Transforming Concurrent Programs with Semaphores into Logically Constrained Term Rewrite Systems*

Misaki Kojima
Graduate School of Informatics
Nagoya University
Nagoya, Japan
k-misaki@trs.css.i.nagoya-u.ac.jp

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

Yutaka Matsubara
Graduate School of Informatics
Nagoya University
Nagoya, Japan
yutaka@ertl.jp

In this paper, we show a transformation of concurrent programs with exclusive control operated by means of semaphores into logically constrained term rewrite systems (LCTRSs, for short), aiming at modeling concurrent programs by LCTRSs. To this end, we first expand arguments of a function symbol that has been introduced to represent configurations of sequential execution so as to store all executed processes. Next, we show how to represent operations for semaphores by rewrite rules, and show a transformation of concurrent programs with semaphores into LCTRSs. We manage waiting queues for semaphores by means of a so-called turn waiting system with number tickets, avoiding recursive data structures such as lists.

1 Introduction

Recently, approaches to program verification by means of logically constrained term rewrite systems (LCTRSs, for short) [6] are well investigated [3, 12, 2, 8, 4, 5]. LCTRSs are known to be useful as computation models of not only functional but also imperative programs. Especially, equivalence checking via LCTRSs is useful to ensure correctness of functions (cf. [3]). Here, equivalence of two functions means that for every input, the functions return the same output or end with the same configuration. On the other hand, previous work [3, 4, 5] targets only sequential programs.

Regarding practical use of computers, we would like to verify that a sequential program is equivalent to its concurrent version or a re-implemented program executing some functions in parallel. When a sequential program has been verified to be correct thanks to test, formal methods, and/or operational experience, it may be possible to use the correctness to verify correctness of its concurrent version by means of equivalence between the sequential and concurrent versions, instead of verifying the concurrent version from scratch. Therefore, the application of an equivalence verification method in previous work to concurrent programs would be useful for program verification. However, to apply the method for sequential programs to concurrent ones, we need to transform constructs for concurrency, which are not in sequential programs, into LCTRSs.

This research aims at transforming concurrent programs into LCTRSs in order to apply verification methods for LCTRSs to equivalence verification of sequential and concurrent automotive embedded systems written in C. In this paper, as a first step, we show how to represent exclusive control, which is not handled in sequential programs, by LCTRSs. In particular, we transform concurrent programs with exclusive control operated by means of semaphores into LCTRSs. Here, we assume that the number of processes is fixed. An advantage of this approach is that to verify automotive embedded systems, we can use rewriting techniques for verification of, e.g., termination, reachability, and equivalence.

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

We target automotive embedded systems, and thus, there may be many layers—from software to hardware—in verifying programs. For example, one may think that smaller primitives should be taken into account or there are more primitives to be taken into account. Though, our present goal is to verify application programs from algorithmic point of view. For this reason, we do not take hardware features into account, and we assume that atomic operations are executed as we expect.

Our transformation follows the following four policies:

- *Each atomic operation is represented by a single rewrite rule.* Reduction steps by LCTRS correspond to the application of single rewrite rules. Each atomic operation should be represented in order that any other operations (processes) cannot interrupt the atomic operation until the execution of the atomic operation finishes. Therefore, atomic operations are modeled by single rewrite rules.
- *Each rewrite rule represents an action of at most one process.*[1] It is usual that the more rules we have, the longer it takes to verify LCTRSs. If a rewrite rule refers to states of several processes and represents the action of one or more processes simultaneously, then the number of rewrite rules would increase exponentially according to combinations of processes. However, most of such rules represent the same actions. For efficiency of verification, we would like to keep down the increase of rules. For this reason, each rewrite rule is expected to refer to the state of (at most) a single process and represents an action of the process.
- *Recursive data structures such as lists are not adopted for exclusive control. Rewriting induction* [10, 3] used as a method for equivalence verification, makes a case analysis w.r.t. a *reduction-complete* position which ensures the exhaustiveness of the case analysis. Decidability of reduction-complete positions is not known yet. Even for decidable constraints, it is not so easy to decide whether a position is reduction-complete along with recursive data structures such as lists; it may be undecidable or not known yet. For this reason, we must facilitate a verification process of rewriting induction by avoiding the use of recursive data structure as much as possible. Therefore, we do not adopt lists for waiting queues of semaphores. Note that we do not prohibit to use any recursive data structure at all. For example, we use stacks for function calls because such stacks do not prevent us from checking reduction-completeness of LCTRSs obtained from imperative programs.
- *All basic data in programs are represented as bit vectors.* Previous work [5] represents all built-in data in programs as bit vectors in order to model programs more precisely. For example, 32-bit integers are represented as bit vectors with length 32. This paper follows the representation, modeling metadata for managing semaphores as bit vectors.

To represent configurations of sequential programs, previous work [3, 4, 5] introduces a function symbol env for the environment of execution. The function symbol takes as arguments both a state of executing a function and values stored in global variables. To model concurrent programs by LCTRSs, we also use the function symbol env to represent configurations (Section 3). However, we increase its arguments in order for the symbol to include states of several processes which are executed simultaneously, along with values stored in global variables.

We represent operations for semaphores by rewrite rules as well as assignment statements. For the sake of simplicity, this paper assumes that processes requiring a semaphore stand by in the order of request. To represent and handle a waiting queue without using any lists, we adopt a so-called *turn*

[1] A rewrite rule may represent a built-in or user-defined computation, that is, the rule does not represent any action of processes.

waiting system with number tickets; we make each process have a value corresponding to a number ticket for the queue by wrapping a state *st* of the process and the ticket *tckt* in a common constructor p: $\mathsf{p}(st, tckt)$; we also make the environment have both a "display board" to permit processes to acquire the semaphore and a "ticket machine" to issue number tickets, where the display board and ticket machine are implemented as counters. A process is given a number ticket when it needs to wait and waits until the number in the ticket gets called (i.e., matches the value of the display board). In this approach, it is not necessary to refer to the states of any other processes, and thus a rewrite rule represents an action of a single process. We illustrate how to transform a concurrent program with a binary semaphore into an LCTRS (Section 4). Note that the turn waiting system used in LCTRSs is not a physical implementation.

2 Preliminaries

In this section, we recall bit vectors [7, Chapter 6] and LCTRSs [6, 3]. Familiarity with basic notions on term rewriting [1, 9] is assumed.

2.1 Bit Vector Logic

We introduce the bit vector logic to use it as built-ins for LCTRSs.

A *bit vector* with length n is defined as a mapping from $\{0,\dots,n-1\}$ to $\{0,1\}$. The set of bit vectors with length n is denoted by $\mathbb{BV}_n$. For a bit vector $a \in \mathbb{BV}_n$, a_i denotes the i-th bit, i.e., $a_i = a(i)$. We represent bit vectors as binary numerals (sequences of $0,1$) such as $a_{n-1}a_{n-2}\dots a_0$ for $a \in \mathbb{BV}_n$, and furthermore we follow the notation adopted by SMT-LIB[2] in order to distinguish decimal numbers: a constant bit vector $c \in \mathbb{BV}_n$ is denoted by #bc, where c is written as a binary numeral. For example, binary numeral 101 is denoted by #b101 as a bit vector, being distinguished from decimal number 101. Note that a_0 and a_{n-1} are the least and most significant bits of a, respectively.

Bit vector operators are defined for fixed-size vectors. For example, the bitwise-and operator over $\mathbb{BV}_n$ is denoted by $\&_n$. Though, we abbreviate $\&_n$ to &, abusing it for the bitwise-and operators for bit vectors with different lengths. We follow the usual semantics of arithmetic and bitwise operators $(+_S, +_U, \ll, \&, \sim, \dots)$ for bit vectors. Note that for arithmetic operators, we specify the encoding S or U, where S (signed) and U (unsigned) indicates two's complement and binary encodings, respectively.

2.2 Logically Constrained Rewriting

Let $\mathcal{S}$ be a set of *basic sorts* and $\mathcal{V}$ a (countably infinite) set of *variables*, each of which is equipped with a basic sort. A *signature* Σ disjoint from $\mathcal{V}$ is a set of *function symbols* f, each of which is equipped with a *sort declaration* $\iota_1 \times \dots \times \iota_n \Rightarrow \iota$, written as $f : \iota_1 \times \dots \times \iota_n \Rightarrow \iota$, where $\iota_1,\dots,\iota_n,\iota \in \mathcal{S}$. In the rest of this section, we fix $\mathcal{S}$, Σ, and $\mathcal{V}$.

We denote the set of well-sorted *terms* over Σ and $\mathcal{V}$ by $T(\Sigma,\mathcal{V})$. We may write $s : \iota$ if s has basic sort ι. The set of variables occurring in $s_1,\dots,s_n$ is denoted by $\mathcal{V}ar(s_1,\dots,s_n)$. 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 restriction of γ w.r.t. a set X of variables is denoted by $\gamma|_X$: $\gamma|_X(x) = \gamma(x)$ if $x \in X$, and otherwise $\gamma|_X(x) = x$. The application of γ to term s is denoted by $s\gamma$.

[2] `http://smtlib.cs.uiowa.edu`

To define LCTRSs, we consider different kinds of symbols and terms:

- two signatures Σ_{term} and Σ_{theory} such that $\Sigma = \Sigma_{term} \cup \Sigma_{theory}$,
- a mapping $\mathcal{I}$ that assigns to each basic sort ι occurring in Σ_{theory} a set $\mathcal{V}al_\iota$, i.e., $\mathcal{I}(\iota) = \mathcal{V}al_\iota$,
- a mapping $\mathcal{J}$ that 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
- a set $\Sigma_{val,\iota} \subseteq \Sigma_{theory}$ of *value-constants* for each basic sort ι occurring in Σ_{theory}—$\Sigma_{val,\iota}$ is a set of constant symbols $a : \iota$ such that $\mathcal{J}$ gives a bijective mapping from $\Sigma_{val,\iota}$ to $\mathcal{I}(\iota)$.

Note that for each sort, $\mathcal{I}$ specifies the universe, and for each symbol, $\mathcal{J}$ specifies the interpretation. We define $\mathcal{V}al$ and Σ_{val} as $\bigcup_{\iota \in \mathcal{S}} \mathcal{V}al_\iota$ and $\bigcup_{\iota \in \mathcal{S}} \Sigma_{val,\iota}$, respectively. We require that $\Sigma_{term} \cap \Sigma_{theory} \subseteq \Sigma_{val}$. The basic sorts occurring in Σ_{theory} are called *theory sorts*, and the symbols *theory symbols*. The set of theory sorts is denoted by $\mathcal{S}_{theory}$. Symbols in $\Sigma_{theory} \setminus \Sigma_{val}$ 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 $\Sigma_{theory} \supseteq \Sigma_{core}$, where $\mathcal{S}_{theory}$ includes *bool*, a basic sort of *Booleans*, with $\Sigma_{val,bool} = \{\mathsf{true}, \mathsf{false}\}$ and $\mathcal{I}(bool) = \mathbb{B} = \{\top, \bot\}$, $\Sigma_{core} = \Sigma_{val,bool} \cup \{\wedge, \vee, \Longrightarrow : bool \times bool \Rightarrow bool,\ \neg : bool \Rightarrow bool\} \cup \{=_\iota, \neq_\iota : \iota \times \iota \Rightarrow bool \mid \iota \in \mathcal{S}_{theory}\}$, 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 $=$ and $\neq$ when they are clear from context.

A *constraint* is a theory term $\varphi : bool$. A constraint φ is said to be *valid* if $[\![\varphi\gamma]\!] = \top$ for all substitutions γ with $\gamma|_{\mathcal{V}ar(\varphi)} \subseteq \Sigma_{val}$ and *satisfiable* if $[\![\varphi\gamma]\!] = \top$ for some such substitution. A substitution γ is said to *respect* φ if $\gamma|_{\mathcal{V}ar(\varphi)} \subseteq \Sigma_{val}$ and $[\![\varphi\gamma]\!] = \top$.

Example 2.1 We define basic sorts, signatures, interpretations for bit vectors as follows:

- $\mathcal{S}_{theory}$ be $\{bool\} \cup \{bitvec_n \mid n \geq 1\}$, where $bitvec_n$ is a basic sort for bit vectors with length n,
- $\Sigma_{val} = \Sigma_{val,bool} \cup \bigcup_{n \geq 1} \Sigma_{val,bitvec_n}$, where $\Sigma_{val,bitvec_n} = \{\mathsf{a} \mid a \in \mathbb{BV}_n\}$ for $n \geq 1$,
- Σ_{theory} be $\Sigma_{core} \cup \Sigma_{val} \cup (\bigcup_{n \geq 1} \{+_{n,S}, +_{n,U}, -_{n,S}, -_{n,U}, \times_{n,S}, \times_{n,U}, /_{n,S}, /_{n,U}, \%_{n,S}, \%_{n,U}, \&_n, |_n, \ll_n, \gg_n : bitvec_n \times bitvec_n \Rightarrow bitvec_n\}) \cup \{\circ_{m,n} \mid m, n \geq 1\} \cup \{\sim_n \mid n \geq 1\} \cup \{=_n, >_{n,S}, >_{n,U}, \geq_{n,S}, \geq_{n,U} \mid n \geq 1\}$,
- $\mathcal{V}al_{bitvec_n} = \mathbb{BV}_n$ for $n \geq 1$, and
- the interpretations $\mathcal{I}$ and $\mathcal{J}$ follow the usual semantics.

Note that we use a (in sans-serif font) as the function symbol for $a \in \mathbb{BV}_n$ (in *math* font). We omit sizes from operator symbols, abbreviating, e.g., $+_{32,S}$ to $+_S$.

Let $\Sigma = \Sigma_{term} \cup \Sigma_{theory}$, where $\Sigma_{term} = \{\ \mathsf{pow} : bitvec_4 \times bitvec_4 \Rightarrow bitvec_4\ \}$. Then both $bitvec_4$ and $bool$ are theory sorts. Examples of theory terms are $\#b0000 = \#b0000 + \#b0001$ and $x +_U \#b0011 \geq_U y +_U \#b0101$ which are constraints. Term $\#b0101 +_S \#b1000$ is also a (ground) theory term, but not a constraint. Term $\mathsf{pow}(\#b0010, y)$ is not a theory term.

A *constrained rewrite rule* is a triple $\ell \to r\ [\varphi]$ such that ℓ and r are terms of the same basic sort, φ is a constraint, and ℓ has the form $f(\ell_1, \ldots, \ell_n)$ and contains at least one symbol in $\Sigma_{term} \setminus \Sigma_{theory}$ (i.e., ℓ is not a theory term). If $\varphi = \mathsf{true}$, then 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 $\gamma|_{\mathcal{LV}ar(\ell \to r\ [\varphi])} \subseteq \Sigma_{val}$ 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 *value-constants*. Given a set $\mathcal{R}$ of constrained rewrite rules, we let $\mathcal{R}_{\mathtt{calc}}$ be the set $\{f(x_1,\ldots,x_n) \to y\ [y = f(x_1,\ldots,x_n)] \mid f : \iota_1 \times \cdots \times \iota_n \Rightarrow \iota \in \Sigma_{theory} \setminus \Sigma_{val}\}$. We usually call the elements of $\mathcal{R}_{\mathtt{calc}}$ 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: $s[\ell\gamma]_p \to_{\mathcal{R}} s[r\gamma]_p$ if $\ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{\mathtt{calc}}$ and γ respects $\ell \to r\ [\varphi]$.

Now we define a *logically constrained term rewrite system* (LCTRS, for short) as the abstract reduction system $(T(\Sigma,\mathcal{V}), \to_{\mathcal{R}})$ where $\mathcal{R}$ is a set of constrained rewrite rules. LCTRS $(T(\Sigma,\mathcal{V}), \to_{\mathcal{R}})$ is simply denoted by $\mathcal{R}$. 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.

Example 2.2 Consider the signature in Example 2.1. We reduce $\#b0011 -_U \#b0001$ to #b0010 in one step with the calculation rule $x -_U y \to z\ [z = x -_U y]$, and $\#b0011 \times_U (\#b0010 \times_U (\#b0001 \times_U \#b0001))$ to #b0110 in three steps. To implement an LCTRS calculating the *power* function over $\mathbb{BV}_4$ with the binary-encoding interpretation, we use the signature Σ in Example 2.1 and the following LCTRS $\mathcal{R}_1$:

$$\mathcal{R}_1 = \{\ \mathsf{pow}(x,y) \to \#b0001\ [\#b0000 = y] \quad \mathsf{pow}(x,y) \to x \times_U \mathsf{pow}(x, y -_U \#b0001)\ [y \geq_U \#b0001]\ \}$$

where $\mathsf{pow} : \mathit{bitvec}_4 \times \mathit{bitvec}_4 \Rightarrow \mathit{bitvec}_4$. Using the rules in $\mathcal{R}_1$, $\mathsf{pow}(\#b0010, \#b0011)$ reduces in ten steps to #b1000:

$$\begin{aligned}\mathsf{pow}(\#b0010,\#b0011) &\to_{\mathcal{R}_1} \#b0010 \times_U \mathsf{pow}(\#b0010, \#b0011 -_U \#b0010) \\ &\to_{\mathcal{R}_1} \#b0010 \times_U \mathsf{pow}(\#b0010, \quad \#b0010 \quad) \to_{\mathcal{R}_1} \cdots \to_{\mathcal{R}_1} \#b1000\end{aligned}$$

Hereafter, for readability, bit vectors are written as (signed) decimal numbers; the number of digits of a bit vector with decimal notation can be inferred from its basic sort. For example, 4-digit bit vector #b0011 is simply denoted by 3, and from the sort of pow in Example 2.2, we can know that 2 and 3 in $\mathsf{pow}(2,3)$ stand for #b0010 and #b0011, respectively.

3 Modeling Concurrent Programs by LCTRSs

In this section, we show how to model concurrent programs by LCTRSs. Note that programs in this paper follow the usual syntax and semantics of C, and the size of `int` is assumed to be 32 bits.

As described in Section 1, to represent configurations of sequential programs, previous work introduces a function symbol env for the environment of execution [4]; the function symbol takes both a state of an executed function and values stored in global variables as its arguments. To model concurrent programs by LCTRSs, we also use this function symbol to represent configurations. Since this paper assumes that the number of processes is fixed,we increase arguments of env in order for env to include the states of the processes that are executed simultaneously, along with values stored in global variables.[3]

Let us consider the following program such that `func1` and `func2` assign 1 and 2, respectively, to global variable `x`:

[3] In the case where the number of processes is not fixed (e.g., changed dynamically), we may introduce constructors to represent (multi-)sets (e.g., a binary associative and commutative (AC) symbol | and a constant empty for the empty process) and replace the argument for a state by that for a set of states.

```
1  int x = 0;
2
3  void func1() {
4    x = 1;
5    return;
6  }
```

```
7
8  void func2() {
9    x = 2;
10   return;
11 }
```

We first consider the case where `func1` and `func2` are sequentially executed by a process, i.e., a main function defined as

```
12 int main(){
13   func1();
14   func2();
15   return 0;
16 }
```

In previous work, the environment of the sequential execution is represented as $\mathsf{env}(v_{\mathtt{x}}, st)$ where env has sort $\mathit{bitvec}_{32} \times \mathit{result} \to \mathit{env}$. The first argument $v_{\mathtt{x}}$ of env is a value stored in global variable x, and the second argument st of env is a term representing a state of an executed function. We transform the program with sequential execution of `func1` and `func2` into the following rewrite rules:

$$\mathcal{R}_2 = \left\{ \begin{array}{l} \mathsf{env}(x, \mathsf{main}_{14}(\ \mathsf{func1}\ )) \to \mathsf{env}(\ 1, \mathsf{main}_{14}(\ \mathsf{func1}_5\ )) \\ \mathsf{env}(x, \mathsf{main}_{14}(\ \mathsf{func1}_5\ )) \to \mathsf{env}(x, \mathsf{main}_{14}(\ \mathsf{return}\ )) \\ \mathsf{env}(x, \mathsf{main}_{15}(\ \mathsf{func2}\ )) \to \mathsf{env}(\ 2, \mathsf{main}_{15}(\mathsf{func2}_{10})) \\ \mathsf{env}(x, \mathsf{main}_{15}(\mathsf{func2}_{10})) \to \mathsf{env}(x, \mathsf{main}_{15}(\ \mathsf{return}\ )) \\ \mathsf{env}(x, \quad \mathsf{main} \qquad\ ) \to \mathsf{env}(x, \mathsf{main}_{14}(\ \mathsf{func1}\ )) \\ \mathsf{env}(x, \mathsf{main}_{14}(\ \mathsf{return}\ )) \to \mathsf{env}(x, \mathsf{main}_{15}(\ \mathsf{func2}\ )) \\ \mathsf{env}(x, \mathsf{main}_{15}(\ \mathsf{return}\ )) \to \mathsf{env}(x, \quad \mathsf{return} \qquad\ ) \end{array} \right\}$$

where main, func1, $\mathsf{func1}_5$, return, func2, and $\mathsf{func2}_{10}$ are constants with basic sort *result*, and main_{14} and main_{15} are function symbols with sort $\mathit{result} \Rightarrow \mathit{result}$. Note that $\mathsf{func1}_i$ ($\mathsf{func2}_j$) correspond to the state that the program counter lies between lines $i-1$ and i (lines $j-1$ and j). The initial configuration is represented as $\mathsf{env}(0, \mathsf{func1})$, and the sequential execution of `func1` and `func2` corresponds to the rewrite sequence starting from $\mathsf{env}(0, \mathsf{func1})$.

We now illustrate how to adapt the idea above to concurrent programs by means of the program above. Note that this approach can be adapted to parallel computation. We assume that there are two processes: the first one executes `func1` and the other executes `func2` concurrently. For the sake of simplicity, we consider such an informal concurrent semantics because the concrete one is not so relevant to the adaptation of the approach in previous work to concurrent programs. In addition, for any process, we do not consider its main function. The environment is represented as follows:

$$\mathsf{env}(v_{\mathtt{x}}, st_1, st_2)$$

where env has sort $\mathit{bitvec}_{32} \times \mathit{result} \times \mathit{result} \to \mathit{env}$. The first argument $v_{\mathtt{x}}$ of env is a value stored in global variable x as in the sequential case. The second and third arguments st_1, st_2 of env are terms representing states of functions that the first and second process execute, respectively. We transform the program with concurrent execution into the following rewrite rules:

$$\mathcal{R}_3 = \left\{ \begin{array}{ll} \mathsf{env}(x,\ \mathsf{func1},\ st_2) \to \mathsf{env}(1, \mathsf{func1}_5, st_2) & \mathsf{env}(x, st_1,\ \ \mathsf{func2}\ ) \to \mathsf{env}(2, st_1, \mathsf{func2}_{10}) \\ \mathsf{env}(x, \mathsf{func1}_5, st_2) \to \mathsf{env}(x,\ \mathsf{return},\ st_2) & \mathsf{env}(x, st_1, \mathsf{func2}_{10}) \to \mathsf{env}(x, st_1,\ \ \mathsf{return}\ ) \end{array} \right\}$$

The initial configuration is represented as $\mathsf{env}(0,\mathsf{func1},\mathsf{func2})$, and concurrent executions of `func1` and `func2` correspond to rewrite sequences starting from $\mathsf{env}(0,\mathsf{func1},\mathsf{func2})$. Unlike the sequential case, we have four normalizing reductions starting from $\mathsf{env}(0,\mathsf{func1},\mathsf{func2})$, e.g.,

$$\begin{aligned}\mathsf{env}(0,\mathsf{func1},\mathsf{func2}) &\to_{\mathcal{R}_3} \mathsf{env}(1,\mathsf{func1}_5,\ \mathsf{func2}\ ) \to_{\mathcal{R}_3} \mathsf{env}(1,\mathsf{return},\mathsf{func2}\,)\\ &\to_{\mathcal{R}_3} \mathsf{env}(2,\ \mathsf{return},\ \mathsf{func2}_{10}) \to_{\mathcal{R}_3} \mathsf{env}(2,\mathsf{return},\mathsf{return})\\ \mathsf{env}(0,\mathsf{func1},\mathsf{func2}) &\to_{\mathcal{R}_3} \mathsf{env}(2,\ \mathsf{func1},\ \mathsf{func2}_{10}) \to_{\mathcal{R}_3} \mathsf{env}(2,\ \mathsf{func1},\ \mathsf{return})\\ &\to_{\mathcal{R}_3} \mathsf{env}(1,\mathsf{func1}_5,\ \mathsf{return}\ ) \to_{\mathcal{R}_3} \mathsf{env}(1,\mathsf{return},\mathsf{return})\end{aligned}$$

The initial term $\mathsf{env}(0,\mathsf{func1},\mathsf{func2})$ has two different normal forms, and $\mathcal{R}_3$ is not confluent, while $\mathcal{R}_3$ is trivially terminating.

To focus on the main idea, we used simple functions that do not call any function, and thus, we do not introduce any stack for function calls as in [4]. When considering function calls, we combine the approaches in this section and [4] by replacing the second and third arguments of env by a stack.

4 Modeling Operations of Semaphores by LCTRSs

In this section, we illustrate how to model by LCTRSs concurrent programs with exclusive control operated by means of semaphores. A semaphore is a variable to specify the availability of a corresponding shared resource. Note that the range of semaphores are non-negative integers from 0 to some u—$u = 1$ for binary semaphores, and $u > 1$ for counting ones. This paper deals with the operations `down` and `up` of a semaphore `s` for a shared resource *src*, which are defined as follows [11]:

- Operation `down` is executed as `down(&s)` before accessing *src*: If the value of `s` is not 0, then it is decreased by 1; otherwise, the process executing `down` goes into the wait state, queuing up for `s`.
- Operation `up` is executed as `up(&s)` in releasing the access to *src*: If at least one process waits for `s`, then the value of `s` is kept as it is and the process that is at the top of the waiting queue goes into the executable state;[4] otherwise, the value of `s` is increased by 1.
- `down` and `up` are *atomic operations*—when an atomic operation a is executed, any other operations (processes) cannot interrupt until the execution of a finishes.

Note that the above specification can perform for both binary and counting semaphores by switching binary and counting semaphores by means of the initial value of `s`.

In the case where there are exactly two processes, the waiting queue of a semaphore contains at most one process. Therefore, we do not have to take into account any queue for semaphores—if a process executes operation `up` and the queue is not empty, then the waiting process is the other one only. However, when there are three or more processes, we need to handle a queue.

To represent the waiting queue of a semaphore, we adopt a so-called *turn waiting system* with number tickets, which is usually seen at banks, post office, etc. We consider a turn waiting system with the following usual mechanism:

- Numbers written in tickets are positive integers—0 is used to indicate that a process is not waiting.
- There is a display board to call a number written in a ticket, the owner of which can get service.
- There is a ticket machine for turn waiting—when the machine issues a ticket, it prepares the next number ticket for an upcoming process.

[4] The process going into the executable state acquires the semaphore `s` and starts to access the shared resource.

- If a process has to wait for the semaphore, it gets a number ticket from the ticket machine.
- A process having a number ticket waits until the number in the ticket is called by the display board.

Let us consider a program with a shared `int` resource *src*, a semaphore `s` for *src*, and m processes. Then, the environment can be represented as follows:

$$\mathsf{env}(\mathsf{sem}(v_{\mathtt{s}}, v_d, v_t), v_{src}, \mathsf{p}(st_1, n_1), \ldots, \mathsf{p}(st_m, n_m))$$

where

- env has sort $\mathit{semaphore} \times \mathit{bitvec}_{32} \times \overbrace{\mathit{process} \times \cdots \times \mathit{process}}^{m} \to \mathit{env}$,
- p has sort $\mathit{result} \times \mathit{bitvec}_k \to \mathit{process}$,
- sem has sort $\mathit{bitvec}_{32} \times \mathit{bitvec}_k \times \mathit{bitvec}_k \Rightarrow \mathit{semaphore}$, and
- $k = \lfloor \log(m-1) \rfloor + 2$.

The first argument ($\mathsf{sem}(v_{\mathtt{s}}, v_d, v_t)$ above) of env is a term for semaphore `s`. Term $\mathsf{sem}(v_{\mathtt{s}}, v_d, v_t)$ is rooted by a function symbol sem for semaphores, which takes three arguments: The first argument ($v_{\mathtt{s}}$ above) stores a value of `s`, taking a non-negative integer; the second and third arguments perform as a display board and a ticket machine, respectively, for `s`; stored values (v_d, v_t above) are currently-called and next-issued numbers, respectively. The sort of the first argument of sem is bitvec_{32} because the sort of `s` in the program is `int`.[5]

The second argument (v_{src} above) of env stores a value of *src*. The state of the i-th process is stored in the "$i+2$"-th argument ($p(st_i, n_i)$ above) of env: st_i is the state of an executed function of the i-th process, and n_i is a number ticket that the i-th process owns; n_i is 0 if and only if the process is not waiting for `s`.

In the case where there are several semaphores, for each semaphore, we prepare a display board and a ticket machine, increasing arguments of env and p in order for env and p to have turn ticket systems and number tickets, respectively, for all semaphores.

Since the number of processes is assumed to be fixed (e.g., m above), the number of waiting processes for each semaphore is not over the total number of processes, i.e., at most $m-1$. For this reason, we prepare minimum necessary tickets: We reuse them if the next number prepared by the ticket machine is over the number of processes.[6] If number tickets are scaled in k bits (i.e., $2^k - 1$ tickets are prepared[7]) and the counter for the ticket machine is increased by 1 in issuing tickets, then the number following $2^k - 1$ is 0. However, since 0 is used to indicate that a process does not have a ticket (i.e., the process does not wait for the semaphore), we cannot use 0 for any number ticket. Therefore, we use only odd numbers for tickets, initializing a display board and a ticket machine with odd numbers 1 and 3, respectively, and increasing values of the display board and the ticket machine by 2. Viewed in this light, we define k as $\lfloor \log(m-1) \rfloor + 2$.

The initial configuration is represented as $\mathsf{env}(\mathsf{sem}(v'_{\mathtt{s}}, 1, 3), v'_{src}, \mathsf{p}(st'_1, 0), \ldots, \mathsf{p}(st'_m, 0))$ where $v'_{\mathtt{s}}$ and v'_{src} are the initial values of `s` and *src*, respectively, and st'_i is the initial state of the i-th process. Note that if `s` is a binary semaphore, then $v'_{\mathtt{s}} = 1$, and otherwise, $v_{\mathtt{s}}$ is a positive integer greater than 1, which follows the specification of `s`.

In the following, for the page limitation, we restrict `s` to a binary semaphore. The approach below can be adapted to counting semaphores by modifying some rules.

[5] When `s` is a binary semaphore, we may give sort bitvec_1 to the first argument of sem.

[6] The semantics of $+_U$ can reset counters because $+_{n,U}$ is defined as follows: for $a, b, c \in \mathbb{BV}n$, $a +_{n,U} b = c$ iff $\langle c \rangle_U = \langle a \rangle_U + \langle b \rangle_U \mod 2^n$, where given a bit vector d, $\langle d \rangle_U$ is the corresponding decimal number of d w.r.t. binary encoding.

[7] Recall that 0 is not used for tickets.

Taking the above turn waiting system for semaphores into account, we represent operations `down` and `up` by the following rewrite rules:

$$\text{(R1)}\ \ \mathsf{env}(\mathsf{sem}(s,d,t),src,\ldots,\mathsf{p}(st_1,0),\ldots) \to \mathsf{env}(\mathsf{sem}(s-_U 1,\ d,\ t\ ),src,\ldots,\mathsf{p}(st_2,0),\ldots)\ \ [s\neq 0]$$
$$\text{(R2)}\ \ \mathsf{env}(\mathsf{sem}(s,d,t),src,\ldots,\mathsf{p}(st_1,n),\ldots) \to \mathsf{env}(\mathsf{sem}(\ s,\ d,\ t+_U 2),src,\ldots,\mathsf{p}(st_1,t),\ldots)\ \ [s=0\wedge n=0]$$
$$\text{(R3)}\ \ \mathsf{env}(\mathsf{sem}(s,d,t),src,\ldots,\mathsf{p}(st_1,n),\ldots) \to \mathsf{env}(\mathsf{sem}(\ s,\ d,\ t\ ),src,\ldots,\mathsf{p}(st_2,0),\ldots)\ \ [n=d\wedge n\neq 0]$$

$$\text{(R4)}\ \ \mathsf{env}(\mathsf{sem}(s,d,t),src,\ldots,\mathsf{p}(st_3,0),\ldots) \to \mathsf{env}(\mathsf{sem}(\ s,\ d+_U 2,\ t\ ),src,\ldots,\mathsf{p}(st_4,0),\ldots)\ \ [t\neq d+_U 2]$$
$$\text{(R5)}\ \ \mathsf{env}(\mathsf{sem}(s,d,t),src,\ldots,\mathsf{p}(st_3,0),\ldots) \to \mathsf{env}(\mathsf{sem}(s+_U 1,\ d,\ t\ ),src,\ldots,\mathsf{p}(st_4,0),\ldots)\ \ [t=d+_U 2]$$

where src,d,t, are variables for a resource, a display board, a ticket machine, and a number ticket, respectively, but st_1,st_2,st_3,st_4 are not: st_1,st_3 are terms representing the state of the executed function, which calls `down` and `up`, respectively, and $st3,st_4$ are terms representing the next state of st_1,st_3, respectively. In a concrete example, $st_1,\ldots,st_4$ are terms rooted by function symbols that represent intermediate states of execution (see Example 4.1). Notation "$\ldots,\mathsf{p}(st,j),\ldots$" stands for "$p_1,\ldots,p_{i-1},\mathsf{p}(st,j),p_{i+1},\ldots,p_m$", where $p_1,\ldots,p_m$ are distinct variables with basic sort *process*. That is to say, each of rules (R1)–(R5) may consist of (at most) m rules.

For states st_1,st_3 of the i-th process operating `down` or `up`, respectively, rules (R1) and (R2) represent operations of `down`, and (R3)–(R5) represent operations of `up`. The operation that each rule represents is as follows:

(R1) If the process requires semaphore `s` and the value of `s` is not 0 (i.e., the first argument of env is not 1), then it is decreased by 1 (i.e., the process acquires the semaphore) and the state of the function executed in the process moves to the next state st_2.

(R2) If the process requires semaphore `s` but the value of `s` is 0 (i.e., the first argument of env is 0), then the process acquires a number ticket t, the ticket machine—the third argument of sem—prepares the next number ticket "$t+_U 2$" for an upcoming process, and thus the process goes into the wait state represented as $\mathsf{p}(st_1,t)$ with $t\neq 0$.

(R3) If the number n that the process has matches the calling number d on the display board, then the process goes into the executable state, i.e., the state st_1 moves to the next state st_3 representing a state that acquires semaphore `s`.

(R4) If there is a process waiting for semaphore s (i.e., the constraint $d=t+_U 2$ does not hold), then the value of `s` is kept as it is and the next waiting process is called (i.e., the value of d is increased by 2) and the state st_3 moves to the next state st_4.

(R5) If there is no process waiting for semaphore `s` (i.e., the constraint $d=t+_U 2$ holds), then the value of `s` is increased by 1 and the state st_3 moves to the next state st_4.

For (R1), (R4), and (R5), we do not have to restrict the second argument of p to 0 such as $\mathsf{p}(st_i,0)$ ($i\in\{1,3\}$) and $\mathsf{p}(st_{i+1},0)$, i.e., $\mathsf{p}(st_i,n)$ and $\mathsf{p}(st_{i+1},n)$ with n a variable is enough. However, when applying (R1), (R4), and (R5) to terms that are reached from the term for the initial configuration, the value of the second argument of p is always 0. To exclude any improbable case, we restrict n to 0. Since `down` and `up` are atomic operations, each of their actions is not represented by several rules that are sequentially applied, but by a single rule.

If constraint $d\neq t+_U 2$ of (R4) holds, then there is a process waiting for semaphore `s`; otherwise (i.e., $d=t+_U 2$ of (R5) holds), no process is waiting for acquiring `s`. This is because the ticket t is not issued yet and the number ticket lastly issued is being called by the display board d. For example, if the display board shows 11 and the ticket for an upcoming process is 13, then all waiting processes are

called, acquiring the semaphore. By checking whether $t = d +_U 2$ holds, we can decide whether there is a waiting process or not. In this approach, it is not necessary to refer to the states any other processes, and thus a rewrite rule represents an action of a single process. Moreover, since each process has the value corresponding to a non-duplicated number ticket, we can represent the order of the waiting queue without using lists.

One may think that an action of `up` is not represented by a single rule, because the action with a waiting process corresponds to (R3) and (R4). Though, after the application of (R4), the waiting process gets ready for being the executable state. Thus, rule (R4) performs the action of `up`.

Example 4.1 Consider the following reader/writer program with a binary semaphore:

```
1  int s = 1; /* semaphore for x */
2  int x = 0;
3
4  void rdr(void) {
5    int y = 0;
6    down(&s);
7    y = x;
8    up(&s);
9    return;
10 }
11
12 void wrtr(void) {
13   down(&s);
14   x = 1;
15   up(&s);
16   return;
17 }
```

For the sake of simplicity, we assume that there are three processes, the first and second ones execute `reader` and the third executes `writer`. Following both a simplified version of the transformation in [4] and the idea illustrated in this section, the program is transformed into an LCTRS $\mathcal{R}_{smphr}$ shown in Figure 1, where $\mathsf{env} : \mathit{semaphore} \times \mathit{bitvec}_{32} \times \mathit{process} \times \mathit{process} \times \mathit{process} \Rightarrow \mathit{env}$, $\mathsf{sem} : \mathit{bitvec}_{32} \times \mathit{bitvec}_3 \times \mathit{bitvec}_3 \Rightarrow \mathit{semaphore}$, $\mathsf{p} : \mathit{result} \times \mathit{bitvec}_3 \Rightarrow \mathit{process}$, $\mathsf{rdr}, \mathsf{return}, \mathsf{wrtr}, \mathsf{wrtr}_{13}, \mathsf{wrtr}_{15} : \mathit{process}$, and $\mathsf{rdr}_6, \mathsf{rdr}_7, \mathsf{rdr}_8 : \mathit{bitvec}_{32} \Rightarrow \mathit{process}$. The initial configuration is represented as $\mathsf{env}(\mathsf{sem}(1,1,3), 0, \mathsf{p}(\mathsf{rdr}, 0), \mathsf{p}(\mathsf{rdr}, 0), \mathsf{p}(\mathsf{wrtr}, 0))$. The reduction illustrated in Figure 2 corresponds to an execution that the first process acquires `s` and the third and second wait in order.

5 Conclusion

In this paper, assuming that the number of processes is fixed, we showed how to represent exclusive control, which is not handled in sequential programs, by LCTRSs, and transformed a concurrent programs with a binary semaphore into an LCTRS. Our approach for semaphores does not depend on the setting of LCTRSs, and thus, the approach would work for other computation models that have built-in bit vectors or integers. For the page limitation, we did not show the case of counting semaphores. We have to show how to adapt the approach to binary semaphores to counting ones.

We have to evaluate our approach by means of experiments on e.g., equivalence verification. Since our approach does not depend on the setting of LCTRSs, we will show the usefulness of the approach with LCTRSs by means of such an empirical evaluation.

The approach in this paper can be adapted to *mutex*. We will formalize the adaptations to mutex as future work. Another future work is to prove correctness of the transformation. We are also interested in the adaptation to the interrupt operation for waiting queues. Furthermore, we would like to evaluate the usefulness of the transformed LCTRSs by e.g., verifying that the exclusive control in programs is implemented correctly.

$$
\left\{\begin{array}{l}
\mathsf{env}(\mathsf{sem}(s,d,t),x,\mathsf{p}(\ \mathsf{rdr},\ 0),p_2,p_3) \to \mathsf{env}(\mathsf{sem}(\ s,\ d,\ t\ ),x,\mathsf{p}(\mathsf{rdr}_6(0),0),p_2,p_3) \\
\mathsf{env}(\mathsf{sem}(s,d,t),x,\mathsf{p}(\mathsf{rdr}_6(y),0),p_2,p_3) \to \mathsf{env}(\mathsf{sem}(s-_U 1,\ d,\ t\ ),x,\mathsf{p}(\mathsf{rdr}_7(y),0),p_2,p_3)\ [s \neq 0] \\
\mathsf{env}(\mathsf{sem}(s,d,t),x,\mathsf{p}(\mathsf{rdr}_6(y),n),p_2,p_3) \to \mathsf{env}(\mathsf{sem}(\ s,\ d,\ t+_U 2),x,\mathsf{p}(\mathsf{rdr}_6(y),t),p_2,p_3)\ [s = 0 \wedge n = 0] \\
\mathsf{env}(\mathsf{sem}(s,d,t),x,\mathsf{p}(\mathsf{rdr}_6(y),n),p_2,p_3) \to \mathsf{env}(\mathsf{sem}(\ s,\ d,\ t\ ),x,\mathsf{p}(\mathsf{rdr}_7(y),0),p_2,p_3)\ [n = d \wedge n \neq 0] \\
\mathsf{env}(\mathsf{sem}(s,d,t),x,\mathsf{p}(\mathsf{rdr}_7(y),0),p_2,p_3) \to \mathsf{env}(\mathsf{sem}(\ s,\ d,\ t\ ),x,\mathsf{p}(\mathsf{rdr}_8(x),0),p_2,p_3) \\
\mathsf{env}(\mathsf{sem}(s,d,t),x,\mathsf{p}(\mathsf{rdr}_8(y),0),p_2,p_3) \to \mathsf{env}(\mathsf{sem}(\ s,\ d+_U 2,\ t\ ),x,\mathsf{p}(\ \mathsf{return},0),p_2,p_3)\ [t \neq d+_U 2] \\
\mathsf{env}(\mathsf{sem}(s,d,t),x,\mathsf{p}(\mathsf{rdr}_8(y),0),p_2,p_3) \to \mathsf{env}(\mathsf{sem}(s+_U 1,\ d,\ t\ ),x,\mathsf{p}(\ \mathsf{return},0),p_2,p_3)\ [t = d+_U 2]
\end{array}\right\}
$$

$$
\cup \left\{\begin{array}{l}
\mathsf{env}(\mathsf{sem}(s,d,t),x,p_1,\mathsf{p}(\ \mathsf{rdr},\ 0),p_2) \to \mathsf{env}(\mathsf{sem}(\ s,\ d,t),x,p_1,\mathsf{p}(\mathsf{rdr}_6(0),0),p_2) \\
\qquad\qquad \cdots \\
\mathsf{env}(\mathsf{sem}(s,d,t),x,p_1,\mathsf{p}(\mathsf{rdr}_8(y),0),p_2) \to \mathsf{env}(\mathsf{sem}(s+_U 1,d,t),x,p_1,\mathsf{p}(\ \mathsf{return},0),p_2)\ [t = d+_U 2]
\end{array}\right\}
$$

$$
\cup \left\{\begin{array}{l}
\mathsf{env}(\mathsf{sem}(s,d,t),x,p_1,p_2,\mathsf{p}(\ \mathsf{wrtr},\ 0)) \to \mathsf{env}(\mathsf{sem}(s-_U 1,\ d,\ t\ ),x,p_1,p_2,\mathsf{p}(\mathsf{wrtr}_{13},0))\ [s \neq 0] \\
\mathsf{env}(\mathsf{sem}(s,d,t),x,p_1,p_2,\mathsf{p}(\ \mathsf{wrtr},\ n)) \to \mathsf{env}(\mathsf{sem}(\ s,\ d,\ t+_U 2),x,p_1,p_2,\mathsf{p}(\ \mathsf{wrtr},\ t))\ [s = 0 \wedge n = 0] \\
\mathsf{env}(\mathsf{sem}(s,d,t),x,p_1,p_2,\mathsf{p}(\ \mathsf{wrtr},\ n)) \to \mathsf{env}(\mathsf{sem}(\ s,\ d,\ t\ ),x,p_1,p_2,\mathsf{p}(\mathsf{wrtr}_{13},0))\ [n = d \wedge n \neq 0] \\
\mathsf{env}(\mathsf{sem}(s,d,t),x,p_1,p_2,\mathsf{p}(\mathsf{wrtr}_{13},0)) \to \mathsf{env}(\mathsf{sem}(\ s,\ d,\ t\ ),1,p_1,p_2,\mathsf{p}(\mathsf{wrtr}_{15},0)) \\
\mathsf{env}(\mathsf{sem}(s,d,t),x,p_1,p_2,\mathsf{p}(\mathsf{wrtr}_{15},0)) \to \mathsf{env}(\mathsf{sem}(\ s,\ d+_U 2,\ t\ ),x,p_1,p_2,\mathsf{p}(\mathsf{return},0))\ [t \neq d+_U 2] \\
\mathsf{env}(\mathsf{sem}(s,d,t),x,p_1,p_2,\mathsf{p}(\mathsf{wrtr}_{15},0)) \to \mathsf{env}(\mathsf{sem}(s+_U 1,\ d,\ t\ ),x,p_1,p_2,\mathsf{p}(\mathsf{return},0))\ [t = d+_U 2]
\end{array}\right\}
$$

Figure 1: an LCTRS $\mathcal{R}_4$ for the reader/writer program

$$
\begin{array}{rl}
 & \mathsf{env}(\mathsf{sem}(\ 1,\ 1,\ 3\ ),0,\mathsf{p}(\ \mathsf{rdr},\ 0),\mathsf{p}(\ \mathsf{rdr},\ 0),\mathsf{p}(\ \mathsf{wrtr},\ 0)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 1,\ 1,\ 3\ ),0,\mathsf{p}(\mathsf{rdr}_6(0),0),\mathsf{p}(\ \mathsf{rdr},\ 0),\mathsf{p}(\ \mathsf{wrtr},\ 0)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 1,\ 1,\ 3\ ),0,\mathsf{p}(\mathsf{rdr}_6(0),0),\mathsf{p}(\mathsf{rdr}_6(0),0),\mathsf{p}(\ \mathsf{wrtr},\ 0)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(1-_U 1,\ 1,\ 3\ ),0,\mathsf{p}(\mathsf{rdr}_7(0),0),\mathsf{p}(\mathsf{rdr}_6(0),0),\mathsf{p}(\ \mathsf{wrtr},\ 0)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 0,\ 1,\ 3\ ),0,\mathsf{p}(\mathsf{rdr}_7(0),0),\mathsf{p}(\mathsf{rdr}_6(0),0),\mathsf{p}(\ \mathsf{wrtr},\ 0)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 0,\ 1,\ 3+_U 2),0,\mathsf{p}(\mathsf{rdr}_7(0),0),\mathsf{p}(\mathsf{rdr}_6(0),0),\mathsf{p}(\ \mathsf{wrtr},\ 3)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 0,\ 1,\ 5\ ),0,\mathsf{p}(\mathsf{rdr}_7(0),0),\mathsf{p}(\mathsf{rdr}_6(0),0),\mathsf{p}(\ \mathsf{wrtr},\ 3)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 0,\ 1,\ 5\ ),0,\mathsf{p}(\mathsf{rdr}_8(0),0),\mathsf{p}(\mathsf{rdr}_6(0),0),\mathsf{p}(\ \mathsf{wrtr},\ 3)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 0,\ 1,\ 5+_U 2),0,\mathsf{p}(\mathsf{rdr}_8(0),0),\mathsf{p}(\mathsf{rdr}_6(0),5),\mathsf{p}(\ \mathsf{wrtr},\ 3)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 0,\ 1,\ 7\ ),0,\mathsf{p}(\mathsf{rdr}_8(0),0),\mathsf{p}(\mathsf{rdr}_6(0),5),\mathsf{p}(\ \mathsf{wrtr},\ 3)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 0,\ 1+_U 2,\ 7\ ),0,\mathsf{p}(\ \mathsf{return},\ 0),\mathsf{p}(\mathsf{rdr}_6(0),5),\mathsf{p}(\ \mathsf{wrtr},\ 3)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 0,\ 3,\ 7\ ),0,\mathsf{p}(\ \mathsf{return},\ 0),\mathsf{p}(\mathsf{rdr}_6(0),5),\mathsf{p}(\ \mathsf{wrtr},\ 3)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 0,\ 3,\ 7\ ),0,\mathsf{p}(\ \mathsf{return},\ 0),\mathsf{p}(\mathsf{rdr}_6(0),5),\mathsf{p}(\mathsf{wrtr}_{13},0)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 0,\ 3,\ 7\ ),1,\mathsf{p}(\ \mathsf{return},\ 0),\mathsf{p}(\mathsf{rdr}_6(0),5),\mathsf{p}(\mathsf{wrtr}_{15},0)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 0,\ 3+_U 2,\ 7\ ),1,\mathsf{p}(\ \mathsf{return},\ 0),\mathsf{p}(\mathsf{rdr}_6(0),5),\mathsf{p}(\mathsf{return},0)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 0,\ 5,\ 7\ ),1,\mathsf{p}(\ \mathsf{return},\ 0),\mathsf{p}(\mathsf{rdr}_6(0),5),\mathsf{p}(\mathsf{return},0)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 0,\ 5,\ 7\ ),1,\mathsf{p}(\ \mathsf{return},\ 0),\mathsf{p}(\mathsf{rdr}_7(0),0),\mathsf{p}(\mathsf{return},0)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 0,\ 5,\ 7\ ),1,\mathsf{p}(\ \mathsf{return},\ 0),\mathsf{p}(\mathsf{rdr}_8(1),0),\mathsf{p}(\mathsf{return},0)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(0+_U 1,\ 5,\ 7\ ),1,\mathsf{p}(\ \mathsf{return},\ 0),\mathsf{p}(\ \mathsf{return},\ 0),\mathsf{p}(\mathsf{return},0)) \\
\to_{\mathcal{R}_4} & \mathsf{env}(\mathsf{sem}(\ 1,\ 5,\ 7\ ),1,\mathsf{p}(\ \mathsf{return},\ 0),\mathsf{p}(\ \mathsf{return},\ 0),\mathsf{p}(\mathsf{return},0))
\end{array}
$$

Figure 2: a reduction of $\mathcal{R}_4$

References

[1] Franz Baader & Tobias Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press, doi:10.1017/CBO9781139172752.

[2] 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.

[3] 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.

[4] 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, doi:10.4204/EPTCS.289.3.

[5] 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.

[6] 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.

[7] Daniel Kroening & Ofer Strichman (2016): *Decision Procedures: An Algorithmic Point of View*, 2 edition. Springer, doi:10.1007/978-3-662-50497-0.

[8] 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.

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

[10] Uday S. Reddy (1990): *Term Rewriting Induction*. In: *Proc. CADE 1990*, *LNCS* 449, Springer, pp. 162–177, doi:10.1007/3-540-52885-7_86.

[11] Andrew S. Tanenbaum & Albert S. Woodhull (2006): *Operating systems — design and implementation*, 3 edition. Pearson Education.

[12] 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.