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

In this paper, we show a new approach to transformations of an imperative program with function calls and global variables into a logically constrained term rewriting system. The resulting system represents transitions of the whole execution environment with a call stack. More precisely, we prepare a function symbol for the whole environment, which stores values for global variables and a call stack as its arguments. For a function call, we prepare rewrite rules to push the frame to the stack and to pop it after the execution. Any running frame is located at the top of the stack, and statements accessing global variables are represented by rewrite rules for the environment symbol. We show a precise transformation based on the approach and prove its correctness.

1 Introduction

Recently, analyses of imperative programs (written in C, Java Bytecode, etc.) via transformations into term rewriting systems have been investigated [2, 3, 6, 11]. 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, 6, 9, 13]. To capture the existing approaches for constrained rewriting in one setting, the framework of a *logically constrained term rewriting system* (an LCTRS, for short) has been proposed [7]. Transformations of C programs with integers, characters, arrays of integers, global variables, and so on into LCTRSs have been discussed in [5].

A basic idea of transforming functions defined in simple imperative programs over the integers, so-called *while* programs, 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 function `sum1` in Figure 1, which is written in the 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 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 transition for the execution of the function `sum1` can be modeled by an LCTRS as follows [6, 5]:

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

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

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

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

Note that the auxiliary function symbol u_1 can be considered locations stored in the *program counter*. The transformed LCTRS is useful to verify the original program [5]. For example, the theorem proving method based on *rewriting induction* [12] can automatically prove that $\forall n \in \mathbb{Z}.\ \mathsf{sum1}(n) = \frac{n(n+1)}{2}$, i.e., correctness of the C program [13, 8, 5].

A function call is added as an extra argument of the auxiliary symbol that corresponds to the statement of the call. Let us consider the following function in addition to `sum1` in Figure 1:

```
int g(int x){
  int z = 0;
  z = sum1(x);
  return x * z;
}
```

This function is transformed into the following rules:

$$\{\ \mathsf{g}(x) \to \mathsf{u}_2(x,0),\ \ \mathsf{u}_2(x,z) \to \mathsf{u}_3(x,z,\mathsf{sum1}(x)),\ \ \mathsf{u}_3(x,z,\mathsf{return}(y)) \to \mathsf{u}_4(x,y),\ \ \mathsf{u}_4(x,z) \to \mathsf{return}(x \times z)\ \}$$

The auxiliary function symbol u_2 calls $\mathsf{sum1}$ in the third argument of u_3 by means of the rule for u_2.

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. This program is transformed into the following LCTRS [5]:

$$\mathcal{R}_2 = \left\{ \begin{array}{rll} \mathsf{sum1}(x,num) & \to \mathsf{u}_1(x,0,0,num+1), & \\ \mathsf{u}_1(x,i,z,num) & \to \mathsf{u}_1(x,i+1,z+i+1,num) & [\quad i < x \quad], \\ \mathsf{u}_1(x,i,z,num) & \to \mathsf{return}(z,num) & [\,\neg(i<x)\,], \\ \mathsf{g}(x,num) & \to \mathsf{u}_2(x,num,0), & \\ \mathsf{u}_2(x,num,z) & \to \mathsf{u}_2'(x,num+1,z), & \\ \mathsf{u}_2'(x,num,z) & \to \mathsf{u}_3(x,num,z,\mathsf{sum1}(x,num)), & \\ \mathsf{u}_3(x,num_{old},z,\mathsf{return}(y,num_{new})) & \to \mathsf{u}_4(x,num_{new},y), & \\ \mathsf{u}_4(x,num,z) & \to \mathsf{return}(x \times z,num) & \end{array} \right\}$$

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 does not have to be passed to a particular function or a process because another function or process may access the global variable.

```
int num = 0;

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

```
int g(int x){
  int z = 0;
  num = num + 1;
  z = sum1(x);
  return x * z;
}
```

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

In this paper, we show another approach to transformations of imperative programs with function calls and global variables into LCTRSs. Our target languages are call-by-value imperative languages such as C. For this reason, we use a small subclass of C programs over the integers as fundamental imperative programs. We show a precise transformation along the approach and prove its correctness.

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 not only user-defined functions but also 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 located 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 as well as the previous transformation, and transform statements accessing global variables into rewrite rules for the introduced symbol for the environment.

This paper is organized as follows. In Section 2, we recall LCTRSs and a small imperative language. In Section 3, using an example, we show a new approach to transformations of imperative programs into LCTRSs. In Section 4, we precisely define a transformation and show its correctness. In Section 5, we describe a future direction of this research.

2 Preliminaries

In this section, we recall LCTRSs, following the definitions in [7, 5]. We also recall a small imperative language SIMP^+ with global variables and function calls. Familiarity with basic notions on term rewriting [1, 10] is assumed.

2.1 Logically Constrained Term Rewriting Systems

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 $s_1, \ldots, s_n$ is denoted by $\mathcal{V}ar(s_1, \ldots, s_n)$. 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 *context* $C[\ ]$ is a term containing one *hole* $\Box_\iota : \iota$. For a term $s : \iota$, $C[s]$ denotes the term obtained from $C[]$ by replacing $\Box_\iota$ by s.

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 *theory term*. For ground theory terms, we define the interpretation as $[\![f(s_1, \ldots, s_n)]\!] = \mathcal{J}(f)([\![s_1]\!], \ldots, [\![s_n]\!])$. For every ground theory 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 theory 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 $\mathsf{true}, \mathsf{false} : bool$, $\wedge, \vee, \implies : bool \times bool \Rightarrow bool$, $\neg : bool \Rightarrow bool$, and, for all theory sorts ι, symbols $=_\iota, \neq_\iota : \iota \times \iota \Rightarrow bool$, and an evaluation function $\mathcal{J}$ that interprets these symbols as expected. We omit the sort subscripts from $=$ and $\neq$ when they are clear from context.

The standard integer signature Σ^{int}_{theory} is $\Sigma^{core}_{theory} \cup \{+, -, *, \mathsf{exp}, \mathsf{div}, \mathsf{mod} : int \times int \Rightarrow int\} \cup \{\geq, > : int \times int \Rightarrow bool\} \cup \{\mathsf{n} : int \mid n \in \mathbb{Z}\}$ 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. [5]).

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 theory 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 theory 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 $\ell \to r\ [\varphi]$. We may say that the reduction occurs at position p. A reduction step with $\mathcal{R}_{\mathtt{calc}}$ is called a *calculation*.

Now we define a *logically constrained term rewriting system* (an LCTRS, for short) as the abstract rewriting system $(T(\Sigma, \mathcal{V}), \to_\mathcal{R})$ which is simply written 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. 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 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 ([5]) 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 theory terms are $\mathsf{0} = \mathsf{0} + -\mathsf{1}$ and $x + \mathsf{3} \geq y + -\mathsf{42}$ that are constraints. $\mathsf{5} + \mathsf{9}$ is also a (ground) theory term, but not a constraint. Using calculation steps, a term $\mathsf{3} - \mathsf{1}$ reduces to $\mathsf{2}$ in one step with the calculation rule $x - y \to z\ [z = x - y]$, and $\mathsf{3} \times (\mathsf{2} \times (\mathsf{1} \times \mathsf{1}))$ reduces to $\mathsf{6}$ in three steps. To implement an LCTRS calculating the *factorial* function, we use the signature Σ above and the following rules: $\mathcal{R}_{\mathsf{fact}} = \{\ \mathsf{fact}(x) \to \mathsf{1}\ [x \leq \mathsf{0}],\ \mathsf{fact}(x) \to x \times \mathsf{fact}(x - \mathsf{1})\ [\neg(x \leq \mathsf{0})]\ \}$. Expected starting terms are, e.g., $\mathsf{fact}(\mathsf{42})$ or $\mathsf{fact}(\mathsf{fact}(-\mathsf{4}))$. Using the constrained rewrite rules in $\mathcal{R}_{\mathsf{fact}}$, $\mathsf{fact}(\mathsf{3})$ reduces in ten steps to $\mathsf{6}$.

2.2 SIMP$^+$: a Small Imperative Language with Global Variables and Function Calls

In this section, we recall the syntax of SIMP, a small imperative language (cf. [4]). To deal with global variables and function calls, we add them into the ordinary syntax and semantics of SIMP in a natural way. We refer to such an extended language as SIMP$^+$.

We first show the syntax adopting a C-like notation. A *program* P of SIMP$^+$ is defined by the following BNF:

$$\begin{array}{lcl}
P & ::= & D\ F \\
D & ::= & \varepsilon \mid \texttt{int}\ v\ =\ n;\ D \\
F & ::= & \varepsilon \mid \texttt{int}\ f(\texttt{int}\ x_1,\ldots,\texttt{int}\ x_m)\ =\ \{\ D\ S\ \texttt{return}\ E;\ \}\ F \\
S & ::= & \varepsilon \mid v\ =\ E;\ S \mid v\ =\ f(E,\ldots,E);\ S \mid \texttt{if}(B)\{S\}\texttt{else}\{S\}\ S \mid \texttt{while}(B)\{S\}\ S \\
E & ::= & n \mid v \mid (E + E) \mid (E - E) \\
B & ::= & \mathsf{true} \mid \mathsf{false} \mid (E == E) \mid (E < E) \mid (\neg B) \mid (B \vee B)
\end{array}$$

where $n \in \mathbb{Z}$, $v \in \mathcal{V}$, f is a function name, and we may omit brackets in the usual way. The empty sequence "ε" is used instead of the "skip" command. To simplify discussion, we do not use other operands such as multiplication and division, but we use $\neq$, $\leq$, $>$, $\geq$, $\wedge$, $\implies$, etc, as syntactic sugars. We also use the `for`-statement as a syntactic sugar. We assume that a function name f has a fixed arity, and the definition and call of f are consistent with the arity. A program P consists of declarations of *global variables* (with initialization) and functions. For a program P, we denote the set of global variables appearing in P by $\mathcal{GVar}(P)$: let P be $\texttt{int}\ x_1\ =\ n_1;\ldots;\texttt{int}\ x_k\ =\ n_k;\texttt{int}\ f(\ldots)\ =\ \{\ldots\}\ \ldots$, then $\mathcal{GVar}(P) = \{x_1,\ldots,x_n\}$. We assume that each function f is defined at most once in a program P and any function called in a function defined in P is defined in P. To simplify the semantics, we assume that local variables in function declarations are different from global variables and parameters of functions.

```
int num = 0;                            int main(){
                                          int z = 3;
int sum(int x){                           z = sum(z);
  int z = 0;                              return 0;
  num = num + 1;                        }
  if( x <= 0 ){
    z = 0;
  }else{
    z = sum(x - 1);
    z = x + z;
  }
  return z;
}
```

Figure 3: a SIMP^+ program P_1 obtained by adding the definition of `main` into the program for `sum`.

An *assignment* is defined by a substitution whose range is over the integers, which may be used for terms in the setting of LCTRSs. We deal with SIMP^+ programs that can be successfully compiled as C programs.

Example 2.2 The program P_1 in Figure 3 is a SIMP^+ program, and we have that $\mathcal{GV}ar(P_1) = \{num\}$.

The semantics $\Downarrow_{\text{calc}}$ of integer and boolean expressions is defined as usual (see Figure 4): given an expression e and an assignment σ with $\mathcal{D}om(\sigma) \supseteq \mathcal{V}ar(e)$, we write $(e, \sigma) \Downarrow_{\text{calc}} v$ where v is the resulting value obtained by evaluating e with σ. The *transition system* defining the semantics of a SIMP^+ program P is defined by

- *configurations* of the form $\langle \alpha,\, \sigma_0,\, \sigma_1 \rangle$, where
  - α is of the form "$\delta\ \beta$" with variable declarations δ,[1] and a statement β, and
  - σ_0, σ_1 are assignments for global and local variables, respectively, which are represented by partial functions from variables to integers—the *update* $\sigma[x \mapsto n]$ of an assignment σ w.r.t. x for an integer n is defined as follows: if $x = y$ then $\sigma[x \mapsto n](y) = n$, and otherwise, $\sigma[x \mapsto n](y) = \sigma(y)$,

  and
- a *transition relation* $\Downarrow_P$ between configurations, which is defined as a *big-step* semantics by the inference rules illustrated in Figure 5.

We assume that for any configuration $\langle \alpha,\, \sigma_0,\, \sigma_1 \rangle$ for a program P, the assignment σ_0 is defined for all global variables of P. To compute the result of a function call $f(e_1, \ldots, e_m)$ under assignments σ_0, σ_1 for $\mathcal{GV}ar(P)$ and $\mathcal{V}ar(e_1, \ldots, e_m) \setminus \mathcal{GV}ar(P)$, given a fresh variable x, we start with the configuration $\langle x\ =\ f(e_1, \ldots, e_m),\, \sigma_0,\, \sigma_1[x \mapsto 0] \rangle$. When $\langle x\ =\ f(e_1, \ldots, e_m),\, \sigma_0,\, \sigma_1[x \mapsto 0] \rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1' \rangle$ holds, the execution halts and the result of the function call $f(e_1, \ldots, e_m)$ under σ_0, σ_1 is $\sigma_1'(x)$.

[1] Variable declarations δ may be the empty sequence.

$$\frac{n \in \mathbb{Z}}{(n,\sigma) \Downarrow_{\text{calc}} n} \qquad \frac{x \in \mathcal{V}}{(x,\sigma) \Downarrow_{\text{calc}} \sigma(x)}$$

$$\frac{(e_1,\sigma) \Downarrow_{\text{calc}} n_1 \quad (e_2,\sigma) \Downarrow_{\text{calc}} n_2 \quad n_1 \bowtie n_2 = n \in \mathbb{Z} \quad \bowtie \in \{+,-\}}{(e_1 \bowtie e_2,\sigma) \Downarrow_{\text{calc}} n}$$

$$\frac{(e_1,\sigma) \Downarrow_{\text{calc}} n_1 \quad (e_2,\sigma) \Downarrow_{\text{calc}} n_2 \quad n_1 = n_2}{(e_1 == e_2,\sigma) \Downarrow_{\text{calc}} \mathsf{true}} \qquad \frac{(e_1,\sigma) \Downarrow_{\text{calc}} n_1 \quad (e_2,\sigma) \Downarrow_{\text{calc}} n_2 \quad n_1 \neq n_2}{(e_1 == e_2,\sigma) \Downarrow_{\text{calc}} \mathsf{false}}$$

$$\frac{(e_1,\sigma) \Downarrow_{\text{calc}} n_1 \quad (e_2,\sigma) \Downarrow_{\text{calc}} n_2 \quad n_1 < n_2}{(e_1 < e_2,\sigma) \Downarrow_{\text{calc}} \mathsf{true}} \qquad \frac{(e_1,\sigma) \Downarrow_{\text{calc}} n_1 \quad (e_2,\sigma) \Downarrow_{\text{calc}} n_2 \quad n_1 \geq n_2}{(e_1 < e_2,\sigma) \Downarrow_{\text{calc}} \mathsf{false}}$$

$$\frac{(\varphi,\sigma) \Downarrow_{\text{calc}} \mathsf{false}}{(\neg\varphi,\sigma) \Downarrow_{\text{calc}} \mathsf{true}} \qquad \frac{(\varphi,\sigma) \Downarrow_{\text{calc}} \mathsf{true}}{(\neg\varphi,\sigma) \Downarrow_{\text{calc}} \mathsf{false}}$$

$$\frac{(\varphi_1,\sigma) \Downarrow_{\text{calc}} b_1 \quad (\varphi_2,\sigma) \Downarrow_{\text{calc}} b_2 \quad \mathsf{true} \in \{b_1,b_2\}}{(\varphi_1 \vee \varphi_2),\sigma) \Downarrow_{\text{calc}} \mathsf{true}} \qquad \frac{(\varphi_1,\sigma) \Downarrow_{\text{calc}} \mathsf{false} \quad (\varphi_2,\sigma) \Downarrow_{\text{calc}} \mathsf{false}}{(\varphi_1 \vee \varphi_2),\sigma) \Downarrow_{\text{calc}} \mathsf{false}}$$

Figure 4: the inference rules for the semantics of SIMP^+ expressions.

$$\frac{}{\langle \varepsilon,\, \sigma_0,\, \sigma_1\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0,\, \sigma_1\rangle} \qquad \frac{\langle \beta,\, \sigma_0,\, \sigma_1[x \mapsto n]\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1'\rangle}{\langle \texttt{int}\ x\ =\ n;\ \beta,\, \sigma_0,\, \sigma_1\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1'\rangle}$$

$$\frac{(e,\sigma_0 \cup \sigma_1) \Downarrow_{\text{calc}} n \quad x \in \mathcal{GV}ar(P) \quad \langle \beta,\, \sigma_0[x \mapsto n],\, \sigma_1\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1'\rangle}{\langle x\ =\ e;\ \beta,\, \sigma_0,\, \sigma_1\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1'\rangle}$$

$$\frac{(e,\sigma_0 \cup \sigma_1) \Downarrow_{\text{calc}} n \quad x \notin \mathcal{GV}ar(P) \quad \langle \beta,\, \sigma_0,\, \sigma_1[x \mapsto n]\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1'\rangle}{\langle x\ =\ e;\ \beta,\, \sigma_0,\, \sigma_1\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1'\rangle}$$

$$\frac{(\varphi,\sigma_0 \cup \sigma_1) \Downarrow_{\text{calc}} \mathsf{true} \quad \langle \alpha_1\ \beta,\, \sigma_0,\, \sigma_1\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1'\rangle}{\langle \texttt{if(}\,\varphi\,\texttt{)\{}\ \alpha_1\ \texttt{\}else\{}\ \alpha_2\ \texttt{\}}\ \beta,\, \sigma_0,\, \sigma_1\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1'\rangle}$$

$$\frac{(\varphi,\sigma_0 \cup \sigma_1) \Downarrow_{\text{calc}} \mathsf{false} \quad \langle \alpha_2\ \beta,\, \sigma_0,\, \sigma_1\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1'\rangle}{\langle \texttt{if(}\,\varphi\,\texttt{)\{}\ \alpha_1\ \texttt{\}else\{}\ \alpha_2\ \texttt{\}}\ \beta,\, \sigma_0,\, \sigma_1\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1'\rangle}$$

$$\frac{(\varphi,\sigma_0 \cup \sigma_1) \Downarrow_{\text{calc}} \mathsf{true} \quad \langle \alpha,\, \sigma_0,\, \sigma_1\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1'\rangle \quad \langle \texttt{while(}\,\varphi\,\texttt{)\{}\ \alpha\ \texttt{\}}\ \beta,\, \sigma_0',\, \sigma_1'\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0'',\, \sigma_1''\rangle}{\langle \texttt{while(}\,\varphi\,\texttt{)\{}\ \alpha\ \texttt{\}}\ \beta,\, \sigma_0,\, \sigma_1\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0'',\, \sigma_1''\rangle}$$

$$\frac{(\varphi,\sigma_0 \cup \sigma_1) \Downarrow_{\text{calc}} \mathsf{false} \quad \langle \beta,\, \sigma_0,\, \sigma_1\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1'\rangle}{\langle \texttt{while(}\,\varphi\,\texttt{)\{}\ \alpha\ \texttt{\}}\ \beta,\, \sigma_0,\, \sigma_1\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1'\rangle}$$

$$\frac{\forall i.\ (e_i,\sigma_0 \cup \sigma_1) \Downarrow_{\text{calc}} n_i \quad \langle \alpha,\, \sigma_0,\, \sigma_2\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1'\rangle \quad (e,\sigma_0' \cup \sigma_1') \Downarrow_{\text{calc}} n \quad \langle \beta,\, \sigma_0'',\, \sigma_1''\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0''',\, \sigma_1'''\rangle}{\langle x\ =\ f(e_1,\ldots,e_m);\ \beta,\, \sigma_0,\, \sigma_1\rangle \Downarrow_P \langle \varepsilon,\, \sigma_0''',\, \sigma_1'''\rangle}$$

where

- $\texttt{int}\ f(\texttt{int}\ y_1,\ldots,\texttt{int}\ y_m)\ =\ \{\ \alpha\ \texttt{return}\ e;\ \}$ is in P,
- $\sigma_2 = \{y_1 \mapsto n_1,\ldots,y_m \mapsto n_m\}$,
- if $x \in \mathcal{GV}ar(P)$ then $\sigma_0'' = \sigma_0'[x \mapsto n]$, and otherwise $\sigma_0'' = \sigma_0'$, and
- if $x \in \mathcal{GV}ar(P)$ then $\sigma_1'' = \sigma_1$, and otherwise $\sigma_1'' = \sigma_1[x \mapsto n]$

Figure 5: the inference rules for the semantics of SIMP^+ statements and variable-declarations.

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 [5] using the program P_1 in Figure 3. Unlike $\mathcal{R}_2$ in Section 1, in the following, we do not optimize generated rewrite rules in LCTRSs in order to make it easier to understand how to precisely transform programs. The program P_1 is transformed into the following LCTRS with the sort set $\{int, bool, state\}$ and the standard integer signature Σ^{int}_{theory} [5]:

$$\mathcal{R}_3 = \left\{ \begin{array}{rcll} \mathsf{sum}(x, num) & \to & \mathsf{u}_1(x, num, 0), & \\ \mathsf{u}_1(x, num, z) & \to & \mathsf{u}_2(x, num+1, z), & \\ \mathsf{u}_2(x, num, z) & \to & \mathsf{u}_3(x, num, z) & [\quad x \leq 0 \quad], \\ \mathsf{u}_2(x, num, z) & \to & \mathsf{u}_5(x, num, z) & [\neg(x \leq 0)\,], \\ \mathsf{u}_3(x, num, z) & \to & \mathsf{u}_4(x, num, 0), & \\ \mathsf{u}_4(x, num, z) & \to & \mathsf{u}_9(x, num, z), & \\ \mathsf{u}_5(x, num, z) & \to & \mathsf{u}_6(x, num, z, \mathsf{sum}(x-1)), & \\ \mathsf{u}_6(x, num_{old}, z, \mathsf{return}(y, num_{new})) & \to & \mathsf{u}_7(x, num_{new}, y), & \\ \mathsf{u}_7(x, num, z) & \to & \mathsf{u}_8(x, num, x+z), & \\ \mathsf{u}_8(x, num, z) & \to & \mathsf{u}_9(x, num, z), & \\ \mathsf{u}_9(x, num, z) & \to & \mathsf{return}(z, num), & \\ \mathsf{main}(num) & \to & \mathsf{u}_{10}(num, 3), & \\ \mathsf{u}_{10}(num, z) & \to & \mathsf{u}_{11}(num, z, \mathsf{sum}(z, num)), & \\ \mathsf{u}_{11}(num_{old}, z, \mathsf{return}(y, num_{new})) & \to & \mathsf{u}_{12}(y, num_{new}), & \\ \mathsf{u}_{12}(num, z) & \to & \mathsf{return}(0, num) & \end{array} \right\}$$

where $\mathsf{main} : int \Rightarrow state$, $\mathsf{u}_1, \mathsf{u}_2, \mathsf{u}_3, \mathsf{u}_4, \mathsf{u}_5, \mathsf{u}_7, \mathsf{u}_8, \mathsf{u}_9 : int \times int \times int \Rightarrow state$, $\mathsf{sum}, \mathsf{u}_{10}, \mathsf{u}_{12}, \mathsf{return} : int \times int \Rightarrow state$, $\mathsf{u}_{11} : int \times int \times state \Rightarrow state$, and $\mathsf{u}_6 : int \times int \times int \times state \Rightarrow state$. The declaration of local variable `z` of `sum` is represented by the first rule of $\mathcal{R}_3$, which stores the initial value 0 in the third argument of u_1. The `if`-statement is represented by rules of u_2, u_4, and u_8; The first rule of u_2 enters the body of the `then`-statement if $x \leq 0$ holds, and the second rule of u_2 enters the body of the `else`-statement if $x \leq 0$ does not hold (i.e., $\neg(x \leq 0)$ holds); The end of the `if`-statement is represented by terms rooted by u_9, and the rules of u_4 and u_8 are used to exit the bodies of the `then`- and `else`-statements, respectively.

To represent the function call `sum(x - 1)`, the auxiliary function symbol u_6 takes the term $\mathsf{sum}(x-1, num)$ as the fourth 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 rule of u_1 increments the global variable `num`, and thus, we include the value stored in num in the result of sum by means of $\mathsf{return}(z, num_{new})$. The rule of u_6 is used after the reduction of $\mathsf{sum}(x-1, num)$, receiving the result by means of 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 passing num_{new} to the second argument of u_7. We do the same for the function call `sum(z)` in the auxiliary function symbol

$$
\begin{aligned}
\mathsf{main}(0) &\to_{\mathcal{R}_3} \mathsf{u}_{10}(0,3)\\
&\to_{\mathcal{R}_3} \mathsf{u}_{11}(0,3,\mathsf{sum}(3,0))\\
&\to_{\mathcal{R}_3} \mathsf{u}_{11}(0,3,\mathsf{u}_1(3,0,0))\\
&\to_{\mathcal{R}_3} \mathsf{u}_{11}(0,3,\mathsf{u}_2(3,0+1))\\
&\to_{\mathcal{R}_3} \mathsf{u}_{11}(0,3,\mathsf{u}_2(3,1,0))\\
&\to_{\mathcal{R}_3} \mathsf{u}_{11}(0,3,\mathsf{u}_5(3,1,0))\\
&\to_{\mathcal{R}_3} \mathsf{u}_{11}(0,3,\mathsf{u}_6(3,1,0,\mathsf{sum}(3-1,1)))\\
&\to_{\mathcal{R}_3} \mathsf{u}_{11}(0,3,\mathsf{u}_6(3,1,0,\mathsf{sum}(2,1)))\\
&\to_{\mathcal{R}_3} \cdots\\
&\to_{\mathcal{R}_3} \mathsf{u}_{11}(0,3,\mathsf{u}_7(3,4,3)))\\
&\to_{\mathcal{R}_3} \mathsf{u}_{11}(0,3,\mathsf{u}_8(3,4,3+3)))\\
&\to_{\mathcal{R}_3} \mathsf{u}_{11}(0,3,\mathsf{u}_8(3,4,6)))\\
&\to_{\mathcal{R}_3} \mathsf{u}_{11}(0,3,\mathsf{u}_9(3,4,6))\\
&\to_{\mathcal{R}_3} \mathsf{u}_{11}(0,3,\mathsf{return}(6,4))\\
&\to_{\mathcal{R}_3} \mathsf{u}_{12}(6,4)\\
&\to_{\mathcal{R}_3} \mathsf{return}(0,4)
\end{aligned}
$$

Figure 6: the reduction of $\mathcal{R}_3$ for the execution of the program for `sum`.

u_{11}. For the execution of the program, we have the reduction of $\mathcal{R}_3$ illustrated in Figure 6. Note that the global variable `num` is initialized by 0 and we started from $\mathsf{main}(0)$. From the 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 $\mathcal{R}_3$ above, the function symbol u_6 recursively calls sum in its fourth 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 transform function calls is very naive but not so general. For example, to model parallel execution, a value stored in a global variable does 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 more general settings such as parallel execution, global variables used like shared memories should be located at fixed addresses (i.e., fixed positions of terms) 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.[2] For example, the process of executing the above program is expressed as follows:

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

[2] When we execute n (> 1) processes in parallel, we make env have n extra arguments where the i-th process is executed in the i-th extra argument.

Note that env has the sort *int* $\times$ *state* $\Rightarrow$ *env*, where *env* is a new sort for environment. The first argument of env is the place where values for the global variable `num` are stored, and the second argument of env is the place where functions are executed, e.g., the main function `main` is called as 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` and the first statement of `sum` are local, and thus, we transform the definition of `main` as well as $\mathcal{R}_3$:

$$\left\{\begin{array}{r@{}l} \mathsf{main}() &\to \mathsf{u}_{10}(3), \\ \mathsf{u}_{10}(z) &\to \mathsf{u}_{11}(z, \mathsf{sum}(z)), \\ \mathsf{u}_{11}(z, \mathsf{return}(y)) &\to \mathsf{u}_{11}(y), \\ \mathsf{u}_{12}(z) &\to \mathsf{return}(0), \\ \mathsf{sum}(x) &\to \mathsf{u}_1(x, 0) \end{array}\right\}$$

The symbol return no longer contains values for the global variable `num`. In executing the program (i.e., `main`), the first access to the global variable `num` is the statement "`num = num + 1`" in the definition of sum. The initial term $\mathsf{env}(0, \mathsf{main}())$ can be reduced to $\mathsf{env}(0, \mathsf{u}_{11}(3, \mathsf{u}_1(3, 0)))$, and thus, the first execution of the statement "`num = num + 1`" can be expressed by the following rewrite rule for env:

$$\mathsf{env}(\mathit{num}, \mathsf{u}_{11}(z_0, \mathsf{u}_1(x, z))) \to \mathsf{env}(\mathit{num} + 1, \mathsf{u}_{11}(z_0, \mathsf{u}_2(x, z)))$$

The other statements in the definition of `sum` are local and we transform them into the following rules, as well as $\mathcal{R}_3$:

$$\left\{\begin{array}{r@{}ll} \mathsf{u}_2(x, z) &\to \mathsf{u}_3(x, z) & [\quad x \leq 0 \quad], \\ \mathsf{u}_2(x, z) &\to \mathsf{u}_5(x, z) & [\, \neg(x \leq 0) \,], \\ \mathsf{u}_3(x, z) &\to \mathsf{u}_4(x, 0), & \\ \mathsf{u}_4(x, z) &\to \mathsf{u}_9(x, z), & \\ \mathsf{u}_5(x, z) &\to \mathsf{u}_6(x, z, \mathsf{sum}(x - 1)), & \\ \mathsf{u}_6(x, z, \mathsf{return}(y)) &\to \mathsf{u}_7(x, y), & \\ \mathsf{u}_7(x, z) &\to \mathsf{u}_8(x, x + z), & \\ \mathsf{u}_8(x, z) &\to \mathsf{u}_9(x, z), & \\ \mathsf{u}_9(x, z) &\to \mathsf{return}(z) & \end{array}\right\}$$

Unfortunately, the above rules are not enough to capture all possible executions, e.g. the second execution of "`num = num + 1`", which is done by the second call of `sum`, is not expressed yet. Thus, we prepare the following rule:

$$\mathsf{env}(\mathit{num}, \mathsf{u}_{11}(z_0, \mathsf{u}_6(x', z', \mathsf{u}_1(x, z)))) \to \mathsf{env}(\mathit{num} + 1, \mathsf{u}_{11}(z_0, \mathsf{u}_6(x', z', \mathsf{u}_2(x, z))))$$

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

$$\mathsf{env}(\mathit{num}, \mathsf{u}_{11}(z_0, \mathsf{u}_6(x', z', \mathsf{u}_6(x'', z'', \mathsf{u}_1(x, z))))) \to \mathsf{env}(\mathit{num} + 1, \mathsf{u}_{11}(z_0, \mathsf{u}_6(x', z', \mathsf{u}_6(x'', z'', \mathsf{u}_2(x, z)))))$$

In summary, we need similar rules for all recursive calls of `sum`. The function `sum` may receive all the (finitely many) integers, and we need many similar rules, all of which express the increment of `num`. In addition, we may need other rules for the case where we add other functions calling `sum` into the program. More generally, the nesting of function calls cannot be fixed, and thus, along the above approach, we may need infinitely many rewrite rules. This means that the above approach is not adequate for recursive functions.

The troublesome observed by means of P_1 is caused by the fact that positions where sum is called are not unique in the above approach. We will show another approach to avoid this troublesome in the next section.

3.3 Using a Call Stack for Function Calls

In this section, using P_1 in Figure 3, we show a new representation of function calls for LCTRSs.

The approach to the treatment of global variables in the previous section needs finitely or infinitely 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 rooted by env are not unique due to nestings of auxiliary function symbols, one of which is running and the others are waiting. 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 $\perp : \mathit{process}$ (the empty stack). To adapt to stacks, we change the sort of env. For example, we give $\mathit{int} \times \mathit{process} \Rightarrow \mathit{env}$ to env, and the initial term for the execution of the program is the following one:

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

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 the execution (successfully) halts, we pop the frame of the form $\mathsf{return}(\ldots)$ from the stack.

Along the idea above, the statements of calling functions in P_1 in Figure 3—the rules of $\mathcal{R}_3$ related to u_6 or u_{11}—are transformed into the following rules:

$$\left\{\begin{array}{rl} \mathsf{stack}(\mathsf{u}_5(x,z),s) \to & \mathsf{stack}(\mathsf{sum}(x-1),\mathsf{stack}(\mathsf{u}_6(x,z),s)), \\ \mathsf{stack}(\mathsf{return}(y),\mathsf{stack}(\mathsf{u}_6(x,z),s)) \to & \mathsf{stack}(\mathsf{u}_7(x,y),s), \\ \mathsf{stack}(\mathsf{u}_{10}(n),s) \to & \mathsf{stack}(\mathsf{sum}(n),\mathsf{stack}(\mathsf{u}_{11}(n),s)), \\ \mathsf{stack}(\mathsf{return}(y),\mathsf{stack}(\mathsf{u}_{11}(n))) \to & \mathsf{stack}(\mathsf{u}_3(n),s) \end{array}\right\}$$

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

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

Note that s in the above rule is a variable. The statement “`num = num + 1`” in P_1—the rule of $\mathcal{R}_3$ to increment *num*—is transformed into the following rule:

$$\mathsf{env}(\mathit{num},\mathsf{stack}(\mathsf{u}_1(x,z),s)) \to \mathsf{env}(\mathit{num}+1,\mathsf{stack}(\mathsf{u}_2(x,z),s))$$

In summary, P_1 is transformed into the following LCTRS with the sort set $\{\mathit{int},\mathit{bool},\mathit{state},\mathit{env},\mathit{process}\}$

$$\begin{array}{rl}
\mathsf{env}(0,\mathsf{stack}(\mathsf{main}(),\bot)) & \to_{\mathcal{R}_4} \mathsf{env}(0,\mathsf{stack}(\mathsf{u}_{10}(3),\bot)) \\
& \to_{\mathcal{R}_4} \mathsf{env}(0,\mathsf{stack}(\mathsf{sum}(3),\mathsf{stack}(\mathsf{u}_{11}(3),\bot))) \\
& \to_{\mathcal{R}_4} \mathsf{env}(0,\mathsf{stack}(\mathsf{u}_1(3,0),\mathsf{stack}(\mathsf{u}_{11}(3),\bot))) \\
& \to_{\mathcal{R}_4} \mathsf{env}(0+1,\mathsf{stack}(\mathsf{u}_2(3,0),\mathsf{stack}(\mathsf{u}_{11}(3),\bot))) \\
& \to_{\mathcal{R}_4} \mathsf{env}(1,\mathsf{stack}(\mathsf{u}_2(3,0),\mathsf{stack}(\mathsf{u}_{11}(3),\bot))) \\
& \to_{\mathcal{R}_4} \mathsf{env}(1,\mathsf{stack}(\mathsf{u}_5(3,0),\mathsf{stack}(\mathsf{u}_{11}(3),\bot))) \\
& \to_{\mathcal{R}_4} \mathsf{env}(1,\mathsf{stack}(\mathsf{sum}(3-1),\mathsf{stack}(\mathsf{u}_6(3,0),\mathsf{stack}(\mathsf{u}_{11}(3),\bot)))) \\
& \to_{\mathcal{R}_4} \mathsf{env}(1,\mathsf{stack}(\mathsf{sum}(2),\mathsf{stack}(\mathsf{u}_6(3,0),\mathsf{stack}(\mathsf{u}_{11}(3),\bot)))) \\
& \to_{\mathcal{R}_4} \cdots \\
& \to_{\mathcal{R}_4} \mathsf{env}(4,\mathsf{stack}(\mathsf{u}_7(3,3),\mathsf{stack}(\mathsf{u}_{11}(3),\bot))) \\
& \to_{\mathcal{R}_4} \mathsf{env}(4,\mathsf{stack}(\mathsf{u}_8(3,3+3),\mathsf{stack}(\mathsf{u}_{11}(3),\bot))) \\
& \to_{\mathcal{R}_4} \mathsf{env}(4,\mathsf{stack}(\mathsf{u}_8(3,6),\mathsf{stack}(\mathsf{u}_{11}(3),\bot))) \\
& \to_{\mathcal{R}_4} \mathsf{env}(4,\mathsf{stack}(\mathsf{u}_9(3,6),\mathsf{stack}(\mathsf{u}_{11}(3),\bot))) \\
& \to_{\mathcal{R}_4} \mathsf{env}(4,\mathsf{stack}(\mathsf{return}(6),\mathsf{stack}(\mathsf{u}_{11}(3),\bot))) \\
& \to_{\mathcal{R}_4} \mathsf{env}(4,\mathsf{stack}(\mathsf{u}_{12}(6,\bot))) \\
& \to_{\mathcal{R}_4} \mathsf{env}(4,\mathsf{stack}(\mathsf{return}(0)))
\end{array}$$

Figure 7: the reduction of $\mathcal{R}_4$ for the execution of the program for `sum`.

and the standard integer signature Σ^{int}_{theory}:

$$\mathcal{R}_4 = \left\{ \begin{array}{rll}
\mathsf{sum}(x) & \to \mathsf{u}_1(x,0), & \\
\mathsf{env}(num,\mathsf{stack}(\mathsf{u}_1(x,z),s)) & \to \mathsf{env}(num+1,\mathsf{stack}(\mathsf{u}_2(x,z),s)), & \\
\mathsf{u}_2(x,z) & \to \mathsf{u}_3(x,z) & [\ \ x \leq 0\ \ ], \\
\mathsf{u}_2(x,z) & \to \mathsf{u}_5(x,z) & [\, \neg(x \leq 0)\,], \\
\mathsf{u}_3(x,z) & \to \mathsf{u}_4(x,0), & \\
\mathsf{u}_4(x,z) & \to \mathsf{u}_9(x,z), & \\
\mathsf{stack}(\mathsf{u}_5(x,z),s) & \to \mathsf{stack}(\mathsf{sum}(x-1),\mathsf{stack}(\mathsf{u}_6(x,z),s)), & \\
\mathsf{stack}(\mathsf{return}(y),\mathsf{stack}(\mathsf{u}_6(x,z),s)) & \to \mathsf{stack}(\mathsf{u}_7(x,y),s), & \\
\mathsf{u}_7(x,z) & \to \mathsf{u}_8(x,x+z), & \\
\mathsf{u}_8(x,z) & \to \mathsf{u}_9(x,z), & \\
\mathsf{u}_9(x,z) & \to \mathsf{return}(z), & \\
\mathsf{main}() & \to \mathsf{u}_{10}(3), & \\
\mathsf{stack}(\mathsf{u}_{10}(z),s) & \to \mathsf{stack}(\mathsf{sum}(z),\mathsf{stack}(\mathsf{u}_{11}(z),s)), & \\
\mathsf{stack}(\mathsf{return}(y),\mathsf{stack}(\mathsf{u}_{11}(z))) & \to \mathsf{stack}(\mathsf{u}_{12}(y),s), & \\
\mathsf{u}_{12}(z) & \to \mathsf{return}(0) &
\end{array} \right\}$$

For the execution of the program, we have the reduction of $\mathcal{R}_4$ illustrated in Figure 7.

The function symbol stack is a defined symbol of $\mathcal{R}_4$, 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, we generate $\mathsf{env}(\vec{x},\mathsf{stack}(t,s)) \to \mathsf{env}(\vec{x},\mathsf{stack}(t',s'))$ instead of $\mathsf{stack}(t,s) \to \mathsf{stack}(t',s')$.

4 Formalizing the Transformation Using Stacks

In this section, we formalize the idea of using call stacks, which is illustrated in Section 3, showing a precise transformation of SIMP^+ programs into LCTRSs.

In the following, we deal with a SIMP^+ program P which is of the following form:

$$\begin{array}{l} \mathtt{int}\, x_1 \;=\; n_1;\; \ldots;\; \mathtt{int}\, x_k \;=\; n_k; \\ \mathtt{int}\; \mathsf{f}_1(\mathtt{int}\, y_{1,1},\ldots,\mathtt{int}\, y_{1,m_1})\ \{\ \alpha_1\ \ \mathtt{return}\, e_1;\ \} \\ \quad \ldots \\ \mathtt{int}\; \mathsf{f}_{k'}(\mathtt{int}\, y_{k',1},\ldots,\mathtt{int}\, y_{k',m_{k'}})\ \{\ \alpha_{k'}\ \ \mathtt{return}\, e_{k'};\ \} \end{array} \tag{1}$$

where $\alpha_1,\ldots,\alpha_{k'}$ are statements with local-variable declarations and no function other than $\mathsf{f}_1,\ldots,\mathsf{f}_{k'}$ is called in $\alpha_1,\ldots,\alpha_{k'}$. Note that $\mathsf{f}_1,\ldots,\mathsf{f}_{k'}$ may be self- or mutually recursive. We abuse integer and boolean expressions of SIMP^+ programs as theory terms and formulas, respectively, over the standard integer signature Σ^{int}_{theory}. In the following, we denote the sequences $x_1,\ldots,x_k$ and $y_{i,1},\ldots,y_{i,m_i}$ by $\vec{x}$ and $\overrightarrow{y_i}$, respectively, and the notation $\overrightarrow{y}$ stands for $\overrightarrow{y_i}$ for some $i \in \{1,\ldots,k'\}$.

First, we define an auxiliary function aux_P that takes a term t, a statement β with variable declarations, and a non-negative integer i as input, and returns a triple $(u,\mathcal{R}_\beta,j)$ of a term u, a set $\mathcal{R}_\beta$ of constrained rewrite rules, and a non-negative integer j. The resulting rewrite rules in $\mathcal{R}_\beta$ reduce an instance of $\mathsf{env}(\vec{x},\mathsf{stack}(t,s))$ to an instance of $\mathsf{env}(\vec{x},\mathsf{stack}(u,s))$: if the instance of $\mathsf{env}(\vec{x},\mathsf{stack}(t,s))$ corresponds to a configuration $\langle\beta,\,\sigma,\,\sigma'\rangle$, then the instance of $\mathsf{env}(\vec{x},\mathsf{stack}(u,s))$ corresponds to a configuration $\langle\varepsilon,\,\sigma'',\,\sigma'''\rangle$ such that $\langle\beta,\,\sigma,\,\sigma'\rangle \Downarrow_P \langle\varepsilon,\,\sigma'',\,\sigma'''\rangle$. The input term t is of the form either $\mathsf{f}_{k''}(\overrightarrow{y_{k''}})$ or $\mathsf{u}_{i'}(\overrightarrow{y_{k''}},z_{k'',1},\ldots,z_{k'',m'_{k''}})$ where $z_{k'',1},\ldots,z_{k'',m'_{k''}}$ are locally declared variables in $\alpha_{k''}$ and $\mathsf{u}_{i'}$ is a newly introduced function symbol with $i' < i < j$. The resulting term u is of the form of $\mathsf{u}_{j'}(\overrightarrow{y_{k''}},z_{k'',1},\ldots,z_{k'',m''_{k''}})$ where $m'_{k''} \le m''_{k''}$, $z_{k'',1},\ldots,z_{k'',m''_{k''}}$ are locally declared variables in $\alpha_{k''}$, and $\mathsf{u}_{j'}$ is a newly introduced function symbol with $i \le j' < j$. In the following, we denote the sequence $z_{k'',1},\ldots,z_{k'',m'_{k''}}$ by $\overrightarrow{z_{k''}}$, and the sequence $e'_1,\ldots,e'_{m_i}$ of integer expressions by $\overrightarrow{e'_i}$, and the notation $\overrightarrow{z}$ stands for $\overrightarrow{z_{k''}}$ for some $k'' \in \{1,\ldots,k'\}$.

Definition 4.1 *The auxiliary function aux_P is defined as follows:*

- $aux_P(t,\ \varepsilon,\ i) = (t,\emptyset,i)$,
- $aux_P(g(\overrightarrow{y},\overrightarrow{z}),\ \mathtt{int}\ z' \;=\; n;\ \beta,\ i) = (u,\{\ g(\overrightarrow{y},\overrightarrow{z}) \to \mathsf{u}_i(\overrightarrow{y},\overrightarrow{z},n)\ \}\cup\mathcal{R}_\beta,j)$, *where*
  - $aux_P(\mathsf{u}_i(\overrightarrow{y},\overrightarrow{z},z'),\beta,i+1) = (u,\mathcal{R}_\beta,j)$,
- $aux_P(g(\overrightarrow{y},\overrightarrow{z}),\ z' \;=\; e;\ \beta,\ i) = (u,\{\ C[g(\overrightarrow{y},\overrightarrow{z})] \to (C[\mathsf{u}_i(\overrightarrow{y},\overrightarrow{z})])\{z' \mapsto e\}\ \}\cup\mathcal{R}_\beta,j)$ *if e is an integer expression, where*
  - *if* $\{\vec{x}\}\cap(\{z'\}\cup\mathcal{V}ar(e)) \ne \emptyset$ *then* $C[\,] = \mathsf{env}(\vec{x},\mathsf{stack}(\Box,w))$ *with a fresh variable* $w \notin \{\vec{x},\overrightarrow{y},\overrightarrow{z}\}$, *and otherwise* $C[\,] = \Box$, *and*
  - $aux_P(\mathsf{u}_i(\overrightarrow{y},\overrightarrow{z},z'),\beta,i+1) = (u,\mathcal{R}_\beta,j)$,
- $aux_P(g(\overrightarrow{y},\overrightarrow{z}),\ z' \;=\; \mathsf{f}_{k''}(\overrightarrow{e'_{k''}});\ \beta,\ i) =$

$$(u,\left\{\begin{array}{r} C[\mathsf{stack}(g(\overrightarrow{y},\overrightarrow{z}),w)] \to C[\mathsf{stack}(\mathsf{f}_{k''}(\overrightarrow{e'_{k''}}),\mathsf{stack}(\mathsf{u}_i(\overrightarrow{y},\overrightarrow{z}),w))], \\ C'[\mathsf{stack}(\mathsf{return}(z''),\mathsf{stack}(\mathsf{u}_i(\overrightarrow{y},\overrightarrow{z}),w))] \to (C'[\mathsf{stack}(\mathsf{u}_{i+1}(\overrightarrow{y},\overrightarrow{z}),w)])\{z' \mapsto z''\} \end{array}\right\}\cup\mathcal{R}_\beta,j)$$

where

– w, z'' *are different fresh variables not in* $\{\vec{x}, \overrightarrow{y}, \overrightarrow{z}\}$,
– *if* $\{\vec{x}\} \cap \mathcal{V}ar(\overrightarrow{e'_{k''}}) \neq \emptyset$ *then* $C[\,] = \mathsf{env}(\vec{x}, \Box)$, *and otherwise* $C[\,] = \Box$,
– *if* $z' \in \{\vec{x}\}$ *then* $C'[\,] = \mathsf{env}(\vec{x}, \Box)$, *and otherwise* $C'[\,] = \Box$, *and*
– $aux_P(\mathsf{u}_{i+1}(\overrightarrow{y}, \overrightarrow{z}), \beta, i+2) = (u, \mathcal{R}_\beta, j)$,

- $aux_P(g(\overrightarrow{y}, \overrightarrow{z}),\ \texttt{if(}\varphi\texttt{)\{}\beta_1\texttt{\}else\{}\beta_2\texttt{\}}\ \beta,\ i) =$

$$(u, \left\{\begin{array}{lll} C[g(\overrightarrow{y}, \overrightarrow{z})] \to C[\mathsf{u}_i(\overrightarrow{y}, \overrightarrow{z})] & [\ \varphi\ ], & u_1 \to \mathsf{u}_{j_2}(\overrightarrow{y}, \overrightarrow{z}), \\ C[g(\overrightarrow{y}, \overrightarrow{z})] \to C[\mathsf{u}_{j_1+1}(\overrightarrow{y}, \overrightarrow{z})] & [\neg\varphi\,], & u_2 \to \mathsf{u}_{j_2}(\overrightarrow{y}, \overrightarrow{z}) \end{array}\right\} \cup \mathcal{R}_{\beta_1} \cup \mathcal{R}_{\beta_2} \cup \mathcal{R}_\beta, j)$$

where

– $aux_P(\mathsf{u}_i(\overrightarrow{y}, \overrightarrow{z}), \beta_1, i+1) = (u_1, \mathcal{R}_{\beta_1}, j_1)$,
– $aux_P(\mathsf{u}_{j_1+1}(\overrightarrow{y}, \overrightarrow{z}), \beta_2, j_1+1) = (u_2, \mathcal{R}_{\beta_2}, j_2)$,
– *if* $\{\vec{x}\} \cap \mathcal{V}ar(\varphi) \neq \emptyset$ *then* $C[\,] = \mathsf{env}(\vec{x}, \mathsf{stack}(\Box, w))$ *with a fresh variable* $w \notin \{\vec{x}, \overrightarrow{y}, \overrightarrow{z}\}$, *and otherwise* $C[\,] = \Box$, *and*
– $aux_P(\mathsf{u}_{j_2}(\overrightarrow{y}, \overrightarrow{z}), \beta, j_2+1) = (u, \mathcal{R}_\beta, j)$,

- $aux_P(g(\overrightarrow{y}, \overrightarrow{z}),\ \texttt{while(}\varphi\texttt{)\{}\alpha\texttt{\}}\ \beta,\ i) =$

$$(u', \left\{\begin{array}{lll} C[g(\overrightarrow{y}, \overrightarrow{z})] \to C[\mathsf{u}_i(\overrightarrow{y}, \overrightarrow{z})] & [\ \varphi\ ], & u \to g(\overrightarrow{y}, \overrightarrow{z}), \\ C[g(\overrightarrow{y}, \overrightarrow{z})] \to C[\mathsf{u}_j(\overrightarrow{y}, \overrightarrow{z})] & [\neg\varphi\ ] & \end{array}\right\} \cup \mathcal{R}_\alpha \cup \mathcal{R}_\beta, j')$$

where

– $aux_P(\mathsf{u}_i(\overrightarrow{y}, \overrightarrow{z}), \alpha, i+1) = (u, \mathcal{R}_\alpha, j)$,
– *if* $\{\vec{x}\} \cap \mathcal{V}ar(\varphi) \neq \emptyset$ *then* $C[\,] = \mathsf{env}(\vec{x}, \mathsf{stack}(\Box, w))$ *with a fresh variable* $w \notin \{\vec{x}, \overrightarrow{y}, \overrightarrow{z}\}$, *and otherwise* $C[\,] = \Box$. *and*
– $aux_P(\mathsf{u}_j(\overrightarrow{y}, \overrightarrow{z}), \beta, j+1) = (u', \mathcal{R}_\beta, j')$,

The sorts of generated symbols are determined as follows: $\mathsf{f}_1, \ldots, \mathsf{f}_{k'}, \mathsf{u}_i, \mathsf{u}_{i+1}, \ldots : int \times \cdots \times int \Rightarrow state$, $\mathsf{return} : int \Rightarrow state$, $\mathsf{env} : int \times \cdots \times int \times process \Rightarrow env$, $\mathsf{stack} : state \times process \Rightarrow process$, *and* $\bot : process$.

Using aux_P, the transformation illustrated in Section 3 is defined as follows.

Definition 4.2 *We define conv by* $conv(P) = \bigcup_{i=1}^{k'}(\mathcal{R}_i \cup \{\ C_i[u_i] \to C_i[\mathsf{return}(e_i)]\ \})$, *where* $j_1 = 1$[3] *and for each* $i \in \{1, \ldots, k'\}$,

- $aux_P(\mathsf{f}_i(\overrightarrow{y_i}), \alpha_i, j_i) = (u_i, \mathcal{R}_i, j_{i+1})$, *and*
- *if* $\{\vec{x}\} \cap \mathcal{V}ar(e_i) \neq \emptyset$ *then* $C_i[\,] = \mathsf{env}(\vec{x}, \mathsf{stack}(\Box, w))$ *with a fresh variable* $w \notin \{\vec{x}\} \cup \mathcal{V}ar(u_i)$, *and otherwise* $C_i[\,] = \Box$.

By definition, it is clear that $conv(P)$ is an LCTRS with the sort set $\{int, bool, state, env, process\}$ and the standard integer signature Σ^{int}_{theory}. Note that Definitions 4.1 and 4.2 follow the formulation in [6]. Note also that $\mathcal{R}$ is orthogonal, any term reachable from $(\mathsf{env}(\vec{x}, \mathsf{stack}(\mathsf{f}_i(\overrightarrow{y_i}), s)))(\sigma_0 \cup \sigma_1)$ with a normal form s has at most one redex that is not for $\mathcal{R}_{\mathtt{calc}}$.[4] Since the reduction of $\mathcal{R}_{\mathtt{calc}}$ is convergent, we restrict the reduction of $\mathcal{R}_{\mathtt{calc}}$ to the *leftmost* one. Then, any subderivation $t \to^*_{\mathcal{R}} t'$ of a derivation from $(\mathsf{env}(\vec{x}, \mathsf{stack}(\mathsf{f}_i(\overrightarrow{y_i}), s)))(\sigma_0 \cup \sigma_1)$ has at most one pass from t to t'.

[3] The third argument of aux_P is used to generate new function symbols of the form u_i. We do not have to start with 1, and we can put any non-negative integer into the third argument of aux_P in order to, e.g., avoid the introduction of the same function symbol for two different inputs.

[4] More precisely, the redex of a term reachable from $(\mathsf{env}(\vec{x}, \mathsf{stack}(\mathsf{f}(\overrightarrow{y}), s)))(\sigma_0 \cup \sigma_1)$ is at the root position, position $k+1$, position $(k+1).1$, or position $(k+1).1.p$ for some p.

Example 4.3 Consider the program P_1 in Figure 3. We have that $conv(P_1) = \mathcal{R}_4$.

Finally, we show correctness of the transformation *conv*. Recall that P is assumed to be of the form (1). We first show two auxiliary lemmas.

Lemma 4.4 *Let $\mathcal{R}$ be an LCTRS, e an integer expression, n an integer, and σ an assignments for $\mathcal{V}ar(e)$. Then, $(e,\sigma) \Downarrow_{\mathrm{calc}} n$ if and only if $e\sigma \to^*_{\mathcal{R}} n$.*

Proof. Trivial by the definitions of $\Downarrow_{\mathrm{calc}}$ and $\mathcal{R}_{\mathtt{calc}}$. $\square$

Lemma 4.5 (Correctness of aux_P**)** *Let $\mathcal{R} = conv(P)$, and β a substatement of α_i for some $i \in \{1,\ldots,k'\}$ (i.e., β appears in α_i). Then, both of the following hold:*

(a) $aux_P(t,\beta,i')$ for any t and i' is defined, and

(b) $aux_P(t,\beta,i')$ for some t and i' is computed during the computation of $conv(P)$.

Suppose that $aux_P(g(\overrightarrow{y},\overrightarrow{z}),\beta,i')$ is computed for $conv(P)$. Let $aux_P(g(\overrightarrow{y},\overrightarrow{z}),\beta,i') = (u,\mathcal{R}_\beta,j)$, and s be a normal form of $\mathcal{R}$, σ_0,σ_0' assignments for $\mathcal{GV}ar(P)$, and σ_1,σ_1' assignments for $\{\overrightarrow{y},\overrightarrow{z}\} \cup (\mathcal{V}ar(\beta) \setminus \{\vec{x}\})$. Then, both of the following hold:

(c) $\mathcal{R}_\beta \subseteq \mathcal{R}$,

(d) $\langle \beta,\, \sigma_0,\, \sigma_1 \rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1' \rangle$ if and only if

$$(\mathsf{env}(\vec{x},\mathsf{stack}(g(\overrightarrow{y},\overrightarrow{z}),s))(\sigma_0 \cup \sigma_1) \to^*_{\mathcal{R}} (\mathsf{env}(\vec{x},\mathsf{stack}(u,s)))(\sigma_0' \cup \sigma_1').$$

Proof. By definition, it is clear that (a)–(c) hold. Using Lemma 4.4, the *only-if* and *if* parts of (d) can be proved by induction on the height of the inference for $\langle \beta,\, \sigma_0,\, \sigma_1 \rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1' \rangle$ and the length of $\to^*_{\mathcal{R}}$-steps, respectively. The difference from the proof in [6] is the treatment of global variables and function calls, while [6] adopts a small-step semantics for their imperative programs. Below, we only show the case where β is $z' \ = \ \mathsf{f}_{k''}(\overrightarrow{e'_{k''}});\ \beta'$ for some $k'' \in \{1,\ldots,k'\}$. Let $aux_P(g(\overrightarrow{y},\overrightarrow{z}),\beta,i')$ return

$$(u,\left\{\begin{array}{r} C'[\mathsf{stack}(g(\overrightarrow{y},\overrightarrow{z}),w)] \to C'[\mathsf{stack}(\mathsf{f}_{k''}(\overrightarrow{e'_{k''}}),\mathsf{stack}(\mathsf{u}_{i'}(\overrightarrow{y},\overrightarrow{z}),w))], \\ C''[\mathsf{stack}(\mathsf{return}(z''),\mathsf{stack}(\mathsf{u}_{i'}(\overrightarrow{y},\overrightarrow{z}),w))] \to (C''[\mathsf{stack}(\mathsf{u}_{i'+1}(\overrightarrow{y},\overrightarrow{z}),w)])\{z' \mapsto z''\} \end{array}\right\} \cup \mathcal{R}_{\beta'}, j)$$

where

- w, z'' are different fresh variables not in $\{\vec{x},\overrightarrow{y},\overrightarrow{z}\}$,
- if $\{\vec{x}\} \cap \mathcal{V}ar(\overrightarrow{e'_{k''}}) \neq \emptyset$ then $C'[\,] = \mathsf{env}(\vec{x},\square)$, and otherwise $C'[\,] = \square$,
- if $z' \in \{\vec{x}\}$ then $C''[\,] = \mathsf{env}(\vec{x},\square)$, and otherwise $C''[\,] = \square$, and
- $aux_P(\mathsf{u}_{i'+1}(\overrightarrow{y},\overrightarrow{z}),\beta',i'+2) = (u,\mathcal{R}_{\beta'},j)$,

Then, it follows from (c) that the above two rules are included in $\mathcal{R}$.

We first show the *only-if* part. Assume that $\langle z' \ = \ \mathsf{f}_{k''}(\overrightarrow{e'_{k''}});\ \beta',\, \sigma_0,\, \sigma_1 \rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1' \rangle$ holds with

- $(e'_i,\sigma_0 \cup \sigma_1) \Downarrow_{\mathrm{calc}} n_i$ for all $1 \leq i \leq m_{k''}$,
- $\langle \alpha_{k''},\, \sigma_0,\, \sigma_1 \rangle \Downarrow_P \langle \varepsilon,\, \sigma_0'',\, \sigma_1'' \rangle$,
- $(e_{k''},\sigma_0'' \cup \sigma_1'') \Downarrow_{\mathrm{calc}} n$, and
- $\langle \beta',\, \sigma_0''',\, \sigma_1''' \rangle \Downarrow_P \langle \varepsilon,\, \sigma_0',\, \sigma_1' \rangle$

where

- if $z' \in \mathcal{GV}ar(P)$ then $\sigma_0''' = \sigma_0''[z' \mapsto n]$, and otherwise $\sigma_0''' = \sigma_0''$, and
- if $z' \in \mathcal{GV}ar(P)$ then $\sigma_1''' = \sigma_1$, and otherwise $\sigma_1''' = \sigma_1[z' \mapsto n]$.

It follows from Lemma 4.4 and $C'[\mathsf{stack}(g(\overrightarrow{y},\overrightarrow{z}),w)] \to C'[\mathsf{stack}(\mathsf{f}_{k''}(\overrightarrow{e'_{k''}}),\mathsf{stack}(\mathsf{u}_{i'}(\overrightarrow{y},\overrightarrow{z}),w))] \in \mathcal{R}$ that

$$\begin{aligned}&(\mathsf{env}(\vec{x},\mathsf{stack}(g(\overrightarrow{y},\overrightarrow{z}),s)))(\sigma_0 \cup \sigma_1)\\ &\quad \to_{\mathcal{R}} (\mathsf{env}(\vec{x},\mathsf{stack}(\mathsf{f}_{k''}(\overrightarrow{e'_{k''}}),\mathsf{stack}(\mathsf{u}_{i'}(\overrightarrow{y},\overrightarrow{z}),s))))(\sigma_0 \cup \sigma_1)\\ &\quad = (\mathsf{env}((\vec{x})\sigma_0,\mathsf{stack}(\mathsf{f}_{k''}((\overrightarrow{e'_{k''}})(\sigma_0 \cup \sigma_1)),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s))))\\ &\quad \to_{\mathcal{R}}^* (\mathsf{env}((\vec{x})\sigma_0,\mathsf{stack}(\mathsf{f}_{k''}(n_1,\dots,n_{m_{k''}}),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s)))).\end{aligned}$$

By definition, $aux_P(\mathsf{f}_{k''}(\overrightarrow{y_{k''}}),\alpha_{k''},j_{k''})$ is computed, and let $aux_P(\mathsf{f}_{k''}(\overrightarrow{y_{k''}}),\alpha_{k''},j_{k''}) = (u_{k''},\mathcal{R}_{\alpha_{k''}},j_{k''+1})$. Then, by definition, we have that $\mathcal{R}_{\alpha_{k''}} \cup \{\, C_{k''}[u_{k''}] \to C_{k''}[\mathsf{return}(e_{k''})] \,\} \subseteq \mathcal{R}$ where $C_{k''}[]$ is a context defined in Definition 4.2. Let $\sigma_2 = \{y_1 \mapsto n_1,\ \dots,\ y_{m_{k''}} \mapsto n_{m_{k''}}\}$. Then, by the induction hypothesis, we have that

$$(\mathsf{env}(\vec{x},\mathsf{stack}(\mathsf{f}_{k''}(\overrightarrow{y_{k''}}),s')))(\sigma_0 \cup \sigma_2) \to_{\mathcal{R}}^* (\mathsf{env}(\vec{x},\mathsf{stack}(u_{k''},s')))(\sigma_0'' \cup \sigma_1'').$$

for an arbitrary term s'. Thus, we have that

$$\begin{aligned}&(\mathsf{env}((\vec{x})\sigma_0,\mathsf{stack}(\mathsf{f}_{k''}(n_1,\dots,n_{m_{k''}}),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s))))\\ &\quad = \mathsf{env}((\vec{x})\sigma_0,\mathsf{stack}(\mathsf{f}_{k''}((\overrightarrow{y_{k''}})\sigma_2),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s)))\\ &\quad \to_{\mathcal{R}}^* \mathsf{env}((\vec{x})\sigma_0'',\mathsf{stack}(u_{k''}(\sigma_0'' \cup \sigma_1''),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s)))).\end{aligned}$$

It follows from $C_{k''}[u_{k''}] \to C_{k''}[\mathsf{return}(e_{k''})] \in \mathcal{R}$ and Lemma 4.4 that

$$\begin{aligned}&\mathsf{env}((\vec{x})\sigma_0'',\mathsf{stack}(u_{k''}(\sigma_0'' \cup \sigma_1''),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s))))\\ &\quad \to_{\mathcal{R}}^* \mathsf{env}((\vec{x})\sigma_0'',\mathsf{stack}(\mathsf{return}(e_{k''}(\sigma_0'' \cup \sigma_1'')),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s))))\\ &\quad \to_{\mathcal{R}}^* \mathsf{env}((\vec{x})\sigma_0'',\mathsf{stack}(\mathsf{return}(n),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s)))).\end{aligned}$$

Since $C''[\mathsf{stack}(\mathsf{return}(z''),\mathsf{stack}(\mathsf{u}_{i'}(\overrightarrow{y},\overrightarrow{z}),w))] \to (C''[\mathsf{stack}(\mathsf{u}_{i'+1}(\overrightarrow{y},\overrightarrow{z}),w)])\{z' \mapsto z''\} \in \mathcal{R}$, we have that

$$\begin{aligned}&\mathsf{env}((\vec{x})\sigma_0'',\mathsf{stack}(\mathsf{return}(n),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s))))\\ &\quad \to_{\mathcal{R}}^* \mathsf{env}((\vec{x})\sigma_0''',\mathsf{stack}(\mathsf{u}_{i'+1}((\overrightarrow{y})\sigma_1''',(\overrightarrow{z})\sigma_1'''),s))) = (\mathsf{env}(\vec{x},\mathsf{stack}(\mathsf{u}_{i'+1}(\overrightarrow{y},\overrightarrow{z}),s))))(\sigma_0''' \cup \sigma_1''').\end{aligned}$$

By the induction hypothesis, we have that

$$(\mathsf{env}(\vec{x},\mathsf{stack}(\mathsf{u}_{i'+1}(\overrightarrow{y},\overrightarrow{z}),s))))(\sigma_0''' \cup \sigma_1''') \to_{\mathcal{R}}^* (\mathsf{env}(\vec{x},\mathsf{stack}(u,s))))(\sigma_0' \cup \sigma_1').$$

Therefore, the claim holds.

Next, we show the *if* part. Assume that

$$(\mathsf{env}(\vec{x},\mathsf{stack}(g(\overrightarrow{y},\overrightarrow{z}),s)))(\sigma_0 \cup \sigma_1) \to_{\mathcal{R}}^* (\mathsf{env}(\vec{x},\mathsf{stack}(\mathsf{u}_{i'+1}(\overrightarrow{y},\overrightarrow{z}),s)))(\sigma_0' \cup \sigma_1').$$

Then, since derivations are unique, we can let the above derivation be the following one:

$$\begin{array}{l}
(\mathsf{env}(\vec{x},\mathsf{stack}(g(\overrightarrow{y},\overrightarrow{z}),s)))(\sigma_0\cup\sigma_1)\\
\quad\to_{\mathcal{R}} (\mathsf{env}(\vec{x},\mathsf{stack}(\mathsf{f}_{k''}(\overrightarrow{e'_{k''}}),\mathsf{stack}(\mathsf{u}_{i'}(\overrightarrow{y},\overrightarrow{z}),s))))(\sigma_0\cup\sigma_1)\\
\quad\;= (\mathsf{env}((\vec{x})\sigma_0,\mathsf{stack}(\mathsf{f}_{k''}((\overrightarrow{e'_{k''}})(\sigma_0\cup\sigma_1)),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s))))\\
\quad\to^*_{\mathcal{R}} (\mathsf{env}((\vec{x})\sigma_0,\mathsf{stack}(\mathsf{f}_{k''}(n_1,\ldots,n_{m_{k''}}),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s))))\\
\quad\;= \mathsf{env}((\vec{x})\sigma_0,\mathsf{stack}(\mathsf{f}_{k''}((\overrightarrow{y_{k''}})\sigma_2),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s))))\\
\quad\to^*_{\mathcal{R}} \mathsf{env}((\vec{x})\sigma''_0,\mathsf{stack}(u_{k''}(\sigma''_0\cup\sigma''_1),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s))))\\
\quad\to^*_{\mathcal{R}} \mathsf{env}((\vec{x})\sigma''_0,\mathsf{stack}(\mathsf{return}(e_{k''}(\sigma''_0\cup\sigma''_1)),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s))))\\
\quad\to^*_{\mathcal{R}} \mathsf{env}((\vec{x})\sigma''_0,\mathsf{stack}(\mathsf{return}(n),\mathsf{stack}(\mathsf{u}_{i'}((\overrightarrow{y})\sigma_1,(\overrightarrow{z})\sigma_1),s))))\\
\quad\;= \mathsf{env}(\vec{x},\mathsf{stack}(\mathsf{return}(n),\mathsf{stack}(\mathsf{u}_{i'}(\overrightarrow{y},\overrightarrow{z}),s))))(\sigma''_0\cup\sigma''_1)\\
\quad\to_{\mathcal{R}} \mathsf{env}(\vec{x},\mathsf{stack}(\mathsf{u}_{i'+1}(\overrightarrow{y},\overrightarrow{z}),s)))(\sigma'''_0\cup\sigma'''_1)\\
\quad\to^*_{\mathcal{R}} (\mathsf{env}(\vec{x},\mathsf{stack}(u,s))))(\sigma'_0\cup\sigma'_1)
\end{array}$$

where

- $\sigma_2=\{y_1\mapsto n_1,\ \ldots,\ y_{m_{k''}}\mapsto n_{m_{k''}}\}$,
- if $z'\in\mathcal{GV}ar(P)$ then $\sigma'''_0=\sigma''_0[z'\mapsto n]$, and otherwise $\sigma'''_0=\sigma''_0$, and
- if $z'\in\mathcal{GV}ar(P)$ then $\sigma'''_1=\sigma_1$, and otherwise $\sigma'''_1=\sigma_1[z'\mapsto n]$.

It follows from Lemma 4.4 and the induction hypothesis that

- $(e_i,\sigma_0\cup\sigma_1)\Downarrow_{\mathrm{calc}} n_i$ for all $1\le i\le m$,
- $\langle\alpha_{k''},\,\sigma_0,\,\sigma_1\rangle\Downarrow_P\langle\varepsilon,\,\sigma''_0,\,\sigma''_1\rangle$,
- $(e_{k''},\sigma''_0\cup\sigma''_1)\Downarrow_{\mathrm{calc}} n$, and
- $\langle\beta',\,\sigma'''_0,\,\sigma'''_1\rangle\Downarrow_P\langle\varepsilon,\,\sigma'_0,\,\sigma'_1\rangle$

and thus, $\langle z' = \mathsf{f}_{k''}(\overrightarrow{e'_{k''}});\ \beta',\,\sigma_0,\,\sigma_1\rangle\Downarrow_P\langle\varepsilon,\,\sigma'_0,\,\sigma'_1\rangle$ holds. Therefore, the claim holds. □

Correctness of *conv* can easily be proved by using Lemmas 4.4 and 4.5.

Theorem 4.6 (Correctness of *conv***)** *Let $\mathcal{R}=conv(P)$, $n\in\mathbb{Z}$, s a normal form of $\mathcal{R}$, $i\in\{1,\ldots,k'\}$, σ_0,σ'_0 assignments for $\vec{x}$, and σ_1,σ'_1 assignments for $\overrightarrow{y_i}$. Then, $\langle\alpha_i,\,\sigma_0,\,\sigma_1\rangle\Downarrow_P\langle\varepsilon,\,\sigma'_0,\,\sigma'_1\rangle$ and $(e_i,\sigma'_0\cup\sigma'_1)\Downarrow_{\mathrm{calc}} n$ if and only if* $(\mathsf{env}(\vec{x},\mathsf{stack}(\mathsf{f}_i(\overrightarrow{y_i}),s)))(\sigma_0\cup\sigma_1)\to^*_{\mathcal{R}}(\mathsf{env}(\vec{x},\mathsf{stack}(\mathsf{return}(n),s)))(\sigma'_0\cup\sigma'_1)$.

Proof. We first show the *only-if* part. Assume that $\langle\alpha_i,\,\sigma_0,\,\sigma_1\rangle\Downarrow_P\langle\varepsilon,\,\sigma'_0,\,\sigma'_1\rangle$ and $(e_i,\sigma'_0\cup\sigma'_1)\Downarrow_{\mathrm{calc}} n$. It follows from Lemma 4.5 and $C_i[u_i]\to C_i[\mathsf{return}(e_i)]\in\mathcal{R}$ (where $C_i[\,]$ is a context defined in Definition 4.2) that

$$\begin{array}{rl}
\mathsf{env}((\vec{x})\sigma_0,\mathsf{stack}(\mathsf{f}_i((\overrightarrow{y_i})\sigma_1),s)) & \to^*_{\mathcal{R}} \mathsf{env}((\vec{x})\sigma'_0,\mathsf{stack}(u_i\sigma'_1,s))\\
& \to_{\mathcal{R}} \mathsf{env}((\vec{x})\sigma'_0,\mathsf{stack}(\mathsf{return}(e_i(\sigma'_0\cup\sigma'_1)),s)).
\end{array}$$

It follows from Lemma 4.4 that $e_i(\sigma'_0\cup\sigma'_1)\to^*_{\mathcal{R}} n$, and thus

$$\mathsf{env}((\vec{x})\sigma'_0,\mathsf{stack}(\mathsf{return}(e_i(\sigma'_0\cup\sigma'_1)),s))\to^*_{\mathcal{R}}\mathsf{env}((\vec{x})\sigma'_0,\mathsf{stack}(\mathsf{return}(n),s)).$$

Therefore, the *only-if* part holds.

Next, we show the *if* part. Assume that

$$\begin{aligned}\mathsf{env}((\vec{x})\sigma_0, \mathsf{stack}(\mathsf{f}_i((\overrightarrow{y_i})\sigma_1), s)) &\to^*_{\mathcal{R}} \mathsf{env}((\vec{x})\sigma'_0, \mathsf{stack}(u_i\sigma'_1, s)) \\ &\to^*_{\mathcal{R}} \mathsf{env}((\vec{x})\sigma'_0, \mathsf{stack}(\mathsf{return}(e_i(\sigma'_0 \cup \sigma'_1)), s)) \\ &\to^*_{\mathcal{R}} \mathsf{env}((\vec{x})\sigma'_0, \mathsf{stack}(\mathsf{return}(n), s)).\end{aligned}$$

It follows from Lemmas 4.5 and 4.4 that $\langle \alpha_i, \sigma_0, \sigma_1 \rangle \Downarrow_P \langle \varepsilon, \sigma'_0, \sigma'_1 \rangle$ and $(e_i, \sigma'_0 \cup \sigma'_1) \Downarrow_{\mathrm{calc}} n$. Therefore, the *if* part holds. □

Theorem 4.6 implies that the execution of $\mathsf{f}_i(\overrightarrow{y_i})$ with σ_0, σ_1 does not halt if and only if the reduction from $(\mathsf{env}(\vec{x}, \mathsf{stack}(\mathsf{f}_i(\overrightarrow{y_i}), s)))(\sigma_0 \cup \sigma_1)$ does not terminate. This is because by the semantics, the execution of a program never halts unsuccessfully and either successfully halts or does not halt.

5 Conclusion

In this paper, we proposed a new transformation of imperative programs with function calls and global variables into LCTRSs, and proved correctness of the transformation. A direction of future work is to apply the new transformation 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 $\mathcal{R}_4$ 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.

Acknowledgements We gratefully acknowledge the anonymous reviewers for their useful comments and suggestions to improve the paper.

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.

[4] Maribel Fernández (2014): *Programming Languages and Operational Semantics – A Concise Overview.* Undergraduate Topics in Computer Science, Springer, doi:10.1007/978-1-4471-6368-8.

[5] 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.

[6] 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/`).

[7] 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.

[8] Cynthia Kop & Naoki Nishida (2015): *Constrained Term Rewriting tooL*. In Martin Davis, Ansgar Fehnker, Annabelle McIver & Andrei Voronkov, editors: *Proceedings of the 20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, *Lecture Notes in Computer Science* 9450, Springer, pp. 549–557, doi:10.1007/978-3-662-48899-7_38.

[9] 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/`).

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

[11] 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, doi:10.4230/LIPIcs.RTA.2010.259.

[12] Uday S. Reddy (1990): *Term Rewriting Induction*. In Mark E. Stickel, editor: *Proceedings of the 10th International Conference on Automated Deduction*, *Lecture Notes in Computer Science* 449, Springer, pp. 162–177, doi:10.1007/3-540-52885-7_86.

[13] 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/`).