On Transforming Imperative Programs into Logically Constrained Term Rewrite Systems via Injective Functions from Configurations to Terms

Naoki Nishida, Misaki Kojima and Takumi Kato

August 12, 2022

On Transforming Imperative Programs into Logically Constrained Term Rewrite Systems via Injective Functions from Configurations to Terms*

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

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

Takumi Kato
Nagoya University
Nagoya, Japan

To transform an imperative program into a logically constrained term rewrite system (LCTRS, for short), previous work converts a statement list to rewrite rules in a stepwise manner, and proves the correctness along such a conversion and the big-step semantics of the program. On the other hand, the small-step semantics of a programming language comprises of inference rules that define transition steps of configurations. Partial instances of such inference rules are almost the same as rewrite rules in the transformed LCTRS. In this paper, we aim at establishing a framework for plain definitions and correctness proofs of transformations from programs into LCTRSs. To this end, for the transformation in previous work, we show an injective function from configurations to terms, and reformulate the transformation by means of the injective function. The injective function maps a transition step to a reduction step, and results in a plain correctness proof.

1 Introduction

Recently, approaches to program verification by means of logically constrained term rewrite systems (LCTRSs, for short) [8] are well investigated [4, 12, 2, 9, 5, 6]. LCTRSs are known to be useful as computation models of not only functional but also imperative programs. Especially, equivalence checking by means of LCTRSs is useful to ensure correctness of terminating functions (cf. [4]). Here, equivalence of two functions means that for every input, the functions return the same output or end with the same projection of final configurations. Previous work [4, 5, 6] for sequential programs has been extended to concurrent ones with semaphore-based exclusive control [7]. It is worth extending the transformation to more practical classes of concurrent programs, and an ultimate goal is to apply LCTRSs to verification of practical programs, e.g., automotive embedded systems. To ensure high-reliability of the verification, we have to prove the correctness of transformations in a high reliable manner, e.g., by *formalizing* them in an interactive theorem prover. On the other hand, plainer but more reliable definitions and pen-and-paper proofs are useful in formalizing them in the theorem prover.

Previous work [5] extends the transformation in [4] to programs written in SIMP$^+$, which is an extension of SIMP [3] to global variables and function calls, and shows a pen-and-paper proof for the correctness of the extended transformation under the big-step semantics of SIMP$^+$. The correctness proof is very complex compared with the simplicity of SIMP$^+$. For the ultimate goal mentioned above, such a complex framework of transformations and their correctness proofs is not desired because we would like to verify more practical and complex programs such as automotive embedded systems by means of LCTRSs. For this reason, a framework for plain definitions and correctness proofs of transformations from imperative programs into LCTRSs would be useful in extending previous work to richer fragments of programming languages.

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

Submitted to:
WPTE 2022

In this paper, we aim at establishing a framework for plain definitions and correctness proofs of transformations from imperative programs into LCTRSs. To this end, for the transformation in previous work [5], we show an injective function from configurations of SIMP$^+$ to terms (Section 4.1), and reformulate the transformation by means of the injective function (Section 4.2). The injective function maps a transition step defined by the small-step semantics of SIMP$^+$ to a reduction step of the transformed LCTRS, and results in a plainer correctness proof (Section 4.3). Missing proofs can be seen in the appendix. This paper does not propose any new transformation, but reformulate the existing one and its correctness proof, which must help us to extend the transformation to more practical programs.

2 Preliminaries

In this section, we briefly recall LCTRSs [8, 4]. Familiarity with term rewriting [1, 10] is assumed.

Let $\mathcal{S}$ be a set of *sorts* and $\mathcal{V}$ a (countably infinite) set of *variables*, each of which is equipped with a sort. A *signature* Σ disjoint from $\mathcal{V}$ is a set of *function symbols* f, each of which is equipped with a *sort declaration* $\iota_1 \times \cdots \times \iota_n \Rightarrow \iota$, written as $f : \iota_1 \times \cdots \times \iota_n \Rightarrow \iota$, where $\iota_1, \ldots, \iota_n, \iota \in \mathcal{S}$. In the rest of this section, we fix $\mathcal{S}$, Σ, and $\mathcal{V}$ and use them without notice in the paper. 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 sort ι. 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 of s, $s|_p$ denotes the subterm of s at position p, and $s[t]_p$ denotes the term obtained from s by replacing the term at position p by t, where the sorts of $s|_p$ and t coincide.

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 *domain* $\mathcal{D}om(\gamma)$ of γ is the set of variables x with $\gamma(x) \neq x$, and the *range* of γ is denoted by $\mathcal{R}an(\gamma)$. The application of γ to term s is denoted by $s\gamma$. 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 $\gamma|_X(x) = x$. For two substitutions γ and θ, their composition $\gamma\theta$ is given by $x(\gamma\theta) = \theta(\gamma(x))$ for all variables x.

To define LCTRSs, we consider the following signatures, mappings, and constants: Two signatures Σ_{term} and Σ_{theory} such that $\Sigma = \Sigma_{term} \cup \Sigma_{theory}$; a mapping $\mathcal{I}$ that assigns to each 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 $\mathcal{V}al_{\iota_1} \times \cdots \times \mathcal{V}al_{\iota_n} \Rightarrow \mathcal{V}al_\iota$; a set $\Sigma_{val,\iota} \subseteq \Sigma_{theory}$ of *value-constants* $a : \iota$ for each sort ι occurring in Σ_{theory} such that $\mathcal{J}$ gives a bijection from $\Sigma_{val,\iota}$ to $\mathcal{V}al_\iota$. 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 sorts occurring in Σ_{theory} are called *theory sorts*, and the symbols *theory symbols*. The set of theory sorts is denoted by $\mathcal{S}_{theory}$. Note that $\mathcal{S}_{theory} \subseteq \mathcal{S}$. 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 sort of *Booleans*, with $\Sigma_{val,bool} = \{\mathsf{true}, \mathsf{false}\}$ and $\mathcal{I}(bool) = \{\top, \bot\}$, $\Sigma_{core} = \Sigma_{val,bool} \cup \{\wedge, \vee, \implies : bool \times bool \Rightarrow bool,\ \neg : bool \Rightarrow bool\} \cup \{=_\iota, \neq_\iota : \iota \times \iota \Rightarrow bool \mid \iota \in \mathcal{S}_{theory}\}$, 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 substitution γ is said to *respect* a constraint φ if $\mathcal{R}an(\gamma|_{\mathcal{V}ar(\varphi)}) \subseteq \Sigma_{val}$ and $[\![\varphi\gamma]\!] = \top$. A constraint φ is said to be *valid* if all substitutions γ with $\gamma|_{\mathcal{V}ar(\varphi)} \subseteq \Sigma_{val}$ respect φ, and *satisfiable* if there exists a substitution that respects φ.

Let $\mathcal{S} \supseteq \{int, bool\}$. The *standard integer signature* Σ_{int} is $\Sigma_{core} \cup \{+, -, \times, \mathsf{exp}, \mathsf{div}, \mathsf{mod} : int \times int \Rightarrow int\} \cup \{\geq, >, \leq, < : int \times int \Rightarrow bool\} \cup \Sigma_{val,int}$ where $\Sigma_{val,int} = \{\mathsf{n} : int \mid n \in \mathbb{Z}\}$, $\mathcal{I}(int) = \mathbb{Z}$, and $\mathcal{J}(\mathsf{n}) = n$.

Note that we use n (in sans-serif font) as the function symbol for $n \in \mathbb{Z}$ (in *math* font). We define $\mathcal{J}$ in the natural way, while 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 $k < 0$.

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 $\Sigma_{term} \setminus \Sigma_{theory}$ (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 $\mathcal{V}ar(\varphi) \cup (\mathcal{V}ar(r) \setminus \mathcal{V}ar(\ell))$. A substitution γ is said to *respect* $\ell \to r\ [\varphi]$ if $\mathcal{R}an(\gamma|_{\mathcal{LV}ar(\ell \to r\ [\varphi])}) \subseteq \Sigma_{val}$ 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* (see the definition of $\to_{\mathcal{R}}$ below). Note that we do not deal with *calculation rules* [4] because for any rewrite rule in LCTRSs obtained from SIMP$^+$ programs, no calculation symbol appears in the left- or right- hand sides. 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 $\ell \to r\ [\varphi] \in \mathcal{R}$ 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 2.1 Let $\Sigma = \Sigma_{term} \cup \Sigma_{int}$, where $\Sigma_{term} = \{\ \mathsf{pow} : int \times int \Rightarrow int\ \}$. Then, both *int* and *bool* are theory sorts. Examples of theory terms are $\mathsf{0} = \mathsf{0} + -\mathsf{1}$ and $x + \mathsf{3} \geq y + -\mathsf{42}$ which are constraints. Term $\mathsf{5} + \mathsf{9}$ is also a (ground) theory term, but not a constraint. Term $\mathsf{pow}(\mathsf{2}, y)$ is not a theory term. To implement an LCTRS calculating the *factorial* function over $\mathbb{Z}$, we use the signature Σ above and the LCTRS $\mathcal{R}_1 = \{\ \mathsf{fact}(x) \to \mathsf{subfact}(x, \mathsf{1}),\ \mathsf{subfact}(x,y) \to y\ [x \leq \mathsf{0}],\ \mathsf{subfact}(x,y) \to \mathsf{subfact}(x',y')\ [\neg(x \leq \mathsf{0}) \wedge x' = x - \mathsf{1} \wedge y' = x \times y]\ \}$. The term $\mathsf{fact}(\mathsf{3})$ is reduced by $\mathcal{R}_1$ to $\mathsf{6}$: $\mathsf{fact}(\mathsf{3}) \to_{\mathcal{R}_1} \mathsf{subfact}(\mathsf{3},\mathsf{1}) \to_{\mathcal{R}_1} \mathsf{subfact}(\mathsf{2},\mathsf{3}) \to_{\mathcal{R}_1} \mathsf{subfact}(\mathsf{1},\mathsf{6}) \to_{\mathcal{R}_1} \mathsf{subfact}(\mathsf{0},\mathsf{6}) \to_{\mathcal{R}_1} \mathsf{6}$.

3 Syntax and Semantics of SIMP$^+$

In this section, we recall the syntax and semantics of SIMP$^+$. We follow the syntax and semantics of SIMP$^+$ [5] which is a naive extension of a small imperative language SIMP [3], to global variables and function calls. Note that SIMP$^+$ can be considered as a restricted variant of SIMPLE, a non-trivial imperative language, in [11].

Figure 1 defines the syntax of SIMP$^+$ in BNF. For a nonterminal N, N^* denotes an arbitrary sequence of N: ε, N, $N\ N$, $\ldots$, and $N^*_\diamond$ with a separation symbol $\diamond$ denotes an arbitrary $\diamond$-separated sequence of N: ε, N, $N \diamond N$, $N \diamond N \diamond N$, $\ldots$ We often omit brackets in the usual way. SIMP$^+$ programs generated from nonterminal *Prgrm* are sequences of declarations of global variables and functions. We assume that programs are well-formed: For a program P, any variable in P are declared properly; a function identifier f has a fixed arity, and the definition and call of f are consistent with the arity; each function f is defined exactly once in P and any function called in P is defined in P. We also assume that function main is declared in P as a nullary function. To simplify the semantics, we assume that local variables in function declarations are different from global variables and parameters of functions. We consider a local-variable declaration `int` x `=` v`;` as a statement.

Only assignment statements of the form x `=` f`(...);` are allowed to call functions—no function is called in any expression—because we have to push a frame to a call stack in calling a function, and need a special treatment for such a process. This is not a restriction because e.g., any function call can be replaced by a fresh variable that is assigned the result of the function call. For brevity, x is assumed to be a local variable. In addition, for `return` e`;`, we restrict e to a local variable x.

$$
\begin{array}{rcl}
\mathit{Prgrm} & ::= & \mathit{VDecl}^* \ \mathit{FDecl}^* \\
\mathit{VDecl} & ::= & \texttt{int}\ x\ \texttt{=}\ v\texttt{;} \\
\mathit{FDecl} & ::= & \texttt{int}\ \ f\texttt{(}\mathit{Param}^*_{,}\texttt{)}\ \texttt{\{}\mathit{VDecl}^*\ \mathit{Stmt}^*\texttt{\}} \\
\mathit{Param} & ::= & \texttt{int}\ x \\
A & ::= & v \mid x \mid \texttt{(}A\ \texttt{+}\ A\texttt{)} \mid \texttt{(}A\ \texttt{-}\ A\texttt{)} \\
B & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{(}A\ \mathit{Cmp}\ A\texttt{)} \mid \texttt{(!}\,B\texttt{)} \mid \texttt{(}B\ \texttt{\&\&}\ B\texttt{)} \mid \texttt{(}B\ \texttt{||}\ B\texttt{)} \\
\mathit{Cmp} & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\mathit{Stmt} & ::= & \texttt{\{}\mathit{Stmt}^*\texttt{\}} \mid x\ \texttt{=}\ A\texttt{;} \mid x\ \texttt{=}\ f\texttt{(}A^*_{,}\texttt{);} \mid \texttt{if}\ \texttt{(}B\texttt{)}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt} \mid \texttt{while}\ \texttt{(}B\texttt{)}\ \mathit{Stmt} \mid \texttt{return}\ \ x\texttt{;}
\end{array}
$$

where x is a variable identifier, f is a function identifiers, and v is an integer.

Figure 1: the syntax of SIMP$^+$.

We consider integer and Boolean expressions of SIMP$^+$ as the corresponding theory terms over the standard integer signature Σ_{int} of LCTRSs by implicitly replacing e.g., `<=` by $\leq$, and vice versa. This means that we do not distinguish expressions and theory terms, abusing them in both settings implicitly. For this reason, to define the small-step semantics of SIMP$^+$, we use the interpretation $[\![\cdot]\!]$ of ground theory terms to evaluate expressions under assignments.

We assume w.l.o.g. that SIMP$^+$ programs are of the following form:

$$\texttt{int}\ gv_1\ \texttt{=}\ n_1\texttt{;}\ \ldots\ \texttt{int}\ gv_k\ \texttt{=}\ n_k\texttt{;}\ \ \texttt{int}\ \ f_1\texttt{(}\ldots\texttt{)}\ \texttt{\{}\ldots\texttt{\}}\ \ldots\ \texttt{int}\ \ f_{k'}\texttt{(}\ldots\texttt{)}\ \texttt{\{}\ldots\texttt{\}} \tag{1}$$

where $gv_1,\ldots,gv_k,f_1,\ldots,f_{k'}$ are distinct identifiers and one of $f_1,\ldots,f_{k'}$ is main. In the following, $\overrightarrow{gv}$ denotes the sequence $gv_1,\ldots,gv_k$. In the rest of the paper, we use P as a SIMP$^+$ program of the form (1) without notice.

Example 3.1 Program 1 shows a SIMP$^+$ program P_1 which defines three kinds of summation functions.

An *assignment* is a partial mapping from variable identifiers to integers. We consider variable identifiers in programs as variables in LCTRSs. Thus, assignments can be considered substitutions whose range is restricted to value constants of integers. We abuse assignments as substitutions for terms in the setting of LCTRSs. The empty mapping—the mapping whose domain is empty—is denoted by $\varnothing$.

The *update* $\sigma[x \mapsto n]$ of an assignment σ w.r.t. x for an integer v is defined as follows: If $x = y$ then $\sigma[x \mapsto v](y) = v$, and otherwise, $\sigma[x \mapsto v](y) = \sigma(y)$. Given pairwise distinct variables $x_1,\ldots,x_n$, we abbreviate $(\sigma[x_1 \mapsto v_1])\ldots[x_n \mapsto v_n]$ to $\sigma[x_1 \mapsto v_1,\ldots,x_n \mapsto v_n]$. Let θ be $\{x_1 \mapsto v_1,\ldots,x_n \mapsto v_n\}$ for the update $[x_1 \mapsto v_1,\ldots,x_n \mapsto v_n]$. Then, $\sigma[x_1 \mapsto v_1,\ldots,x_n \mapsto v_n] = \theta\sigma$.

To simplify the representation of a configuration and the corresponding function symbol which is introduced for the location indicated by the configuration, we assume that each statement in a program has a unique label, and use the label to indicate the statement. For a configuration, to indicate a statement to be executed, it is usual to use the statement itself as a component in the configuration, but the structure of the statement is not necessary. In transforming a SIMP$^+$ program into an LCTRS, for each location the configuration indicates, we prepare a function symbol. In this paper, we use line numbers as such labels. Note that each occurrence of a statement is given distinct labels. As an exception, we use the function identifier f as a label of the head statement of the body of f: For a function declaration `int f(...) {`*stmts*`}`, the label of the head statement of *stmts* is f. For a statement *stmt* with a label ρ, $\mathit{Stmt}(\rho)$ denotes *stmt*. For a label ρ, $\mathit{Fun}(\rho)$ denotes the function identifier, the body of which includes

Program 1: a SIMP$^+$ program P_1 defining several summation functions with a global variable.

```
1  int n = 0;
2
3  int sum1(int x1) {
4    int i1 = 0;
5    int z1 = 0;
6    n = n + 1;
7    while ( i1 < x1 )
8    {
9      z1 = z1 + i1 + 1;
10     i1 = i1 + 1;
11   }
12   return z1;
13 }
14
15 int sum2(int x2) {
16   int z2 = 0;
17   n = n + 1;
18   z2 = x2 * (x2 + 1) / 2;
19   return z2;
20 }
21 int sum3(int x3) {
22   int z3 = 0;
23   n = n + 1;
24   if ( x3 <= 0 )
25     z3 = 0;
26   else
27   {
28     z3 = sum3(x3 - 1);
29     z3 = x3 + z3;
30   }
31   return z3;
32 }
33
34 int main() {
35   int ret = 0;
36   int z = 3;
37   z = sum1(z);
38   return ret;
39 }
40
```

the statement of ρ. Note that if $Fun(\rho) = \rho$, then ρ is some function identifier f which is the label of the head statement of the body of f. The set of labels in P is denoted by $Lab(P)$.

Example 3.2 For P_1 in Program 1, $Lab(P_1) = \{\mathsf{sum1}, 5, 6, 7, 8, 9, 10, 12, \mathsf{sum2}, 17, 18, 19, \mathsf{sum3}, 23, 24, 25, 27, 28, 29, 31, \mathsf{main}, 36, 37, 38\}$, $Fun(\mathsf{sum1}) = Fun(5) = \mathsf{sum1}$, and $Stmt(\mathsf{sum1}) = (\texttt{int}\ i_1\ \texttt{= 0;})$.

For each statement *stmt* other than `return` statements, we can statically determine which statement follows after *stmt*: For a list *stmt stmt′ stmts* of statements, *stmt′* follows after *stmt*; for a block statement `{`… *stmt*`}` *stmt′ stmts*, *stmt′* follows after both `{`… *stmt*`}` and *stmt*; for an `if` statement `if`(…) $stmt_1$ `else` $stmt_2$ *stmt′ stmts*, *stmt′* follows after both $stmt_1$ and $stmt_2$; for a `while` statement `while`(…) *stmt stmt′ stmts*, *stmt′* follows after *stmt*. For a statement *stmt* with a label ρ, $Nxt(\rho)$ denotes the statement that follows after *stmt*.

Example 3.3 For P_1 in Program 1, $Nxt(\mathsf{sum1}) = 4$, $Nxt(4) = 5$, $Nxt(7) = 12$, $Nxt(8) = 7$, $Nxt(10) = 7$, and $Nxt(24) = Nxt(25) = Nxt(29) = 31$.

For each statement *stmt* with a label ρ, we can statically determine which local variables are declared, i.e., accessible in executing *stmt*. We denote the set of such variables by $dvars(\rho)$. In addition, we assume a fixed arbitrary order of local variables, and $\overrightarrow{dvars}(\rho)$ denotes the sequence of the variables in $dvars(\rho)$ under such a fixed order.

Example 3.4 For P_1 in Program 1, $dvars(4) = \{x_1\}$, $dvars(5) = \{x_1, , i_1\}$, and $dvars(7) = dvars(8) = \cdots = dvars(12) = \{x_1, i_1, z_1\}$. In addition, $\overrightarrow{dvars}(7) = \ x_1, i_1, z_1$.

Frames for function calls are tuples of the form (ρ, σ), where ρ is the label of a statement and σ is an assignment for local variables such that $\mathcal{D}om(\sigma) = dvars(\rho)$. *Call stacks* for SIMP$^+$ programs are lists of frames, and we use a list constructor :: and [] for such lists. *Configurations* of SIMP$^+$ programs are of the form $\langle cstck, \sigma_0\rangle$ such that *cstck* is a call stack and σ_0 is an assignment for the global variables.

The initial configuration of P is $\langle[(\mathsf{main},\varnothing)], \{gv_i \mapsto n_i \mid 1 \leq i \leq k\}\rangle$.

Example 3.5 The initial configuration of P_1 in Program 1 is $\langle[(\mathsf{main},\varnothing)], \{n \mapsto 0\}\rangle$.

Following [11], we define the small-step semantics for SIMP$^+$. To make all inference rules for the small-step semantics have a common form, we avoid the occurrence of $\bot$ in any inference rules, using $[\![\,(\neg\varphi)(\sigma \cup \sigma_0)\,]\!] = \top$ instead of $[\![\,\varphi(\sigma\cup\sigma_0)\,]\!] = \bot$. The inference rules for the small-step semantics of SIMP$^+$, which define the transition step $\rightharpoonup_P$ over configurations of P, are illustrated in Figure 2. In the rule (**while**$_\top$), ρ does not appear in the resulting configuration $\langle(\rho', \sigma) :: cstck, \sigma_0\rangle$. One may think that the body *stmt* is executed at most once during the iteration. In fact, the body *stmt* is executed as much as needed: Let ρ' be one of the statements executed at the last step for *stmt*; then we have $Nxt(\rho') = \rho$; In executing ρ', the resulting configuration is of the form $\langle(\rho, \sigma') :: cstck, \sigma'_0\rangle$.

Some symbols in Figure 2—ρ, σ, σ_0, *cstck*, and so on—are meta-variables to represent inference rules, and thus, inference rules in Figure 2 can be considered schema defining the transition of configurations. For this reason, we consider *partial configurations*, configurations that may include variables (see the appendix). Call stack and frame included in partial configurations are called *partial* ones (see also the appendix). Partial configurations may include variables for assignments, and thus, may include updates of the form $\sigma[x_1 \mapsto v_1, \ldots, x_m \mapsto v_m]$ such that σ is a variable. To distinguish configurations without variables or updates from partial ones, we call such a configuration a *full configuration*. Note that a full configuration is a partial one. Updates appear only in partial configurations that are not full ones, and for any update of the form $\sigma[x_1 \mapsto v_1, \ldots, x_m \mapsto v_m]$, we consider $v_1, \ldots, v_m$ variables.

In transitioning a full configuration, we use inference rules by instantiating meta-variables w.r.t. the configuration. A label ρ in P instantiates the corresponding inference rule. For example, for rule (**loc**), symbols $x, v, Nxt(\rho)$ are determined by ρ. Further instantiating such an intermediately instantiated rule by $\sigma, \sigma_0, cstck$, we transition a full configuration. Viewed in this light, we formulate intermediately instantiated inference rules. For an inference rule (**rule**) and a label ρ in P, (**rule**)$[\rho]$ denotes the intermediately instantiated rule such that $\sigma, \sigma', \sigma_0, cstck$ are considered variables. In addition, the transitions defined by the rule (**rule**)$[\rho]$ is denoted by (**rule**)$[\rho](\sigma, \sigma_0, cstck)$, e.g., (**loc**)$[\rho](\sigma, \sigma_0, cstck)$ is the transition $\langle(\rho, \sigma) :: cstck, \sigma_0\rangle \rightharpoonup_P \langle(Nxt(\rho), \{x \mapsto v\}\sigma) :: cstck, \sigma_0\rangle$, where $\sigma, \sigma_0, cstck$ are assignments and a call stack, respectively. We define the set *Rules*(P) of the intermediately instantiated inference rules for P as follows:

$$
\begin{aligned}
Rules(P) = {} & \{(\mathbf{loc})[\rho] \mid \rho \in Lab(P),\ Stmt(\rho) = (\mathtt{int}\ x\ \mathtt{=}\ v\mathtt{;})\} \\
& \cup \{(\mathbf{block})[\rho] \mid \rho \in Lab(P),\ Stmt(\rho) = (\mathtt{\{}stmts\mathtt{\}})\} \\
& \cup \{(\mathbf{g\text{-}assign})[\rho] \mid \rho \in Lab(P),\ Stmt(\rho) = (gv_i\ \mathtt{=}\ e\mathtt{;})\} \\
& \cup \{(\mathbf{l\text{-}assign})[\rho] \mid \rho \in Lab(P),\ Stmt(\rho) = (x\ \mathtt{=}\ e\mathtt{;})\} \\
& \cup \{(\mathbf{call})[\rho], (\mathbf{return})[\rho] \mid \rho \in Lab(P),\ Stmt(\rho) = (x\ \mathtt{=}\ g\mathtt{(}\ldots\mathtt{)}\mathtt{;})\} \\
& \cup \{(\mathbf{if}_\top)[\rho], (\mathbf{if}_\bot)[\rho] \mid \rho \in Lab(P),\ Stmt(\rho) = (\mathtt{if}\ \mathtt{(}\varphi\mathtt{)}\ stmt_1\ \mathtt{else}\ stmt_2)\} \\
& \cup \{(\mathbf{while}_\top)[\rho], (\mathbf{while}_\bot)[\rho] \mid \rho \in Lab(P),\ Stmt(\rho) = (\mathtt{while}\ \mathtt{(}\varphi\mathtt{)}\ stmt)\}
\end{aligned}
$$

Example 3.6 For P_1 in Program 1, $Rules(P_1) = \{\,(\mathbf{loc})[\mathsf{sum1}], (\mathbf{loc})[5], (\mathbf{g\text{-}assign})[6], (\mathbf{while}_\top)[y], (\mathbf{while}_\bot)[y], \ldots, (\mathbf{call})[37], (\mathbf{return})[37]\,\}$.

Rules(P) can be considered an LCTRS that uses assignments as values. The class of constraints of such an LCTRS is more complex than that for LCTRSs generated in the previous work [5]. For this reason, we transform SIMP$^+$ programs into LCTRSs over the standard integer signature Σ_{int}.

$$\frac{[\![v_0(\sigma \cup \sigma_0)]\!] = v}{\langle (\rho,\sigma) :: cstck, \sigma_0 \rangle \rightharpoonup \langle (Nxt(\rho), \sigma[x \mapsto v]) :: cstck, \sigma_0 \rangle}\ \textbf{(loc)} \quad \text{where } Stmt(\rho) = (\texttt{int}\ x\ \texttt{=}\ v_0\texttt{;})$$

$$\frac{}{\langle (\rho,\sigma) :: cstck, \sigma_0 \rangle \rightharpoonup \langle (Nxt(\rho), \sigma) :: cstck, \sigma_0 \rangle}\ \textbf{(block)}$$

where $Stmt(\rho) = (\texttt{\{}stmt\ stmts\texttt{\}})$ and ρ' is the label of *stmt*

$$\frac{[\![e(\sigma \cup \sigma_0)]\!] = v}{\langle (\rho,\sigma) :: cstck, \sigma_0 \rangle \rightharpoonup \langle (Nxt(\rho), \sigma) :: cstck, \sigma_0[gv_i \mapsto v] \rangle}\ \textbf{(g-assign)} \quad \text{where } Stmt(\rho) = (gv_i\ \texttt{=}\ e\texttt{;})$$

$$\frac{[\![e(\sigma \cup \sigma_0)]\!] = v}{\langle (\rho,\sigma) :: cstck, \sigma_0 \rangle \rightharpoonup \langle (Nxt(\rho), \sigma[x \mapsto v]) :: cstck, \sigma_0 \rangle}\ \textbf{(l-assign)} \quad \text{where } Stmt(\rho) = (x\ \texttt{=}\ e\texttt{;})$$

$$\frac{[\![e_1(\sigma \cup \sigma_0)]\!] = v_1 \qquad \dots \qquad [\![e_{n'}(\sigma \cup \sigma_0)]\!] = v_{n'}}{\langle (\rho,\sigma) :: cstck, \sigma_0 \rangle \rightharpoonup \langle (g, \varnothing[y_1 \mapsto v_1, \dots, y_{n'} \mapsto v_{n'}]) :: (\rho,\sigma) :: cstck, \sigma_0 \rangle}\ \textbf{(call)}$$

where $Stmt(\rho) = (x\ \texttt{=}\ g\texttt{(}e_1\texttt{,}\dots\texttt{,}e_{n'}\texttt{);})$ and g is declared in P as $\texttt{int}\ \ g\texttt{(int}\ y_1\texttt{,}\dots\texttt{,int}\ y_{n'}\texttt{)}\ \texttt{\{}\dots\texttt{\}}$,

$$\frac{[\![y(\sigma' \cup \sigma_0)]\!] = v}{\langle (\rho',\sigma') :: (\rho,\sigma) :: cstck, \sigma_0 \rangle \rightharpoonup \langle (Nxt(\rho), \sigma[x \mapsto v]) :: cstck, \sigma_0 \rangle}\ \textbf{(return)}$$

where $Stmt(\rho') = (\texttt{return}\ y\texttt{;})$ and $Stmt(\rho) = (x\ \texttt{=}\ g\texttt{(}\dots\texttt{);})$,

$$\frac{[\![\varphi(\sigma \cup \sigma_0)]\!] = \top}{\langle (\rho,\sigma) :: cstck, \sigma_0 \rangle \rightharpoonup \langle (\rho_1, \sigma) :: cstck, \sigma_0 \rangle}\ \textbf{(if}_\top\textbf{)} \qquad \frac{[\![(\neg\varphi)(\sigma \cup \sigma_0)]\!] = \top}{\langle (\rho,\sigma) :: cstck, \sigma_0 \rangle \rightharpoonup \langle (\rho_2, \sigma) :: cstck, \sigma_0 \rangle}\ \textbf{(if}_\bot\textbf{)}$$

where $Stmt(\rho) = (\texttt{if (}\varphi\texttt{)}\ stmt_1\ \texttt{else}\ stmt_2)$ and ρ_i is the label of $stmt_i$

$$\frac{[\![\varphi(\sigma \cup \sigma_0)]\!] = \top}{\langle (\rho,\sigma) :: cstck, \sigma_0 \rangle \rightharpoonup \langle (\rho', \sigma) :: cstck, \sigma_0 \rangle}\ \textbf{(while}_\top\textbf{)}$$

where $Stmt(\rho) = (\texttt{while (}\varphi\texttt{)}\ stmt)$ and ρ' is the label of *stmt*

$$\frac{[\![(\neg\varphi)(\sigma \cup \sigma_0)]\!] = \top}{\langle (\rho,\sigma) :: cstck, \sigma_0 \rangle \rightharpoonup \langle (Nxt(\rho), \sigma) :: cstck, \sigma_0 \rangle}\ \textbf{(while}_\bot\textbf{)} \quad \text{where } Stmt(\rho) = (\texttt{while (}\varphi\texttt{)}\ stmt)$$

Figure 2: Inference rules for the small-step semantics of SIMP$^+$.

Implicitly adding the redundant premise $[\![\texttt{true}(\sigma \cup \sigma_0)]\!] = \top$ to **(loc)**, **(block)**, **(g-assign)**, **(l-assign)**, **(call)**, and **(return)**, all rules in *Rules*(P)—the inference rules in Figure 2— are of the following form:

$$\frac{[\![\varphi(\sigma \cup \sigma_0)]\!] = \top \qquad [\![e_1(\sigma \cup \sigma_0)]\!] = v_1 \qquad \dots \qquad [\![e_m(\sigma \cup \sigma_0)]\!] = v_m}{cnfg \rightharpoonup cnfg'} \tag{2}$$

where *cnfg* is a partial configuration without any update, and *cnfg*$'$ is a partial configuration that may include an update.

4 Transforming SIMP$^+$ Programs into LCTRSs via Injective Functions

In this section, using an injective function, we first transform an intermediately instantiated rule in *Rules*(P) into a constrained rewrite rule, generating an LCTRS $\mathfrak{T}(P)$. Then, further using the injective function, we show a correctness proof of the transformation.

We introduce the following function symbols to Σ_{term}:

- a $(k+1)$-ary function symbol $\mathsf{cnfg} : \mathit{cstack} \times \mathit{int} \times \cdots \times \mathit{int} \to \mathit{config}$ for configurations,
- a binary constructor $\mathsf{stack} : \mathit{frame} \times \mathit{cstack} \to \mathit{cstack}$ and a constant $\mathsf{empty} : \mathit{cstack}$ for call stacks,
- an n-ary constructor $f : \mathit{int} \times \cdots \times \mathit{int} \to \mathit{frame}$ for an n-ary function identifier f, and
- an n-ary constructor $f_\rho : \mathit{int} \times \cdots \times \mathit{int} \to \mathit{frame}$ for a frame $(\rho, \ldots)$ such that $n = |\mathit{dvars}(\rho)|$, $f = \mathit{Fun}(\rho)$, and $\rho \neq f$.

In the following, f_ρ denotes f if $\mathit{Fun}(\rho) = \rho$ (i.e., $\rho = f$).[1]

4.1 Injective Functions from Configurations to Terms

An idea to simplify a formulation of the transformation is the use of an injective function ξ from configurations to terms so that

- we transform an intermediately instantiated inference rule of the form (2) into a constrained rewrite rule $\xi(\mathit{cnfg}) \to \xi(\mathit{cnfg}')\ [\varphi \wedge v_1 = e_1 \wedge \cdots \wedge v_{n'} = e_{n'}]$, and
- for the correctness of the transformation, we show that for full configurations $\mathit{cnfg}, \mathit{cnfg}'$ and a ground term $t : \mathit{config}$,
  - if $\mathit{cnfg} \rightharpoonup_P \mathit{cnfg}'$, then $\xi(\mathit{cnfg}) \to_{\mathfrak{T}(P)} \xi(\mathit{cnfg}')$, and
  - if $\xi(\mathit{cnfg}) \to_{\mathfrak{T}(P)} t$, then $\mathit{cnfg} \rightharpoonup_P \xi^{-1}(t)$.

To define ξ for partial configurations, we prepare two injective functions δ_{config} and ζ_{config} such that $\xi = \zeta_{\mathrm{config}}^{-1} \circ \delta_{\mathrm{config}}$; δ_{config} maps partial configurations to pairs of linear terms and substitutions; ζ_{config} maps terms with sort *config* to such pairs.

In applying δ_{config} to a full configuration *cnfg*, a frame (ρ, σ) in the configuration is intermediately transformed into $(f_\rho(\overrightarrow{\mathit{dvars}}(\rho)), \sigma)$, where $\mathit{Fun}(\rho) = f$. When f is recursively called in *cnfg*, there may exist two frames (ρ_1, σ_1) and (ρ_2, σ_2) such that $\mathit{Fun}(\rho_1) = \mathit{Fun}(\rho_2) = f$ and $\{y_1, \ldots, y_{n'}\} \subseteq \mathcal{D}om(\sigma_1) \cap \mathcal{D}om(\sigma_2)$, where f is declared in P as `int` f`(int` y_1`,` $\ldots$`,int` $y_{n'}$`) {` $\ldots$ `}`. If we transform (ρ_1, σ_1) and (ρ_2, σ_2) into $(f_{\rho_1}(\overrightarrow{\mathit{dvars}}(\rho_1)), \sigma_1)$ and $(f_{\rho_2}(\overrightarrow{\mathit{dvars}}(\rho_2)), \sigma_2)$, respectively, then the resulting term of $\delta_{\mathrm{config}}(\mathit{cnfg})$ contains both $f_{\rho_1}(\overrightarrow{\mathit{dvars}}(\rho_1))$ and $f_{\rho_2}(\overrightarrow{\mathit{dvars}}(\rho_2))$ as subterms, and thus, the resulting term cannot be linear. In addition, we cannot combine σ_1 and σ_2 as a substitution.

To make the resulting term linear, we rename variables in $\mathit{dvars}(\rho)$ by means of the location of frames in call stacks. Given a call stack $(\rho_n, \theta_n) :: (\rho_{n-1}, \theta_{n-1}) :: \cdots :: (\rho_0, \theta_0) :: []$ and a partial call stack $(\rho_n, \theta_n) :: (\rho_{n-1}, \theta_{n-1}) :: \cdots :: (\rho_0, \theta_0) :: s$ with s a variable, the *height* of the frame (ρ_h, θ_h) with $0 \le h \le n$ is h. For a frame (ρ, σ) with height $h \geq 0$, a variable x in $\mathit{dvars}(\rho)$ is renamed to x^h; if $h = 0$, then we abbreviate x^h to x; $\mathit{dvars}^h(\rho)$ denotes the set of the renamed variables for $\mathit{dvars}(\rho)$, and $\overrightarrow{\mathit{dvars}}^h(\rho)$ denotes the sequence of the renamed variables for $\overrightarrow{\mathit{dvars}}(\rho)$; σ^h denotes the substitution obtained from σ by renaming the domain via h, i.e., $\sigma^h = \{x^h \mapsto x\sigma \mid x \in \mathit{dvars}(\rho)\}$.

Now we define injective functions ξ, δ_{config}, and ζ_{config}.

Definition 4.1 **(ξ, δ_{config})** *The mapping $\delta_{\mathrm{frame},h}$ from partial frames into pairs of linear terms and substitutions is defined as follows:*

[1] In this case, ρ is the first statement of the body of the definition for f, and we use f instead of f_ρ because f is more natural as the entry point of f than f_ρ.

- $\delta_{\mathrm{frame},h}((\rho,\sigma)) = (f_\rho(\overrightarrow{dvars}^h(\rho)), \sigma^h)$, *where* σ *is an assignment with* $\mathcal{D}om(\sigma) = dvars(\rho)$,
- $\delta_{\mathrm{frame},h}((\rho,\sigma)) = (f_\rho(\overrightarrow{dvars}^h(\rho)), \varnothing)$, *where* σ *is a variable,*
- $\delta_{\mathrm{frame},h}((\rho,\sigma[y \mapsto v])) = (f_\rho(\overrightarrow{dvars}^h(\rho)), \{y^h \mapsto v\})$, *where* $Fun(\rho) \neq \rho$, *both* v,σ *are variables, and* $y \in dvars(\rho)$, *and*
- $\delta_{\mathrm{frame},h}((\rho,\varnothing[y_1 \mapsto v_1,\ldots,y_m \mapsto v_m])) = (f_\rho(\overrightarrow{dvars}^h(\rho)), \{y_1^h \mapsto v_1,\ldots,y_m^h \mapsto v_m\})$, *where* $Fun(\rho) = \rho$, *all* $v_1,\ldots,v_m,\sigma$ *are variables, and* $dvars(\rho) = \{y_1,\ldots,y_m\}$.

The mapping δ_{cstack} *from partial call stacks is inductively defined as follows:*

- $\delta_{\mathrm{cstack}}(s) = (s,\varnothing)$, *where s is a variable,*
- $\delta_{\mathrm{cstack}}([\,]) = (\mathsf{empty},\varnothing)$, *and*
- $\delta_{\mathrm{cstack}}(frm :: cstck) = (\mathsf{stack}(t,s), \theta \cup \theta')$,[2] *where h is the length of cstck and* $\delta_{\mathrm{frame},h}(frm) = (t,\theta)$ *and* $\delta_{\mathrm{cstack}}(cstck) = (s,\theta')$.

The mapping δ_{config} *from partial configurations to pairs of linear terms and substitutions is defined as follows:*

- $\delta_{\mathrm{config}}(\langle cstck,\sigma_0\rangle) = (\mathsf{cnfg}(s,\overrightarrow{gv}), \theta\cup\sigma_0)$,[3] *where* σ_0 *is an assignment with* $\mathcal{D}om(\sigma_0) = \{\overrightarrow{gv}\}$,
- $\delta_{\mathrm{config}}(\langle cstck,\sigma_0\rangle) = (\mathsf{cnfg}(s,\overrightarrow{gv}), \theta)$, *where* σ_0 *is a variable, and*
- $\delta_{\mathrm{config}}(\langle cstck,\sigma_0[gv_i \mapsto v_i]\rangle) = (\mathsf{cnfg}(s,\overrightarrow{gv}), \theta\cup\{gv_i \mapsto v_i\})$,[4] *where* v_i,σ_0 *are variables,*

where $\delta_{\mathrm{cstack}}(cstck) = (s,\theta)$. *The mapping* ξ *from partial configurations into terms is defined as* $\xi(cnfg) = u\sigma$, *where* $\delta_{\mathrm{config}}(cnfg) = (u,\sigma)$.

By definition, it is clear that all δ_{config}, δ_{cstack}, and $\delta_{\mathrm{frame},h}$ return pairs of linear terms and substitutions.

Proposition 4.2 *All* δ_{config}, δ_{cstack}, *and* $\delta_{\mathrm{frame},h}$ *are injective.*

The injectivity of ξ is not so trivial. For this reason, we define a mapping from terms with sort *config* to pairs of linear terms and substitutions.

Definition 4.3 (ζ_{config}) *The mapping* $\zeta_{\mathrm{frame},h}$ *from terms to pairs of linear terms and substitutions is defined as follows:*

- $\zeta_{\mathrm{frame},h}(f_\rho(v_1,\ldots,v_m)) = (f_\rho(y_1^h,\ldots,y_m^h), \{y_1^h \mapsto v_1, \ldots, y_m^h \mapsto v_m\})$, *where* $v_1,\ldots,v_m$ *are integers and* $\overrightarrow{dvars}(\rho) = y_1,\ldots,y_m$,
- $\zeta_{\mathrm{frame},h}(f_\rho(\overrightarrow{dvars}^h(\rho))) = (f_\rho(\overrightarrow{dvars}^h(\rho)), \varnothing)$,
- $\zeta_{\mathrm{frame},h}(f_\rho(y_1^h,\ldots,y_{i-1}^h,v,y_{i+1}^h,\ldots,y_m^h)) = (f_\rho(\overrightarrow{dvars}^h(\rho)), \{y_i^h \mapsto v,\})$, *where v is a variable and* $\overrightarrow{dvars}(\rho) = y_1,\ldots,y_m$, *and*
- $\zeta_{\mathrm{frame},h}(f_\rho(v_1,\ldots,v_m)) = (f_\rho(y_1^h,\ldots,y_m^h), \{y_1^h \mapsto v_1, \ldots, y_m^h \mapsto v_m\})$, *where* $v_1,\ldots,v_m$ *are pairwise distinct variables and* $\overrightarrow{dvars}(\rho) = y_1,\ldots,y_m$.

The mapping ζ_{cstack} *from terms to pairs of linear terms and substitutions is inductively defined as follows:*

- $\zeta_{\mathrm{cstack}}(s) = (s,\varnothing)$, *where s is a variable,*

[2] Note that $\theta\cup\theta'$ is a substitution because $\mathcal{D}om(\theta)\cap\mathcal{D}om(\theta') = \emptyset$.
[3] Note that $\theta\cup\sigma_0$ is a substitution because $\mathcal{D}om(\theta)\cap\mathcal{D}om(\sigma_0) = \emptyset$.
[4] Note that $\theta\cup\{gv_i \mapsto v\}$ is a substitution because $gv_i \notin \mathcal{D}om(\theta)$.

- $\zeta_{\rm cstack}(\mathsf{empty}) = (\mathsf{empty}, \varnothing)$, *and*
- $\zeta_{\rm cstack}(\mathsf{stack}(\mathit{frm}, s)) = (\mathsf{stack}(t, s'), \theta \cup \theta')$, *where h is the number of occurrences of* stack *in s, $\zeta_{{\rm frame},h}(\mathit{frm}) = (t, \theta)$, and $\zeta_{\rm cstack}(s) = (s', \theta')$.*

The mapping $\zeta_{\rm config}$ from terms with sort config to pairs of linear terms and substitutions is defined as follows:

- $\zeta_{\rm config}(\mathsf{cnfg}(s, v_1, \ldots, v_k)) = (\mathsf{cnfg}(s', \overrightarrow{gv}), \theta \cup \{gv_1 \mapsto v_1, \ldots, gv_k \mapsto v_k\})$, *where $v_1, \ldots, v_k$ are integers,*
- $\zeta_{\rm config}(\mathsf{cnfg}(s, \overrightarrow{gv})) = (\mathsf{cnfg}(s', \overrightarrow{gv}), \theta)$, *and*
- $\zeta_{\rm config}(\mathsf{cnfg}(s, gv_1, \ldots, gv_{i-1}, v, gv_{i+1}, \ldots, gv_k)) = (\mathsf{cnfg}(s', \overrightarrow{gv}), \theta \cup \{gv_i \mapsto v\})$, *where v is a variable,*

where $\zeta_{\rm cstack}(s) = (s', \theta)$.

Proposition 4.4 *All $\zeta_{\rm config}, \zeta_{\rm cstack}, \zeta_{{\rm frame},h}$ are injective.*

The proof of Proposition 4.4 is analogous to Proposition 4.2. The following proposition is a direct consequence of Propositions 4.2 and 4.4.

Proposition 4.5 $\xi = \zeta_{\rm config}^{-1} \circ \delta_{\rm config}$ *and ξ is injective.*

4.2 Formulating a Transformation of SIMP$^+$ programs into LCTRSs via ξ

In this section, using ξ, we formulate a transformation of SIMP$^+$ programs into LCTRSs.

Definition 4.6 (transformation $\mathfrak{T}$) *For a rule $(\mathbf{rule})[\rho]$ of the form*

$$\frac{[\![\varphi(\sigma \cup \sigma_0)]\!] = \top \qquad [\![e_1(\sigma \cup \sigma_0)]\!] = v_1 \qquad \ldots \qquad [\![e_m(\sigma \cup \sigma_0)]\!] = v_m}{\mathit{cnfg} \rightharpoonup \mathit{cnfg}'}$$

in $\mathit{Rules}(P)$, $\mathfrak{T}_{\rm rule}((\mathbf{rule})[\rho]) = \xi(\mathit{cnfg}) \to \xi(\mathit{cnfg}')\ [\varphi \wedge v_1 = e_1 \wedge \cdots \wedge v_{n'} = e_{n'}]$. For readability, we use h instead of 1 *in renaming variables.*[5] *A transformation $\mathfrak{T}$ that takes a SIMP$^+$ program as input and returns an LCTRS is defined as $\mathfrak{T}(P) = \{\ \mathfrak{T}_{\rm rule}((\mathbf{rule})[\rho]) \mid (\mathbf{rule})[\rho] \in \mathit{Rules}(P)\ \}$.*

Example 4.7 P_1 in Program 1 is transformed into the LCTRS in Figure 3 (see the appendix for detail).

4.3 A Correctness Proof for $\mathfrak{T}$ via ξ

Finally, using ξ, we show a correctness proof for $\mathfrak{T}$.

Lemma 4.8 (soundness of $\to_{\mathfrak{T}(P)}$) *For full configurations $\mathit{cnfg}_1, \mathit{cnfg}_2$ of P, if $\mathit{cnfg}_1 \rightharpoonup_P \mathit{cnfg}_2$, then $\xi(\mathit{cnfg}_1) \to_{\mathfrak{T}(P)} \xi(\mathit{cnfg}_2)$.*

Lemma 4.9 (completeness of $\to_{\mathfrak{T}(P)}$) *For a full configuration cnfg_1 of P and a ground term t with sort config, if $\xi(\mathit{cnfg}_1) \to_{\mathfrak{T}(P)} t$, then $\mathit{cnfg}_1 \rightharpoonup_P \xi^{-1}(t)$.*

Thanks to the injectivity of ξ, the proof of Lemma 4.9 is analogous to Lemma 4.8.

[5] The maximum height of partial call stacks in rules is two and we need one symbol for superscripts to rename variables.

$$
\left\{
\begin{array}{rll}
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}(x_1),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_5(x_1,v),s),n) & [& v=0 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_5(x_1,i_1),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_6(x_1,i_1,v),s),n) & [& v=0 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_6(x_1,i_1,z_1),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_7(x_1,i_1,z_1),s),v) & [& v=n+1 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_7(x_1,i_1,z_1),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_8(x_1,i_1,z_1),s),n) & [& i_1<x_1 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_7(x_1,i_1,z_1),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_{12}(x_1,i_1,z_1),s),n) & [& \neg(i_1<x_1) \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_8(x_1,i_1,z_1),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_9(x_1,i_1,z_1),s),n) & & \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_9(x_1,i_1,z_1),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_{10}(x_1,i_1,v),s),n) & [& v=z_1+i_1+1 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_{10}(x_1,i_1,z_1),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_7(x_1,v,z_1),s),n) & [& v=i_1+1 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum2}(x_2),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum2}_{17}(x_2,v),s),n) & [& v=0 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum2}_{17}(x_2,z_2),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum2}_{18}(x_2,z_2),s),v) & [& v=n+1 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum2}_{18}(x_2,z_2),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum2}_{19}(x_2,v),s),n) & [& v=x_2\times(x_2+1)/2 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}(x_3),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{23}(x_3,v),s),n) & [& v=0 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{23}(x_3,z_3),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{24}(x_3,z_3),s),v) & [& v=n+1 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{24}(x_3,z_3),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{25}(x_3,z_3),s),n) & [& x_3\leq 0 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{24}(x_3,z_3),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{27}(x_3,z_3),s),n) & [& \neg(x_3\leq 0) \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{25}(x_3,z_3),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{31}(x_3,v),s),n) & [& v=0 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{27}(x_3,z_3),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{28}(x_3,z_3),s),n) & & \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{28}(x_3,z_3),s),n) \to \qquad\qquad\qquad\qquad & & \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}(v_1),\mathsf{stack}(\mathsf{sum3}_{28}(x_3,z_3),s)),n) & [& v_1=x_3-1 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{31}(x_3^h,z_3^h),\mathsf{stack}(\mathsf{sum3}_{26}(x_3,z_3),s)),n) \qquad\qquad & & \\
\to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{29}(x_3,v),s),n) & [& v=z_3^h \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{29}(x_3,z_3),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{31}(x_3,v),s),n) & [& v=x_3+z_3 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{main},s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{main}_{36}(v),s),n) & [& v=0 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{main}_{36}(ret),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{main}_{37}(ret,v),s),n) & [& v=3 \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{main}_{37}(ret,z),s),n) \to \qquad\qquad\qquad\qquad & & \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}(v_1),\mathsf{stack}(\mathsf{main}_{37}(ret,z),s)),n) & [& v_1=z \;] \\
\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_{12}(x_1^h,i_1^h,z_1^h),\mathsf{stack}(\mathsf{main}_{37}(ret,z),s)),n) \qquad\qquad & & \\
\to \mathsf{cnfg}(\mathsf{stack}(\mathsf{main}_{38}(ret,v),s),n) & [& v=z_1^h \;]
\end{array}
\right\}
$$

Figure 3: The LCTRS obtained from Program 1.

Theorem 4.10 (correctness of $\mathfrak{T}$) *Let $cnfg_0$ be the initial (full) configuration of P. Then, for any natural number n, both of the following hold:*

- *For a full configuration $cnfg$, if $cnfg_0 \rightharpoonup_P^n cnfg$, then $\xi(cnfg_0) \to_{\mathfrak{T}(P)}^n \xi(cnfg)$, and*
- *for a ground term t : config, if $\xi(cnfg_0) \to_{\mathfrak{T}(P)}^n t$, then $cnfg_0 \rightharpoonup_P^n \xi^{-1}(t)$.*

Proof (Sketch). Using Lemmas 4.8 and 4.9, both claims can straightforwardly be proved by induction on n. $\square$

5 Conclusion

In this paper, we showed an injective function from configurations of SIMP$^+$ programs to terms, and then, using the injective function, we reformulate the definition and correctness proof of the transformation proposed in [5]. To show the usefulness of the proposed approach, we will compare the approach in this paper with the definition and correctness proof in [5] from the viewpoint of how plainer the approach is. Our future work is to extend this approach to SIMPLE [11], and then to concurrent programs with semaphore-based exclusive control in [7].

Acknowledgements We gratefully acknowledge the anonymous reviewers for their useful comments 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] Ştefan Ciobâcă & Dorel Lucanu (2018): *A Coinductive Approach to Proving Reachability Properties in Logically Constrained Term Rewriting Systems*. In Didier Galmiche, Stephan Schulz & Roberto Sebastiani, editors: *Proceedings of the 9th International Joint Conference on Automated Reasoning*, *Lecture Notes in Computer Science* 10900, Springer, pp. 295–311, doi:10.1007/978-3-319-94205-6_20.

[3] 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.

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

[5] Yoshiaki Kanazawa & Naoki Nishida (2019): *On Transforming Functions Accessing Global Variables into Logically Constrained Term Rewriting Systems*. In Joachim Niehren & David Sabel, editors: *Proceedings of the 5th International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, *Electronic Proceedings in Theoretical Computer Science* 289, Open Publishing Association, pp. 34–52, doi:10.4204/EPTCS.289.3.

[6] 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.

[7] Misaki Kojima, Naoki Nishida & Yutaka Matsubara (2020): *Transforming Concurrent Programs with Semaphores into Logically Constrained Term Rewrite Systems*. In Adrián Riesco & Vivek Nigam, editors: *Informal Proceedings of the 7th International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, pp. 1–12.

[8] 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.

[9] Naoki Nishida & Sarah Winkler (2018): *Loop Detection by Logically Constrained Term Rewriting*. In Ruzica Piskac & Philipp Rümmer, editors: *Proceedings of the 10th Working Conference on Verified Software: Theories, Tools, and Experiments*, *Lecture Notes in Computer Science* 11294, Springer, pp. 309–321, doi:10.1007/978-3-030-03592-1_18.

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

[11] Grigore Rosu & Traian-Florin Serbanuta (2014): *K Overview and SIMPLE Case Study*. In Mark Hills, editor: *Proceedings of International K Workshop*, *Electronic Notes in Theoretical Computer Science* 304, pp. 3–56, doi:10.1016/j.entcs.2014.05.002.

[12] Sarah Winkler & Aart Middeldorp (2018): *Completion for Logically Constrained Rewriting*. In Hélène Kirchner, editor: *Proceedings of the 3rd International Conference on Formal Structures for Computation and Deduction*, *LIPIcs* 108, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 30:1–30:18, doi:10.4230/LIPIcs.FSCD.2018.30.

A Partial Configurations

Partial configurations are formally defined as follows.

Definition A.1 (partial frames configurations) Partial frames *are inductively defined as follows:*

- *If σ is either a variable or an assignment with $\mathcal{D}om(\sigma) = dvars(\rho)$, then (ρ,σ) is a partial frame,*
- *if $Fun(\rho) \neq \rho$, v is a variable, σ is a variable for assignments, and $y \in \mathcal{D}om(\sigma)$, then $(\rho,\sigma[y \mapsto v])$ is a partial frame, and*
- *if $Fun(\rho) = \rho$, all $v_1,\ldots,v_m$ are variables, and $dvars(\rho) = \{y_1,\ldots,y_m\}$, then $(\rho,\varnothing[y_1 \mapsto v_1,\ldots, v_m \mapsto v_m])$ is a partial frame.*

Partial call stacks *are inductively defined as follows:*

- $[\,]$ *and a variable s for call stacks are partial call stacks, and*
- *if frm is a* partial frame *and s is a partial call stack, then $frm :: s$ is a partial call stack.*

Partial configurations *are inductively defined as follows:*

- *If $cstck$ is a* partial call stack *and σ_0 is either a variable or an assignment with $\mathcal{D}om(\sigma_0) = \{gv_1,\ldots,gv_k\}$, then $\langle cstck,\sigma_0\rangle$ is a partial configuration, and*
- *if $cstck$ is a partial call stack, v_i is a variable, σ_0 is a variable for assignments, and $1 \leq i \leq k$, then $\langle cstck,\sigma_0[gv_i \mapsto v_i]\rangle$ is a partial configuration.*

B Missing Proofs

Proposition 4.2 *All δ_{config}, δ_{cstack}, and $\delta_{\text{frame},h}$ are injective.*

Proof (Sketch). By definition, $\delta_{\text{frame},h}$ is injective: The second case is distinguishable from the other cases because of $\varnothing$; the first is distinguishable from the third and fourth because σ^h is an assignment but $\{y^h \mapsto v\}$ and $\{y^h_1 \mapsto v_1, \ldots, y^h_m \mapsto v_m\}$ are mappings from variables to variables; the third and fourth are distinguishable depending whether $Fun(\rho) = \rho$. It follows from the injectivity of $\delta_{\text{frame},h}$ that δ_{cstack} is injective. It follows from the injectivity of δ_{cstack} and the definition of δ_{config} that δ_{config} is injective: The second case is distinguishable from the first and third because of $\mathcal{D}om(\theta)$; the first and third are distinguishable because $(\theta \cup \sigma_0)(gv_i)$ is an integer and $(\theta \cup \{gv_i \mapsto v_i\})(gv_i)$ is a variable. □

Lemma 4.8 *Let P be a SIMP$^+$ program of the form (1) and For full configurations $cnfg_1, cnfg_2$ of P, if $cnfg_1 \rightharpoonup_P cnfg_2$, then $\xi(cnfg_1) \rightarrow_{\mathfrak{T}(P)} \xi(cnfg_2)$.*

Proof. We make a case analysis depending on which rule in $Rules(P)$ is applied to $cnfg_1$.

- Consider the case where one of $(\textbf{loc})[\rho]$, $(\textbf{g-assign})[\rho]$, $(\textbf{l-assign})[\rho]$, and $(\textbf{if}_\top)[\rho]$ is applied to $cnfg_1$. The rule can be represented as follows:

$$\frac{[\![\, \varphi(\sigma \cup \sigma_0) \,]\!] = \top \qquad [\![\, e(\sigma \cup \sigma_0) \,]\!] = v}{\langle (\rho,\sigma) :: s, \sigma_0\rangle \rightharpoonup \langle (\rho',\sigma') :: s, \sigma'_0\rangle}$$

  where

  - a fresh variable v for integers is assigned to a variable $y \in \{\overrightarrow{gv}, \overrightarrow{dvars}(\rho')\}$, and
  - if $y = gv_i$, then $\sigma' = \sigma$ and $\sigma'_0 = \sigma[gv_i \mapsto v]$, and otherwise, $\sigma' = \sigma[y \mapsto v]$ and $\sigma'_0 = \sigma_0$.

We only show the more complex case where $y \in dvars(\rho')$.
By the definition of $\mathfrak{T}_{\mathrm{rule}}$, we have that

$$\delta_{\mathrm{config}}(\langle(\rho,\sigma) :: s,\sigma_0\rangle) = (\mathsf{cnfg}(\mathsf{stack}(f_\rho(\overrightarrow{dvars}(\rho)),s),\overrightarrow{gv}),\varnothing)$$

and

$$\delta_{\mathrm{config}}(\langle(\rho',\sigma[y\mapsto v]) :: s,\sigma_0\rangle) = (\mathsf{cnfg}(\mathsf{stack}(f_{\rho'}(\overrightarrow{dvars}(\rho')),s),\overrightarrow{gv}),\{y\mapsto v\})$$

Thus, $\mathfrak{T}(P)$ includes the following constrained rule:

$$\begin{aligned}&\mathsf{cnfg}(\mathsf{stack}(f_\rho(\overrightarrow{dvars}(\rho)),s),\overrightarrow{gv}) \to\\ &\qquad\qquad \mathsf{cnfg}(\mathsf{stack}(f_{\rho'}(\overrightarrow{dvars}(\rho')),s),\overrightarrow{gv})\{y\mapsto v\}\ [\varphi\wedge v=e]\end{aligned}$$

Since the rule is applied to $cnfg_1$, there exist assignments σ_1,σ_2, an integer v', and a full call stack $cstck$ such that

- $cnfg_1 = \langle(\rho,\sigma_1) :: cstck,\sigma_2\rangle$,
- $cnfg_2 = \langle(\rho',\sigma_1[y\mapsto v']) :: cstck,\sigma_2\rangle$.
- $[\![\varphi(\sigma_1\cup\sigma_2)]\!] = \top$, and
- $[\![e(\sigma_1\cup\sigma_2)]\!] = v'$.

We now show that $\xi(cnfg_1) \to_{\mathfrak{T}(P)} \xi(cnfg_2)$. By the definition of ξ, we have that

$$\xi(cnfg_1) = \xi(\langle(\rho,\sigma_1) :: cstck,\sigma_2\rangle) = \mathsf{cnfg}(\mathsf{stack}(f_\rho(\overrightarrow{dvars}(\rho)),cstck'),\overrightarrow{gv})(\sigma_1\cup\sigma_2\cup\sigma_3)$$

and

$$\begin{aligned}\xi(cnfg_2) &= \xi(\langle(\rho',\sigma_1[y\mapsto v']) :: cstck,\sigma_2\rangle)\\ &= \mathsf{cnfg}(\mathsf{stack}(f_{\rho'}(\overrightarrow{dvars}(\rho')),cstck'),\overrightarrow{gv})(\sigma_1[y\mapsto v']\cup\sigma_2\cup\sigma_3)\\ &= \mathsf{cnfg}(\mathsf{stack}(f_{\rho'}(\overrightarrow{dvars}(\rho')),cstck'),\overrightarrow{gv})\{y\mapsto v'\}(\sigma_1\cup\sigma_2\cup\sigma_3)\end{aligned}$$

where $\xi(cstk) = (cstk',\sigma_3)$. Let $\theta = \sigma_1\cup\sigma_2\cup\sigma_3\cup\{s\mapsto cstck'(\sigma_1\cup\sigma_2\cup\sigma_3)\}$. Then, we have that $[\![\varphi\theta]\!] = \top$ and $[\![e\theta]\!] = v'$, and thus, θ respects the above constrained rewrite rule. Therefore, $\xi(cnfg_1) \to_{\mathfrak{T}(P)} \xi(cnfg_2)$.

- Consider the case where one of (**block**)$[\rho]$, (**if**$_\top$)$[\rho]$, (**if**$_\bot$)$[\rho]$, (**while**$_\top$)$[\rho]$, and (**while**$_\bot$)$[\rho]$ is applied to $cnfg_1$. The rule can be represented as follows:

$$\frac{[\![\varphi(\sigma\cup\sigma_0)]\!] = \top}{\langle(\rho,\sigma) :: s,\sigma_0\rangle \rightharpoonup \langle(\rho',\sigma) :: s,\sigma_0\rangle}$$

This case is analogous to the previous one.

- Consider the case where (**call**)$[\rho]$ is applied to $cnfg_1$. The rule can be represented as follows:

$$\frac{[\![e_1(\sigma\cup\sigma_0)]\!] = v_1 \qquad \dots \qquad [\![e_{n'}(\sigma\cup\sigma_0)]\!] = v_{n'}}{\langle(\rho,\sigma) :: s,\sigma_0\rangle \rightharpoonup \langle(g,\varnothing[y_1\mapsto v_1,\dots,y_{n'}\mapsto v_{n'}]) :: (\rho,\sigma) :: s,\sigma_0\rangle}\ \textbf{(call)}$$

where $Stmt(\rho) = (x\,\texttt{=}\,g\texttt{(}e_1\texttt{,}\dots\texttt{,}e_{n'}\texttt{);})$ and $\overrightarrow{dvars}(g) = y_1,\dots,y_{n'}$.
By the definition of $\mathfrak{T}_{\mathrm{rule}}$, we have that

$$\delta_{\mathrm{config}}(\langle(\rho,\sigma) :: s,\sigma_0\rangle) = (\mathsf{cnfg}(\mathsf{stack}(f_\rho(\overrightarrow{dvars}(\rho)),s),\overrightarrow{gv}),\varnothing)$$

and

$$\begin{aligned}&\delta_{\mathrm{config}}(\langle (g,\varnothing[y_1 \mapsto v_1,\dots,y_{n'} \mapsto v_{n'}]) :: (\rho,\sigma) :: s, \sigma_0\rangle)\\ &\quad = (\mathsf{cnfg}(\mathsf{stack}(g(y^h_1,\dots,y^h_{n'}),\mathsf{stack}(f_\rho(\overrightarrow{dvars}(\rho)),s)),\overrightarrow{gv}), \{y^h_1 \mapsto v_1,\,\dots,\,y^h_{n'} \mapsto v_{n'}\})\end{aligned}$$

Thus, $\mathfrak{T}(P)$ includes the following constrained rule:

$$\begin{aligned}&\mathsf{cnfg}(\mathsf{stack}(f_\rho(\overrightarrow{dvars}(\rho)),s),\overrightarrow{gv}) \to\\ &\quad \mathsf{cnfg}(\mathsf{stack}(g(y^h_1,\dots,y^h_{n'}),\mathsf{stack}(f_\rho(\overrightarrow{dvars}(\rho)),s)),\overrightarrow{gv})\{y^h_1 \mapsto v_1,\,\dots,\,y^h_{n'} \mapsto v_{n'}\}\\ &\qquad\qquad [\,v_1 = y^h_1 \wedge \cdots \wedge v_{n'} = y^h_{n'}\,]\end{aligned}$$

Since the rule is applied to $cnfg_1$, there exist assignments σ_1, σ_2, integers $v'_1,\dots,v'_{n'}$, and a full call stack $cstck$ such that

- $cnfg_1 = \langle (\rho,\sigma_1) :: cstck, \sigma_2\rangle$,
- $cnfg_2 = \langle (g,\varnothing[y_1 \mapsto v'_1,\dots,y'_{n'} \mapsto v_{n'}]) :: (\rho,\sigma_1) :: cstck, \sigma_2\rangle$, and
- $[\![\, e_i(\sigma_1 \cup \sigma_2)\,]\!] = v'_i$ for all $1 \le i \le n'$.

We now show that $\xi(cnfg_1) \to_{\mathfrak{T}(P)} \xi(cnfg_2)$. By the definition of ξ, we have that

$$\xi(cnfg_1) = \xi(\langle (\rho,\sigma_1) :: cstck, \sigma_2\rangle) = \mathsf{cnfg}(\mathsf{stack}(f_\rho(\overrightarrow{dvars}(\rho)), cstck'),\overrightarrow{gv})(\sigma_1 \cup \sigma_2 \cup \sigma_3)$$

and

$$\begin{aligned}&\xi(cnfg_2) = \xi(\langle (g,\varnothing[y_1 \mapsto v'_1,\dots,y'_{n'} \mapsto v_{n'}]) :: (\rho,\sigma_1) :: cstck, \sigma_2\rangle)\\ &= \mathsf{cnfg}(\mathsf{stack}(g(y^h_1,\dots,y^h_{n'}),\mathsf{stack}(f_\rho(\overrightarrow{dvars}(\rho)),cstck')),\overrightarrow{gv})(\sigma_1[y^h_1 \mapsto v'_1,\dots,y^h_{n'} \mapsto v'_{n'}] \cup \sigma_2 \cup \sigma_3)\\ &= \mathsf{cnfg}(\mathsf{stack}(g(y^h_1,\dots,y^h_{n'}),\mathsf{stack}(f_\rho(\overrightarrow{dvars}(\rho)),cstck')),\overrightarrow{gv})\{y^h_1 \mapsto v'_1,\dots,y^h_{n'} \mapsto v'_{n'}\}(\sigma_1 \cup \sigma_2 \cup \sigma_3)\end{aligned}$$

where $\xi(cstk) = (cstk', \sigma_3)$. Let $\theta = \sigma_1 \cup \sigma_2 \cup \sigma_3 \cup \{s \mapsto cstck'(\sigma_1 \cup \sigma_2 \cup \sigma_3)\}$. Then, we have that $[\![\, e_i\theta\,]\!] = v'_i$ for all $1 \le i \le n'$, and thus, θ respects the above constrained rewrite rule. Therefore, $\xi(cnfg_1) \to_{\mathfrak{T}(P)} \xi(cnfg_2)$.

- Consider the remaining case where (**return**)$[\rho]$ is applied to $cnfg_1$. This case is analogous to the case where (**call**)$[\rho]$ is applied to $cnfg_1$. □

C Detail of Example 4.7

For example, (**loc**)[$\mathsf{sum1}$] in $Rules(P_1)$ is transformed as follows:

$$\begin{aligned}\mathfrak{T}_{\mathrm{rule}}((\mathbf{loc})[\mathsf{sum1}]) &= \mathfrak{T}_{\mathrm{rule}}\Big(\frac{[\![\,0(\sigma \cup \sigma_0)\,]\!] = v}{\langle (\mathsf{sum1},\sigma) :: s, \sigma_0\rangle \to_{P_1} \langle (5,\sigma[i_1 \mapsto v]) :: s, \sigma_0\rangle}\Big)\\ &= \xi(\langle (\mathsf{sum1},\sigma) :: s, \sigma_0\rangle) \to \xi(\langle (5,\sigma[i_1 \mapsto v]) :: s, \sigma_0\rangle)\ [\,v = 0\,]\\ &= \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}(x_1),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_5(x_1,i_1),s),n)\{i_1 \mapsto v\}\ [\,v = 0\,]\\ &= \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}(x_1),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_5(x_1,v),s),n)\ [\,v = 0\,]\end{aligned}$$

where $Stmt(\mathsf{sum1}) = (\texttt{int } i_1 \texttt{ = 0;})$, s, σ, σ_0, v are variables,

$$\xi(\langle (\mathsf{sum1},\sigma) :: s, \sigma_0\rangle) = (\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}(x_3),s),n),\varnothing)$$

by $\delta_{\mathrm{cstack}}((\mathsf{sum1},\sigma) :: s) = (\mathsf{stack}(\mathsf{sum1}(x_3),s),\varnothing)$ and $\delta_{\mathrm{frame},0}((\mathsf{sum1},\sigma)) = (\mathsf{sum1}(x_3),\varnothing)$, and

$$\xi(\langle (5,\sigma[i_1 \mapsto v]) :: s, \sigma_0\rangle) = (\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum1}_5(x_1,i_1),s),n),\{i_1 \mapsto v\})$$

by $\delta_{\text{cstack}}((5,\sigma[i_1 \mapsto v]) :: s) = (\mathsf{stack}(\mathsf{sum1}_5(x_1,i_1),s),\{i_1 \mapsto v\})$ and $\delta_{\text{cstack}}((5,\sigma[i_1 \mapsto v])) = (\mathsf{sum1}_5(x_1,i_1),\{i_1 \mapsto v\})$.

In addition, (**call**)[28] in $\mathit{Rules}(P_1)$ is transformed as follows:

$$\begin{array}{l}\mathfrak{T}_{\text{rule}}((\mathbf{call})[28]) = \mathfrak{T}_{\text{rule}}\left(\dfrac{[\![\,(x_3-1)(\sigma\cup\sigma_0)\,]\!] = v_1}{\langle (28,\sigma) :: s, \sigma_0\rangle \to_{P_1} \langle (\mathsf{sum3},\varnothing[x_3 \mapsto v_1]) :: (28,\sigma) :: s, \sigma_0\rangle}\right)\\ = \xi(\langle (28,\sigma) :: s,\sigma_0\rangle) \to \xi(\langle (\mathsf{sum3},\varnothing[x_3\mapsto v_1]) :: (28,\sigma) :: s,\sigma_0\rangle)\ [\,v_1 = x_3 - 1\,]\\ = \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{28}(x_3,z_3),s),n) \to \mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}(v_1),\mathsf{stack}(\mathsf{sum3}_{28}(x_3,z_3),s)),n)\ [\,v_1 = x_3 - 1\,]\end{array}$$

where $\mathit{Stmt}(28) = (z_3\ \texttt{=}\ \mathsf{sum3}\texttt{(}x_3\texttt{-1);})$, all s,σ,σ_0,v_1 are variables,

$$\xi(\langle (28,\sigma) :: s,\sigma_0\rangle) = (\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}_{28}(x_3,z_3),s),n),\varnothing)$$

by $\delta_{\text{cstack}}((28,\sigma) :: s) = (\mathsf{stack}(\mathsf{sum3}_{28}(x_3,z_3),s),\varnothing)$ and $\delta_{\text{frame},0}((28,\sigma)) = (\mathsf{sum3}_{28}(x_3,z_3),\varnothing)$, and

$$\begin{array}{l}\xi(\langle (\mathsf{sum3},\varnothing[x_3\mapsto v_1]) :: (28,\sigma) :: s,\sigma_0\rangle) =\\ \qquad\qquad (\mathsf{cnfg}(\mathsf{stack}(\mathsf{sum3}(x_3^h),\mathsf{stack}(\mathsf{sum3}_{28}(x_3,z_3),s),n),\{x_3^h \mapsto v_1\})\end{array}$$

by $\delta_{\text{cstack}}((\mathsf{sum3},\varnothing[x_3 \mapsto v_1]) :: (28,\sigma) :: s) = (\mathsf{stack}(\mathsf{sum3}(x_3^h),\mathsf{stack}(\mathsf{sum3}_{28}(x_3,z_3),s),\{x_3^h \mapsto v_1\})$, $\delta_{\text{frame},0}((28,\sigma)) = (\mathsf{sum3}_{28}(x_3,z_3),\varnothing)$, and $\delta_{\text{frame},h}((\mathsf{sum3},\varnothing[x_3\mapsto v_1])) = (\mathsf{sum3}(x_3^h),\{x_3^h \mapsto v_1\})$. The initial configuration $\langle [(\mathsf{main},\varnothing)],\{n \mapsto 0\}\rangle$ is represented by the term $\mathsf{cnfg}(\mathsf{stack}(\mathsf{main},\mathsf{empty}),0)$.