the last decade, approaches to program verification by means of logically constrained term rewrite systems (LCTRSs, for short) [13] are well investigated [7, 17, 4, 15, 8, 9, 12, 10, 11]. LCTRSs are known to be useful as computation models of not only functional but also imperative programs. Especially, equivalence checking by means of LCTRSs is useful to ensure correctness of terminating functions (cf. [7]). Here, equivalence of two functions means that for every input, the functions return the same output or end with the same projection of final configurations.

The transformation in [7] for *SIMP programs*[1] [6] (C integer programs[2]) has been extended to global variables and function calls [8]. Then, the extended transformation has further been extended to concurrent ones with semaphore-based exclusive control [12]. A general ultimate goal is to apply LCTRSs to verification of practical programs, e.g., automotive embedded systems.

*This work was partially supported by JSPS KAKENHI Grant Number 18K11160 and DENSO Corporation.

[1] A SIMP program is originally a statement consisting of assignment, sequencing, conditionals, loops, and integer variables. On the other hand, a SIMP program in [7, 8] and this paper is a collection of function declarations which are defined by a statement with `return` statement and without any function call.

[2] `https://termination-portal.org/wiki/C_Integer_Programs`

Submitted to:
WPTE 2023

© N. Nishida, M. Kojima, & A. Matsumi
This work is licensed under the
Creative Commons Attribution License.

Program 1: a SIMP program $\mathcal{P}_{\mathtt{loop}}$ with a triple loop

```
int loop(int n) {
  int ret = 0;
  int i = 0;
  while( i < n ) {
    int j = 0;
    while( j < i ) {
      int k = 0;
      while( k < j ) {
        ret = ret + 1;
        k = k + 1;
      }
      j = j + 1;
    }
    i = i + 1;
  }
  return ret;
}
```

In the transformations mentioned above, function calls are represented by the nesting of function symbols such as *call stacks*: In [7], the auxiliary function symbol for a statement with a function call has an argument to store the execution of the called function; in [8], a function symbol introduced for configurations stores a call stack as an argument. On the other hand, the nesting of statements is not preserved as any structural feature of LCTRSs.

Example 1.1 Let us consider Program 1 which is a SIMP program which is executed like C language.[3] Program 1 is transformed into the LCTRS $\mathcal{R}_{\mathtt{loop}}$ in Figure 1 [7] (cf. [3]). For some reason explained in Example 1.2 below, we do not explain the detail of the transformation, e.g., the meaning of the introduced auxiliary function symbols such as loop_2.

The `while` statements are nested, but $\mathcal{R}_{\mathtt{loop}}$ does not have the nesting of any function symbol that is neither a value nor a built-in operator. If we know how to transform Program 1 into $\mathcal{R}_{\mathtt{loop}}$ in detail, we would be able to transform $\mathcal{R}_{\mathtt{loop}}$ back into Program 1. In this case, we do not have to keep Program 1 because we can know it by transforming $\mathcal{R}_{\mathtt{loop}}$ back into a program. In the case where we have the LCTRS but not either the original program or the transformation process, we would be able to transform the LCTRS into a *control flow graph* which corresponds to a SIMP program and to, e.g., recover loop information by means of control flow analysis (cf. [1]).

To make the analysis of the transformed LCTRS easier, we often simplify it by composing two or more successive rules. LCTRS $\mathcal{R}_{\mathtt{loop}}$ is simplified to the LCTRS $\mathcal{R}'_{\mathtt{loop}}$ in Figure 2. For example, the first three rules of $\mathcal{R}_{\mathtt{loop}}$ are composed to the first rule of $\mathcal{R}'_{\mathtt{loop}}$. For such a simplified LCTRS, we would be able to recover loop information by means of a control flow graph obtained from the simplified LCTRS.

Let $\mathcal{P}$ be a SIMP program and $\mathfrak{T}$ be a transformation of SIMP programs into LCTRSs such that $\mathfrak{T}$ has some sufficient properties to analyze $\mathcal{P}$ by means of the resulting LCTRS $\mathfrak{T}(\mathcal{P})$. For example, to

[3]Unlike C language, the range of variables with type `int` is not the 32-bit integers but the integers.

$$\mathcal{R}_{\mathtt{loop}} = \left\{ \begin{array}{rcll}
\mathsf{loop}(n) & \to & \mathsf{loop}_2(n) & \\
\mathsf{loop}_2(n) & \to & \mathsf{loop}_3(n,0) & \\
\mathsf{loop}_3(n,ret) & \to & \mathsf{loop}_4(n,ret,0) & \\
\mathsf{loop}_4(n,ret,i) & \to & \mathsf{loop}_5(n,ret,i) & [\ i<n\ ] \\
\mathsf{loop}_4(n,ret,i) & \to & \mathsf{loop}_{16}(n,ret,i) & [\neg(i<n)] \\
\mathsf{loop}_5(n,ret,i) & \to & \mathsf{loop}_6(n,ret,i,0) & \\
\mathsf{loop}_6(n,ret,i,j) & \to & \mathsf{loop}_7(n,ret,i,j) & [\ j<i\ ] \\
\mathsf{loop}_6(n,ret,i,j) & \to & \mathsf{loop}_{14}(n,ret,i) & [\neg(j<i)] \\
\mathsf{loop}_7(n,ret,i,j) & \to & \mathsf{loop}_8(n,ret,i,j,0) & \\
\mathsf{loop}_8(n,ret,i,j,k) & \to & \mathsf{loop}_9(n,ret,i,j,k) & [\ k<j\ ] \\
\mathsf{loop}_8(n,ret,i,j,k) & \to & \mathsf{loop}_{12}(n,ret,i,j) & [\neg(k<j)] \\
\mathsf{loop}_9(n,ret,i,j,k) & \to & \mathsf{loop}_{10}(n,ret+1,i,j,k) & \\
\mathsf{loop}_{10}(n,ret,i,j,k) & \to & \mathsf{loop}_8(n,ret,i,j,k+1) & \\
\mathsf{loop}_{12}(n,ret,i,j) & \to & \mathsf{loop}_6(n,ret,i,j+1) & \\
\mathsf{loop}_{14}(n,ret,i) & \to & \mathsf{loop}_4(n,ret,i+1) & \\
\mathsf{loop}_{16}(n,ret,i) & \to & \mathsf{return}(ret) &
\end{array} \right\}$$

Figure 1: the LCTRS $\mathcal{R}_{\mathtt{loop}}$ obtained from Program 1 [7].

$$\mathcal{R}'_{\mathtt{loop}} = \left\{ \begin{array}{rcll}
\mathsf{loop}(n) & \to & \mathsf{loop}_4(n,0,0) & \\
\mathsf{loop}_4(n,ret,i) & \to & \mathsf{loop}_6(n,ret,i,0) & [\ i<n\ ] \\
\mathsf{loop}_4(n,ret,i) & \to & \mathsf{return}(ret) & [\neg(i<n)] \\
\mathsf{loop}_6(n,ret,i,j) & \to & \mathsf{loop}_8(n,ret,i,j,0) & [\ j<i\ ] \\
\mathsf{loop}_6(n,ret,i,j) & \to & \mathsf{loop}_4(n,ret,i+1) & [\neg(j<i)] \\
\mathsf{loop}_8(n,ret,i,j,k) & \to & \mathsf{loop}_8(n,ret+1,i,j,k+1) & [\ k<j\ ] \\
\mathsf{loop}_8(n,ret,i,j,k) & \to & \mathsf{loop}_6(n,ret,i,j+1) & [\neg(k<j)]
\end{array} \right\}$$

Figure 2: the LCTRS $\mathcal{R}'_{\mathtt{loop}}$ obtained from $\mathcal{R}_{\mathtt{loop}}$ by the simplification based on *chaining* [7].

prove termination of $\mathcal{P}$, it is expected that termination of the LCTRS $\mathfrak{T}(\mathcal{P})$ implies termination of any execution of $\mathcal{P}$. Note that the resulting LCTRS $\mathfrak{T}(\mathcal{P})$ does not have to be transformed back into the original program $\mathcal{P}$. On the other hand, some syntactic features of the program $\mathcal{P}$ would be useful for the analysis of the LCTRS $\mathfrak{T}(\mathcal{P})$.

It is worth developing theories and methods for LCTRSs themselves, and it is meaningful enough to focus only on LCTRSs and to e.g., analyze $\mathfrak{T}(\mathcal{P})$ without meta-information such as the transformation process and the meaning of the introduced auxiliary function symbols. For this reason, we do not want to rely on such meta-information, do not want LCTRSs to keep it, and do not transform LCTRSs into control flow graphs to e.g., recover loop information.

Example 1.2 Let us consider which function symbols in $\mathcal{R}'_{\mathtt{loop}}$ in Figure 2 correspond to the inner loops of $\mathcal{P}_{\mathtt{loop}}$ in Program 1. The answer is that

- loop_4 represents the outermost one,
- loop_6 represents the second outer one, and
- loop_8 represents the innermost one.

The transformation process can provide the above correspondences.

Let us now assume that we do not know how the LCTRS $\mathcal{R}'_{\texttt{loop}}$ is obtained from Program 1 and what the function symbols loop_i mean. From the third rule with the right-hand side $\mathsf{return}(ret)$, we guess the first correspondence above; then, from the fifth rule with the right-hand side $\mathsf{loop}_4(n, ret, i+1)$, we guess the second one above; finally, we guess the remaining one.

The heuristic analysis in Example 1.2 needs a sort of graph traversal like control flow analysis, which may be expensive for what we want to know. For this reason, we would like to avoid such traversal-based analysis.

For e.g., the analysis in Example 1.2, what we can rely on must be structural features. In applying $\mathfrak{T}$ to $\mathcal{P}$, some structural features of $\mathcal{P}$ can be preserved as structural features of $\mathfrak{T}(\mathcal{P})$. Structural features of LCTRSs are mainly related to rewrite rules and terms. Rewrite rules may be simplified in advance in order to e.g., ease the analysis. For this reason, we focus on structural features of terms in LCTRSs. One of the simplest structural features of the original program $\mathcal{P}$ and the resulting LCTRS $\mathfrak{T}(\mathcal{P})$ must be the nesting of statements and function symbols, respectively.

In this paper, we propose a nesting-preserving transformation of a SIMP program into an LCTRS. The transformation $\mathfrak{T}_{np}$ is mostly based on previous work and introduces the nesting of function symbols that correspond to statements in the original program. For example, instead of $\mathsf{loop}_7(n, ret, i, j)$ in $\mathcal{R}_{\texttt{loop}}$, we generate $\mathsf{loop}_4(n, ret, i, \mathsf{loop}_6(j, \mathsf{loop}_7))$.

To propose $\mathfrak{T}_{np}$, we use the transformation $\mathfrak{T}$ in previous work but do not modify $\mathfrak{T}$. Instead, we propose a construction of an injective *tree homomorphism* [5] as a post-process of $\mathfrak{T}$, which is applied to $\mathfrak{T}(\mathcal{P})$ (Section 3). We prove correctness of the nesting-preserving transformation by showing that the tree homomorphism is sound and complete for the reduction of $\mathfrak{T}(\mathcal{P})$ (Section 4). Our transformation proceeds as follows: Given a SIMP program P,

1. we apply $\mathfrak{T}$ to $\mathcal{P}$, obtaining the LCTRS $\mathfrak{T}(\mathcal{P})$ and the correspondence between introduced auxiliary function symbols and statements in $\mathcal{P}$;
2. using the correspondence, we define an injective tree homomorphism $\xi_{\mathcal{P}}$ for the introduced function symbols such that $\xi_{\mathcal{P}}$ is ε-free, linear, complete [5, Section 1.4], and *syntactically injective*—for each function symbol f, $\xi_{\mathcal{P}}(f)$ includes f and f does not appear in $\xi_{\mathcal{P}}(g)$ for any $g \neq f$;
3. we apply $\xi_{\mathcal{P}}$ to $\mathfrak{T}(\mathcal{P})$, defining $\mathfrak{T}_{np}$ as follows:

$$\mathfrak{T}_{np}(\mathcal{P}) = \{\xi_{\mathcal{P}}(\ell) \to \xi_{\mathcal{P}}(r)\ [\phi] \mid \ell \to r\ [\phi] \in \mathfrak{T}(\mathcal{P})\}$$

Note that the signature of $\mathfrak{T}_{np}(\mathcal{P})$ is not the same as that of $\mathfrak{T}(\mathcal{P})$, while their theory signatures are the same. Note also that the inverse relation of $\xi_{\mathcal{P}}$ is a mapping, i.e., $\xi_{\mathcal{P}}^{-1}$ is well defined. It follows from the properties of $\xi_{\mathcal{P}}$ that $\xi_{\mathcal{P}}$ is sound and complete for the reduction of $\mathfrak{T}(P)$: For any natural number n and any term s over the signature of $\mathfrak{T}(\mathcal{P})$,

- for any term t over the signature of $\mathfrak{T}(\mathcal{P})$, if $s \to^n_{\mathfrak{T}(\mathcal{P})} t$, then $\xi_{\mathcal{P}}(s) \to^n_{\mathfrak{T}_{np}(\mathcal{P})} \xi_{\mathcal{P}}(t)$, and
- for any term t' over the signature of $\mathfrak{T}_{np}(\mathcal{P})$, if $\xi_{\mathcal{P}}(s) \to^n_{\mathfrak{T}_{np}(\mathcal{P})} t'$, then $\xi_{\mathcal{P}}^{-1}(t')$ is defined and $s \to^n_{\mathfrak{T}(\mathcal{P})} \xi_{\mathcal{P}}^{-1}(t')$.

To define $\xi_{\mathcal{P}}$, we use the correspondence between the introduced auxiliary function symbols and statements in $\mathcal{P}$, which we do not want to keep. Though, the use of the correspondence is temporal, and we do not keep it after the application of $\mathfrak{T}_{np}$ to $\mathcal{P}$. In addition, thanks to the invertibility of $\xi_{\mathcal{P}}$, it is easy to obtain $\mathfrak{T}(\mathcal{P})$ from $\mathfrak{T}_{np}(\mathcal{P})$ without the correspondence or even $\xi_\mathcal{P}$. The application of $\xi_\mathcal{P}^{-1}$ to $\mathfrak{T}_{np}(\mathcal{P})$ is just a *flattening* for the nesting of the auxiliary function symbols.

An advantage of the above approach (i.e., the use of $\xi_\mathcal{P}$) is that we do not have to prove correctness of transforming SIMP programs into LCTRSs, and it suffices to prove soundness and completeness of $\xi_\mathcal{P}$ for the reduction of the LCTRS $\mathfrak{T}(\mathcal{P})$. In general, the correctness proof of transformations of programs into LCTRSs is very complicated. Our approach relies on the correctness of $\mathfrak{T}$. On the other hand, the soundness and completeness proof of $\xi_\mathcal{P}$ is not so complicated, and there already exist similar proofs in e.g., [14, Section 6.1].

2 Logically Constrained Term Rewrite Systems

In this section, we briefly recall logically constrained term rewrite systems [13, 7]. Familiarity with basic notions on term rewriting [2, 16] is assumed.

A *logically constrained term rewrite systems* (LCTRS, for short) is a set $\mathcal{R}$ of *constrained rewrite rules* $\ell \to r\ [\phi]$ over a signature Σ consisting of two signatures Σ_{theory} and Σ_{term}: $\Sigma = \Sigma_{theory} \cup \Sigma_{term}$. The theory signature Σ_{theory} defines built-in objects—values in $\mathcal{V}al$ ($\subseteq \Sigma_{theory}$) and operators—and is equipped with interpretations $\mathcal{I}$ and $\mathcal{J}$: For each theory sort ι, $\mathcal{I}$ specifies the universe of ι, and $\mathcal{J}$ assigns to a ground term with ι to an element in $\mathcal{I}(\iota)$. The non-theory signature Σ_{term} is a set of function symbols for user-defined functions and constructors. The constrained rewrite rules in $\mathcal{R}$ specify the behavior of some function symbols in Σ_{term}. We require that $\Sigma_{term} \cap \Sigma_{theory} \subseteq \mathcal{V}al$. 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 *theory term*. For ground theory terms, we define the *interpretation* $[\![\cdot]\!]$ as $[\![f(s_1,\ldots,s_n)]\!] = \mathcal{J}(f)([\![s_1]\!],\ldots,[\![s_n]\!])$. Note that for every ground theory term s, there is a unique value-constant c such that $[\![s]\!] = [\![c]\!]$. We may use infix notation for calculation symbols.

We typically choose a theory signature with $\Sigma_{theory} \supseteq \Sigma^{core}_{theory}$, where Σ^{core}_{theory} includes *bool*, a sort of *Booleans*, such that $\mathcal{V}al_{bool} = \{\mathsf{true},\mathsf{false}\}$ and $\mathcal{I}(bool) = \{\top,\bot\}$, $\Sigma^{core}_{theory} = \mathcal{V}al_{bool} \cup \{\wedge, \vee, \implies : bool \times bool \Rightarrow bool,\ \neg : bool \Rightarrow bool\} \cup \{=_\iota, \neq_\iota : \iota \times \iota \Rightarrow bool \mid \iota \text{ is a theory sort in } \Sigma_{theory}\}$, and $\mathcal{J}$ interprets these symbols as expected: $\mathcal{J}(\mathsf{true}) = \top$ and $\mathcal{J}(\mathsf{false}) = \bot$. We omit the sort subscripts from $=_\iota$ and $\neq_\iota$ when they are clear from context. A theory term with sort *bool* is called a *constraint*.

The standard integer signature Σ^{int}_{theory} is $\Sigma^{core}_{theory} \cup \{+,-,\times,\mathsf{exp},\mathsf{div},\mathsf{mod} : int \times int \Rightarrow int\} \cup \{\geq, > : int \times int \Rightarrow bool\} \cup \mathcal{V}al_{int}$ where $\mathcal{S} \supseteq \{int, bool\}$, $\mathcal{V}al_{int} = \{\mathsf{n} : int \mid n \in \mathbb{Z}\}$, $\mathcal{I}(int) = \mathbb{Z}$, and $\mathcal{J}(\mathsf{n}) = n$. Note that 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.

A *constrained rewrite rule* is a triple $\ell \to r\ [\phi]$ such that ℓ and r are terms of the same sort, ϕ is a constraint, and ℓ has the form $f(\ell_1,\ldots,\ell_n)$ that is not a theory term. If $\phi = \mathsf{true}$, then we may write $\ell \to r$. We define $\mathcal{LV}ar(\ell \to r\ [\phi])$ as $\mathcal{V}ar(\phi) \cup (\mathcal{V}ar(r) \setminus \mathcal{V}ar(\ell))$. We say that a substitution γ *respects* $\ell \to r\ [\phi]$ if $\mathcal{R}an(\gamma|_{\mathcal{LV}ar(\ell \to r\ [\phi])}) \subseteq \mathcal{V}al$ and $[\![\phi\gamma]\!] = \top$. Note that it is allowed to have $\mathcal{V}ar(r) \not\subseteq \mathcal{V}ar(\ell)$. Given a set $\mathcal{R}$ of constrained rewrite rules, $\mathcal{R}_{calc}$ denotes the set $\{f(x_1,\ldots,x_n) \to y\ [y = f(x_1,\ldots,x_n)] \mid f \in \Sigma_{theory} \setminus \mathcal{V}al,\ x_1,\ldots,x_n,y \in \mathcal{V}\}$. The elements of $\mathcal{R}_{calc}$ are also called constrained rewrite rules (or *calculation rules*) even though their left-hand side is a theory term. The *rewrite relation* $\to_\mathcal{R}$ is a binary relation on terms, defined as follows: for a term s, $s[\ell\gamma]_p \to_\mathcal{R} s[r\gamma]_p$ if $\ell \to r\ [\phi] \in \mathcal{R} \cup \mathcal{R}_{calc}$ and γ respects $\ell \to r\ [\phi]$.

Example 2.1 Let $\mathcal{S} = \{int, bool\}$, $\Sigma = \Sigma_{term} \cup \Sigma^{int}_{theory}$ and $\Sigma_{term} = \{\mathsf{fact} : int \Rightarrow int\} \cup \{\mathsf{n} : int \mid n \in \mathbb{Z}\}$. To implement an LCTRS calculating the *factorial* function over $\mathbb{Z}$, we use the signature Σ above and the following LCTRS:

$$\mathcal{R}_{\mathsf{fact}} = \left\{ \begin{array}{ll} \quad\mathsf{fact}(x) \to \mathsf{subfact}(x,1) & \\ \mathsf{subfact}(x,y) \to y & [0 \geq x] \\ \mathsf{subfact}(x,y) \to \mathsf{subfact}(x',y') & [x > 0 \wedge x' = x-1 \wedge y' = x \times y] \end{array} \right\}$$

The term $\mathsf{fact}(3)$ is reduced by $\mathcal{R}_{\mathsf{fact}}$ to 6: $\mathsf{fact}(3) \to_{\mathcal{R}_{\mathsf{fact}}} \mathsf{subfact}(3,1) \to_{\mathcal{R}_{\mathsf{fact}}} \mathsf{subfact}(2,3) \to_{\mathcal{R}_{\mathsf{fact}}} \mathsf{subfact}(1,6) \to_{\mathcal{R}_{\mathsf{fact}}} \mathsf{subfact}(0,6) \to_{\mathcal{R}_{\mathsf{fact}}} 6$.

3 A Nesting-Preserving Transformation

In this section, using $\mathcal{P}_{\texttt{loop}}$ and $\mathcal{R}_{\texttt{loop}}$ in Program 1 and Figure 1, respectively, we explain an overview of our nesting-preserving transformation and formulate the transformation using tree homomorphims.

We assume that each statement in a SIMP program $\mathcal{P}$ has a unique label ρ, and we use line numbers as such labels. We consider the sequential composition "`;`" as the list concatenation, and exclude it from the nesting of statements. The `while` statements on Lines 4–15, 6–13, and 8–11 have the nesting.

We first explain the nesting-preserving transformation of the outermost `while` statement on Lines 4–15. Any configuration at the beginning of Line 4 is of the form $\mathsf{loop}_4(n, ret, i)$. During the execution of the `while` statement, we keep the symbol loop_4, and thus, the symbol does not reduce to other symbols until completing the execution of the corresponding `while` statement. Instead, we add an extra argument to loop_4, and the argument is used to represent the execution of the body of the `while` statement as follows:

- To represent the entry location of the body, we introduce a fresh constant ent.
- We replace $\mathsf{loop}_4(t_{\texttt{n}}, t_{\texttt{ret}}, t_{\texttt{i}})$ in $\mathcal{R}_{\texttt{loop}}$ by $\mathsf{loop}_4(t_{\texttt{n}}, t_{\texttt{ret}}, t_{\texttt{i}}, \mathsf{ent})$.
- The location just before the local-variable declaration `int j = 0;` on Line 5 is represented by loop_5, and we put it into the last argument of loop_4. To this end, we replace $\mathsf{loop}_5(t_{\texttt{n}}, t_{\texttt{ret}}, t_{\texttt{i}})$ in $\mathcal{R}_{\texttt{loop}}$ by $\mathsf{loop}_4(t_{\texttt{n}}, t_{\texttt{ret}}, t_{\texttt{i}}, \mathsf{loop}_5)$.
- The location at the beginning of Line 6 is represented by $\mathsf{loop}_6(t_{\texttt{j}}, \mathsf{ent})$ which is again stored as the last argument of loop_4. The new loop_6 keeps the value stored in program variable `j` instead of loop_4 in $\mathcal{R}_{\texttt{loop}}$. In addition, since loop_6 represents a `while` statement with the nesting of statements, we make loop_6 have the second argument for the execution of the body of the corresponding `while` statement. To this end, we replace $\mathsf{loop}_6(t_{\texttt{n}}, t_{\texttt{ret}}, t_{\texttt{i}}, t_{\texttt{j}})$ in $\mathcal{R}_{\texttt{loop}}$ by $\mathsf{loop}_4(t_{\texttt{n}}, t_{\texttt{ret}}, t_{\texttt{i}}, \mathsf{loop}_6(t_{\texttt{j}}, \mathsf{ent}))$.
- We repeat the same for the inner `while` statements represented by loop_6 and loop_8.
- The location just before the assignment `i = i + 1;` on Line 14 is represented by loop_{14}, and we put it into the last argument of loop_4. To this end, we replace $\mathsf{loop}_{14}(t_{\texttt{n}}, t_{\texttt{ret}}, t_{\texttt{i}})$ in $\mathcal{R}_{\texttt{loop}}$ by $\mathsf{loop}_4(t_{\texttt{n}}, t_{\texttt{ret}}, t_{\texttt{i}}, \mathsf{loop}_{14})$.

The resulting LCTRS is illustrated in Figure 3.

What we did to preserve the nesting of statements is just a replacement of terms, and can be formulated by a *tree homomorphism*.

Definition 3.1 (tree homomorphism [5, Section 1.4]) *Let Σ and Σ' be signatures, possibly not disjoint. For each $n \geq 0$, we define the set $\mathcal{X}_n$ of n variables such that $\mathcal{X}_n = \{x_1, \dots, x_n\} \subseteq \mathcal{V}$. Let ξ_Σ be a mapping which associates with each $f \in \Sigma$ of arity n a term in $T(\Sigma', \mathcal{X}_n)$: $\xi_\Sigma(f) \in T(\Sigma', \mathcal{X}_n)$. The* tree homomorphism *$\xi : T(\Sigma, \mathcal{V}) \to T(\Sigma', \mathcal{V})$ determined by ξ_Σ is inductively defined as follows:*

$$\left\{\begin{array}{rcll}
\mathsf{loop}(n) & \to & \mathsf{loop}_2(n) & \\
\mathsf{loop}_2(n) & \to & \mathsf{loop}_3(n,0) & \\
\mathsf{loop}_3(n,ret) & \to & \mathsf{loop}_4(n,ret,0,\mathsf{ent}) & \\
\mathsf{loop}_4(n,ret,i,\mathsf{ent}) & \to & \mathsf{loop}_4(n,ret,i,\mathsf{loop}_5) & [\ \ i<n\ \ ] \\
\mathsf{loop}_4(n,ret,i,\mathsf{ent}) & \to & \mathsf{loop}_{16}(n,ret,i) & [\neg(i<n)] \\
\mathsf{loop}_4(n,ret,i,\mathsf{loop}_5) & \to & \mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(0,\mathsf{ent})) & \\
\mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{ent})) & \to & \mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{loop}_7)) & [\ \ j<i\ \ ] \\
\mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{ent})) & \to & \mathsf{loop}_4(n,ret,i,\mathsf{loop}_{14}) & [\neg(j<i)] \\
\mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{loop}_7)) & \to & \mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{loop}_8(0,\mathsf{ent}))) & \\
\mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{loop}_8(k,\mathsf{ent}))) & \to & \mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{loop}_8(k,\mathsf{loop}_9))) & [\ \ k<j\ \ ] \\
\mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{loop}_8(k,\mathsf{ent}))) & \to & \mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{loop}_{12})) & [\neg(k<j)] \\
\mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{loop}_8(k,\mathsf{loop}_9))) & \to & \mathsf{loop}_4(n,ret+1,i,\mathsf{loop}_6(j,\mathsf{loop}_8(k,\mathsf{loop}_{10}))) & \\
\mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{loop}_8(k,\mathsf{loop}_{10}))) & \to & \mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{loop}_8(k+1,\mathsf{ent}))) & \\
\mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{loop}_{12})) & \to & \mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j+1,\mathsf{ent})) & \\
\mathsf{loop}_4(n,ret,i,\mathsf{loop}_{14}) & \to & \mathsf{loop}_4(n,ret,i+1,\mathsf{ent}) & \\
\mathsf{loop}_{16}(n,ret,i) & \to & \mathsf{return}(ret) &
\end{array}\right\}$$

Figure 3: the nesting-preserved LCTRS obtained from Program 1 [7].

- $\xi(x) = x$ *for a variable* $x \in \mathcal{V}$*, and*
- $\xi(f(t_1,\dots,t_n)) = \xi_\Sigma(f)\{x_i \mapsto \xi(t_i) \mid 1 \le i \le n\}$ *for an n-ary function symbol* $f \in \Sigma$*.*

A tree homomorphism ξ *determined by* ξ_Σ *is called*

- linear *if* $\xi_\Sigma(f)$ *is a linear term for all* $f \in \Sigma$*,*
- ε-free *if* $\xi_\Sigma(f)$ *is not a variable for any* $f \in \Sigma$*, and*
- complete *if* $\mathcal{V}ar(\xi_\Sigma(f)) = \mathcal{X}_n$ *for all n-ary* $f \in \Sigma$*.*

Note that not all tree homomorphisms are surjective.

To consider injective tree homomorphisms, we show a syntactic sufficient condition for injectivity of tree homomorphisms.

Definition 3.2 *For a term t, we denote by* $\mathcal{S}yms(t)$ *the set of function symbols in t. A complete tree homomorphism* ξ *determined by* ξ_Σ *for* Σ *is called* syntactically injective *if for each* $f \in \Sigma$, $\mathcal{S}yms(\xi_\Sigma(f)) \setminus \left(\bigcup_{g\in\Sigma, g\neq f} \mathcal{S}yms(\xi_\Sigma(g))\right) \neq \emptyset$*, i.e.,* $\xi_\Sigma(f)$ *includes a* unique *function symbol which is not introduced by any other symbol g in* Σ*.*

Note that a syntactically injective tree homomorphism is ε-free.

Proposition 3.3 *A syntactically injective tree homomorphism is injective.*

Finally, we formulate our nesting-preserving transformation by means of tree homomorphisms. Let $\mathfrak{T}$ be the transformation in [7] of a SIMP program into an LCTRS, where for each statement with label ρ in the definition of a function f, a fresh function symbol is introduced, which is denoted by f_ρ. Such a function symbol can be considered a location just before execution the corresponding statement. We represent the nesting of statements at a statement ρ by a list of labels which is denoted by $\mathcal{N}esting(\rho)$:

If ρ is not a substatement of any other statement, then $\mathcal{N}esting(\rho) = \varepsilon$; If ρ is a direct substatement of ρ',[4] then $\mathcal{N}esting(\rho) = \mathcal{N}esting(\rho')@[\rho']$.

Example 3.4 For $\mathcal{P}_{\texttt{loop}}$ in Program 1, $\mathcal{N}esting(2) = \mathcal{N}esting(3) = \mathcal{N}esting(4) = \mathcal{N}esting(16) = \varepsilon$, $\mathcal{N}esting(5) = \mathcal{N}esting(6) = \mathcal{N}esting(14) = [4]$, $\mathcal{N}esting(7) = \mathcal{N}esting(8) = \mathcal{N}esting(12) = [4,6]$, and $\mathcal{N}esting(9) = \mathcal{N}esting(10) = [4,6,8]$.

Definition 3.5 (a nesting-preserving transformation $\mathfrak{T}_{np}$) *Let $\mathcal{P}$ be a SIMP program. We introduce a fresh constant* ent *to the signature of $\mathfrak{T}(\mathcal{P})$. We define a mapping ξ_Σ as follows:*

- $\xi_\Sigma(f) = f(x_1,\ldots,x_n)$ *for any n-ary function symbol f which is defined in $\mathcal{P}$,*
- $\xi_\Sigma(f_\rho) = f_{\rho_1}(x_1,\ldots,x_{n_1}, f_{\rho_2}(x_{n_1+1},\ldots,x_{n_2}, f_{\rho_3}(\ldots, f_{\rho_k}(x_{n_{k-1}+1},\ldots,x_{n_k},t_\rho)\ldots)))$ *for any n-ary function symbol f_ρ that is introduced by $\mathfrak{T}$, where*
  - $\mathcal{N}esting(\rho) = [\rho_1,\rho_2,\ldots,\rho_k]$,
  - $f_\rho(y_1,\ldots,y_n)$ *appears in $\mathfrak{T}(\mathcal{P})$ as an left-hand side such that*
    - $y_1,\ldots,y_n$ *are the variables in the definition of f in $\mathcal{P}$,*
    - y_i *is declared before y_j for any i,j with $i<j$,*
    - $y_1,\ldots,y_{n_1}$ *are the local variables of $\mathcal{P}$, which are accessible at ρ_1, and*
    - $y_1,\ldots,y_{n_j}$ *with $1<j<k$ are the local variables of $\mathcal{P}$, which are accessible at ρ_j,*

    and
  - *if ρ has a substatement (i.e., a conditional or loop statement), then $t_\rho = f_\rho(x_{n_k+1},\ldots,x_n,\mathsf{ent})$, and otherwise, $t_\rho = f_\rho(x_{n_k+1},\ldots,x_n)$.*
- $\xi_\Sigma(\mathsf{return}) = \mathsf{return}(x_1)$, *and*
- $\xi_\Sigma(g) = g(x_1,\ldots,x_n)$ *for any n-ary theory symbol g.*

Let $\xi_\mathcal{P}$ be the tree homomorphism determined by ξ_Σ above. We define a transformation $\mathfrak{T}_{np}$ of $\mathcal{P}$ into an LCTRS as follows:

$$\mathfrak{T}_{np}(\mathcal{P}) = \{\xi_\mathcal{P}(\ell) \to \xi_\mathcal{P}(r)\ [\phi] \mid \ell \to r\ [\phi] \in \mathfrak{T}(\mathcal{P})\}$$

By definition, it is clear that $\xi_\mathcal{P}$ in Definition 3.5 is ε-free, linear, complete, and syntactically injective: for any n-ary function symbol h, $\xi_\Sigma(h)$ is a linear non-variable term, $\mathcal{V}ar(\xi_\Sigma(h)) = \mathcal{X}_n$, and $h \in \mathcal{S}yms(\xi_\Sigma(h)) \setminus (\bigvee_{h'\in\Sigma,h'\neq h}\mathcal{S}yms(\xi_\Sigma(h')))$—$\xi_\Sigma(h)$ contains a function symbol that does not appear in $\bigvee_{h'\in\Sigma,h'\neq h}\mathcal{S}yms(\xi_\Sigma(h'))$.

Example 3.6 Consider the SIMP program $\mathcal{P}_{\texttt{loop}}$ and the LCTRS $\mathcal{R}_{\texttt{loop}}$ in Program 1 and Figure 1, respectively, again. The mapping ξ_Σ in Definition 3.5 is defined for $\mathcal{R}_{\texttt{loop}}$ as follows:

- $\xi_\Sigma(\mathsf{loop}) = \mathsf{loop}(x_1)$,
- $\xi_\Sigma(\mathsf{loop}_2) = \mathsf{loop}_2(x_1)$,
- $\xi_\Sigma(\mathsf{loop}_3) = \mathsf{loop}_3(x_1,x_2)$,
- $\xi_\Sigma(\mathsf{loop}_4) = \mathsf{loop}_4(x_1,x_2,x_3,\mathsf{ent})$,
- $\xi_\Sigma(\mathsf{loop}_5) = \mathsf{loop}_4(x_1,x_2,x_3,\mathsf{loop}_5)$,
- $\xi_\Sigma(\mathsf{loop}_6) = \mathsf{loop}_4(x_1,x_2,x_3,\mathsf{loop}_6(x_4,\mathsf{ent}))$,

[4]There is no other statement ρ'' such that ρ is a substatement of ρ'' and ρ'' is a substatement of ρ'.

- $\xi_\Sigma(\mathsf{loop}_7) = \mathsf{loop}_4(x_1,x_2,x_3,\mathsf{loop}_6(x_4,\mathsf{loop}_7))$,
- $\xi_\Sigma(\mathsf{loop}_8) = \mathsf{loop}_4(x_1,x_2,x_3,\mathsf{loop}_6(x_4,\mathsf{loop}_8(x_5,\mathsf{ent})))$,
- $\xi_\Sigma(\mathsf{loop}_9) = \mathsf{loop}_4(x_1,x_2,x_3,\mathsf{loop}_6(x_4,\mathsf{loop}_8(x_5,\mathsf{loop}_9)))$,
- $\xi_\Sigma(\mathsf{loop}_{10}) = \mathsf{loop}_4(x_1,x_2,x_3,\mathsf{loop}_6(x_4,\mathsf{loop}_8(x_5,\mathsf{loop}_{10})))$,
- $\xi_\Sigma(\mathsf{loop}_{12}) = \mathsf{loop}_4(x_1,x_2,x_3,\mathsf{loop}_6(x_4,\mathsf{loop}_{12}))$,
- $\xi_\Sigma(\mathsf{loop}_{14}) = \mathsf{loop}_4(x_1,x_2,x_3,\mathsf{loop}_{14})$,
- $\xi_\Sigma(\mathsf{loop}_{16}) = \mathsf{loop}_{16}(x_1,x_2,x_3)$, and
- $\xi_\Sigma(\mathsf{return}) = \mathsf{return}(x_1)$.

Let $\xi_{\mathcal{P}_{\mathtt{loop}}}$ be the tree homomorphism determined by ξ_Σ above. Then, $\xi_{\mathcal{P}_{\mathtt{loop}}}(\mathcal{R}_{\mathtt{loop}})$ is equivalent to the LCTRS in Figure 3. The tree homomorphism is also applicable to the simplified LCTRS $\mathcal{R}'_{\mathtt{loop}}$:

$$\xi_{\mathcal{P}_{\mathtt{loop}}}(\mathcal{R}'_{\mathtt{loop}}) =$$
$$\left\{\begin{array}{rcll}
\mathsf{loop}(n) & \to & \mathsf{loop}_4(n,0,0,\mathsf{ent}) & \\
\mathsf{loop}_4(n,ret,i,\mathsf{ent}) & \to & \mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(0,\mathsf{ent})) & [\ i<n\ ] \\
\mathsf{loop}_4(n,ret,i,\mathsf{ent}) & \to & \mathsf{return}(ret) & [\neg(i<n)] \\
\mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{ent})) & \to & \mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{loop}_8(0,\mathsf{ent}))) & [\ j<i\ ] \\
\mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{ent})) & \to & \mathsf{loop}_4(n,ret,i+1,\mathsf{ent}) & [\neg(j<i)] \\
\mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{loop}_8(k,\mathsf{ent}))) & \to & \mathsf{loop}_4(n,ret+1,i,\mathsf{loop}_6(j,\mathsf{loop}_8(k+1,\mathsf{ent}))) & [\ k<j\ ] \\
\mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j,\mathsf{loop}_8(k,\mathsf{ent}))) & \to & \mathsf{loop}_4(n,ret,i,\mathsf{loop}_6(j+1,\mathsf{ent})) & [\neg(k<j)]
\end{array}\right\}$$

Note that the LCTRS in Figure 3 can be simplified to $\xi_{\mathcal{P}_{\mathtt{loop}}}(\mathcal{R}'_{\mathtt{loop}})$ as well as the simplification of $\mathcal{R}_{\mathtt{loop}}$ to $\mathcal{R}'_{\mathtt{loop}}$.

4 Reduction Preservation by Tree Homomorphisms

In this section, we show that a syntactically injective linear tree homomorphism is sound and complete for the reduction of an LCTRS $\mathcal{R}$ (cf. [14, Section 6.1]).

Let Σ and Σ' be signatures that have the same theory signature, i.e., $\Sigma = \Sigma_{term} \cup \Sigma_{theory}$, $\Sigma' = \Sigma'_{term} \cup \Sigma_{theory}$, and $\Sigma_{term} \cap \Sigma_{theory} = \Sigma'_{term} \cap \Sigma_{theory}$. Let $\mathcal{R}$ be an LCTRS over Σ, and ξ a syntactically injective linear tree homomorphism determined by a mapping $\xi_\Sigma : \Sigma \to T(\Sigma',\mathcal{V})$ such that

- $\xi_\Sigma(f) \in T(\Sigma' \setminus \Sigma_{theory}, \mathcal{X}_n)$ for any n-ary function symbol $f \in \Sigma_{term} \setminus \Sigma_{theory}$, and
- $\xi_\Sigma(g) = g(x_1,\dots,x_n)$ for any n-ary theory symbol $g \in \Sigma_{theory}$.

Note that $\xi_\Sigma(t) = t$ for any theory term $t \in T(\Sigma_{theory},\mathcal{V})$. Let γ be a substitution over Σ. We denote the substitutions $\{x \mapsto \xi(x\gamma) \mid x \in \mathcal{D}om(\gamma)\}$ and $\{x \mapsto \xi^{-1}(x\gamma) \mid x \in \mathcal{D}om(\gamma)\}$ by γ_ξ and $\gamma_{\xi^{-1}}$, respectively. By definition, it is clear that for every rewrite rule $\ell \to r\ [\phi]$ over Σ, $\xi(\ell) \to \xi(r)\ [\phi]$ is a constrained rewrite rule over Σ'. We denote the LCTRS $\{\xi(\ell) \to \xi(r)\ [\phi] \mid \ell \to r\ [\phi] \in \mathcal{R}\}$ by $\xi(\mathcal{R})$. We further assume that

- $\xi_\Sigma(f)$ is basic w.r.t. $\xi(\mathcal{R})$ for any defined symbol f of $\mathcal{R}$, and
- $\xi_\Sigma(f)$ is a constructor term of $\xi(\mathcal{R})$ for any constructor f of $\mathcal{R}$.

The linearity and injectivity of ξ distributes the application of ξ and ξ^{-1} to subterms and substitutions, and implies soundness and completeness for $\to_{\mathcal{R}}$.

Lemma 4.1 *Let s,t be terms in $T(\Sigma,\mathcal{V})$, p a position of s, and γ a substitution. Then, $\xi(s[t\gamma]_p) = (\xi(s))[(\xi(t))\gamma_\xi]_{p'}$ for some position p' of $\xi(s)$.*

Lemma 4.2 *Let s,t be terms in $T(\Sigma,\mathcal{V})$, s' a term in $T(\Sigma',\mathcal{V})$, p' a position of s', and γ' a substitution over Σ'. If $\xi(s) = s'[(\xi(t))\gamma']_{p'}$, then $s = (\xi^{-1}(s'))[t\gamma'_{\xi^{-1}}]_p$ for some position p of s.*

Theorem 4.3 (soundness and completeness of ξ for $\to_\mathcal{R}$) *For any $n \geq 0$ and any term $s \in T(\Sigma,\mathcal{V})$, both of the following hold:*

- *For any term $t \in T(\Sigma,\mathcal{V})$, if $s \to^n_\mathcal{R} t$, then $\xi(s) \to^n_{\xi(\mathcal{R})} \xi(t)$, and*
- *for any term $t' \in T(\Sigma',\mathcal{V})$, if $\xi(s) \to^n_{\xi(\mathcal{R})} t'$, then $\xi^{-1}(t')$ is defined and $s \to^n_\mathcal{R} \xi^{-1}(t')$.*

5 Conclusion

In this paper, we proposed a nesting-preserving transformation of a SIMP program $\mathcal{P}$ into an LCTRS by proposing a construction of a tree homomorphism $\xi_\mathcal{P}$ which is used as a post-process of the transformation $\mathfrak{T}$ in previous work. To be more precise, the transformation $\mathfrak{T}_{np}$ is the composition of $\mathfrak{T}$ and $\xi_\mathcal{P}$. The tree homomorphism $\xi_\mathcal{P}$ is ε-free, linear, complete, and syntactically injective, and hence injective. The inverse of $\xi_\mathcal{P}$ can be considered a flattening, and the LCTRS obtained from $\mathcal{P}$ by $\mathfrak{T}_{np}$ can be transformed back into the LCTRS $\mathfrak{T}(\mathcal{P})$. The approach in this paper can be extended to other kinds of LCTRSs, e.g., LCTRSs with bit-vector arithmetic [9, 12].

The language SIMP and its extension have no statement of jumps such as `goto`, `continue`, and `break`, but the transformations $\mathfrak{T}$ and $\mathfrak{T}_{np}$ can be extended to such jumps. The nest-preserving one $\mathfrak{T}_{np}$ would be more appropriate for the extension to jumps than $\mathfrak{T}$ because all the local variables in a block are arguments of the nested symbol for the block and thus, we can easily drop the local variables in exiting the block by means of a jump.

The aim of the nesting-preserving transformation is to help analyses of the transformed LCTRSs. However, we have not measured the usefulness of the preservation of nesting in any analysis. As our future work, we will evaluate the nesting-preserving transformation by means of experiments of e.g., proving termination of LCTRSs with bit-vector arithmetic.

References

[1] Frances E. Allen (1970): *Control flow analysis*. In: *Proc. Symposium on Compiler Optimization 1970*, ACM, pp. 1–19, doi:10.1145/800028.808479.

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

[3] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf & Nir Piterman (2016): *T2: Temporal Property Verification*. In: *Proc. TACAS 2016*, *LNCS* 9636, Springer, pp. 387–393, doi:10.1007/978-3-662-49674-9_22.

[4] Ştefan Ciobâcă & Dorel Lucanu (2018): *A Coinductive Approach to Proving Reachability Properties in Logically Constrained Term Rewriting Systems*. In: *Proc. IJCAR 2018*, *LNCS* 10900, Springer, pp. 295–311, doi:10.1007/978-3-319-94205-6_20.

[5] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison & Marc Tommasi (2007): *Tree Automata Techniques and Applications*. Available on: `http://www.grappa.univ-lille3.fr/tata`.

[6] Maribel Fernández (2014): *Programming Languages and Operational Semantics – A Concise Overview*. UTiCS, Springer, doi:10.1007/978-1-4471-6368-8.

[7] Carsten Fuhs, Cynthia Kop & Naoki Nishida (2017): *Verifying Procedural Programs via Constrained Rewriting Induction*. *ACM Trans. Comput. Log.* 18(2), pp. 14:1–14:50, doi:10.1145/3060143.

[8] Yoshiaki Kanazawa & Naoki Nishida (2019): *On Transforming Functions Accessing Global Variables into Logically Constrained Term Rewriting Systems*. In: *Proc. WPTE 2018*, *EPTCS* 289, Open Publishing Association, pp. 34–52.

[9] Yoshiaki Kanazawa, Naoki Nishida & Masahiko Sakai (2019): *On Representation of Structures and Unions in Logically Constrained Rewriting*. IEICE Technical Report SS2018-38, IEICE. Vol. 118, No. 385, pp. 67–72, in Japanese.

[10] Misaki Kojima & Naoki Nishida (2022): *On Reducing Non-Occurrence of Specified Runtime Errors to All-Path Reachability Problems of Constrained Rewriting*. In: *Informal Proc. WPTE 2022*, pp. 1–16. Available at `https://easychair.org/publications/preprint/TM7q`.

[11] Misaki Kojima & Naoki Nishida (2023): *From Starvation Freedom to All-Path Reachability Problems in Constrained Rewriting*. In: *Proc. PADL 2023*, *LNCS* 13880, Springer Nature Switzerland, pp. 161–179, doi:10.1007/978-3-031-24841-2_11.

[12] Misaki Kojima, Naoki Nishida & Yutaka Matsubara (2020): *Transforming Concurrent Programs with Semaphores into Logically Constrained Term Rewrite Systems*. In: *Informal Proc. WPTE 2020*, pp. 1–12.

[13] Cynthia Kop & Naoki Nishida (2013): *Term Rewriting with Logical Constraints*. In: *Proc. FroCoS 2013*, *LNCS* 8152, Springer, pp. 343–358, doi:10.1007/978-3-642-40885-4_24.

[14] Naoki Nishida, Masahiko Sakai & Toshiki Sakabe (2012): *Soundness of Unravelings for Conditional Term Rewriting Systems via Ultra-Properties Related to Linearity*. *Log. Methods Comput. Sci.* 8(3-4), pp. 1–49.

[15] Naoki Nishida & Sarah Winkler (2018): *Loop Detection by Logically Constrained Term Rewriting*. In: *Proc. VSTTE 2018*, *LNCS* 11294, Springer, pp. 309–321, doi:10.1007/978-3-030-03592-1_18.

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

[17] Sarah Winkler & Aart Middeldorp (2018): *Completion for Logically Constrained Rewriting*. In: *Proc. FSCD 2018*, *LIPIcs* 108, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 30:1–30:18, doi:10.4230/LIPIcs.FSCD.2018.30.