Transforming Proof Tableaux of Hoare Logic into Inference Sequences of Rewriting Induction

Shinnosuke Mizutani
Graduate School of Information Science
Nagoya University
Nagoya, Japan
mizutani_s@trs.cm.is.nagoya-u.ac.jp

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

A proof tableau of Hoare logic is an annotated program with pre- and post-conditions, which corresponds to an inference tree of Hoare logic. In this paper, using an example, we illustrate a top-down transformation of a proof tableau for partial correctness into an inference sequence of rewriting induction for constrained rewriting. The resulting sequence is a valid proof for the equation corresponding to the Hoare triple if the constrained rewriting system obtained from the program is terminating. Such a transformation enables us to apply techniques for proving termination of constrained rewriting to proving total correctness of programs together with proof tableaux for partial correctness.

1 Introduction

In the field of term rewriting, automated reasoning about *inductive theorems* has been well investigated. Here, an inductive theorem of a term rewriting system (TRS) is an equation that is inductively valid, i.e., all of its ground instances are theorems of the TRS. As principles for proving inductive theorems, we cite *inductionless induction* [14, 10] and *rewriting induction* (RI) [16], both of which are called *implicit induction principles*. Frameworks based on the RI principle (RI frameworks, for short) consist of inference rules to prove inductive validity, and RI-based methods are procedures within RI frameworks to apply inference rules under specified strategies. In recent years, various RI-based methods for *constrained rewriting* (see, e.g., constrained TRSs [9, 18], conditional and constrained TRSs [2], $\mathbb{Z}$-TRSs [6], and logically constrained TRSs [11]) have been developed [2, 19, 6, 12, 8]. Constrained systems have built-in semantics for some function and predicate symbols and have been used as a computation model of not only functional but also imperative programs [4, 7, 9, 5, 20, 12, 8].

For program verification, several techniques have been investigated in the literature, e.g., *model checking*, *Hoare logic*, etc. On the other hand, constrained rewriting can be used as a model of some imperative programs (cf. [8]), and RI frameworks for constrained rewriting are tuned to verification of imperative programs, e.g. equivalence of two functions under the same specification. In some cases where a proof based on Hoare logic needs a loop invariant, some RI frameworks succeed in proving equivalence of an imperative program and its functional specification (cf. [8]). From such experiences, we are interested in the difference between RI frameworks and other verification methods.

In this paper, using an example, we illustrate a top-down transformation of a *proof tableau* of Hoare logic into an inference sequence of rewriting induction for logically constrained TRSs (LCTRSs). Here, a proof tableau is an annotated *while* program with pre- and post-conditions, which corresponds to an inference tree of Hoare logic. The resulting inference sequence is a valid proof for an inductive theorem corresponding to the Hoare triple for the proof tableau if the LCTRS obtained from the program is terminating. More precisely, given a proof tableau for *partial correctness*, we proceed as follows: (1) We transform the *while* program obtained by removing annotations from the proof tableau into an LCTRS; (2) We prepare rewrite rules to verify the postcondition in the proof tableau; (3) We prepare a

Submitted to:
WPTE 2017

constrained equation corresponding to the Hoare triple of the proof tableau; (4) Starting with the equation, we transform the proof tableau into an inference sequence of RI from top to bottom, where we do not prove termination in constructing inference sequences of RI; (5) We prove termination of the LCTRS with generated rules. Termination of the LCTRS with generated rules ensures that the resulting inference sequence is a valid proof of RI (i.e., the equation is an inductive theorem of the LCTRS), and also that the *while* program is *totally correct*. The transformation enables us to apply techniques for proving termination of constrained rewriting to proving total correctness of programs together with proof tableaux for partial correctness.

The contribution of this paper is a top-down transformation of proof tableaux for partial correctness to inference sequences of RI, which applies techniques for proving termination of constrained rewriting to proving total correctness.

2 Preliminaries

In this section, we recall *logically constrained term rewriting systems* (LCTRS, for short), following the definitions in [11, 8]. We also recall *while programs*, and then introduce a conversion of *while* programs to LCTRSs. Familiarity with basic notions on term rewriting [1, 15] 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 is denoted by $\mathcal{V}ar(s)$. Given a term s and a *position* p (a sequence of positive integers) of s, $s|_p$ denotes the subterm of s at position p, and $s[t]_p$ denotes s with the subterm at position p replaced by t.

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

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

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

$=_\iota, \neq_\iota : \iota \times \iota \Rightarrow \mathit{bool}$, and an evaluation function $\mathcal{J}$ that interprets these symbols as expected. We omit the sort subscripts from $=$ and $\neq$ when clear from context.

The standard integer signature $\Sigma^{\mathit{int}}_{\mathit{theory}}$ is $\Sigma^{\mathit{core}}_{\mathit{theory}} \cup \{+, -, *, \mathsf{exp}, \mathsf{div}, \mathsf{mod} : \mathit{int} \times \mathit{int} \Rightarrow \mathit{int}\} \cup \{\geq, > : \mathit{int} \times \mathit{int} \Rightarrow \mathit{bool}\} \cup \{\mathsf{n} : \mathit{int} \mid n \in \mathbb{Z}\}$ with values true, false, and n for all integers $n \in \mathbb{Z}$. Thus, we use n (in sans-serif font) as the function symbol for $n \in \mathbb{Z}$ (in *math* font). We define $\mathcal{J}$ in the natural way, except: since all $\mathcal{J}(f)$ must be total functions, we set $\mathcal{J}(\mathsf{div})(n,0) = \mathcal{J}(\mathsf{mod})(n,0) = \mathcal{J}(\mathsf{exp})(n,k) = 0$ for all n and all $k < 0$.

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

Now we define a *logically constrained term rewriting system* (LCTRS) as the abstract rewriting system $(T(\Sigma, \mathcal{V}), \to_{\mathcal{R}})$. An LCTRS is usually given by supplying Σ, $\mathcal{R}$, and an informal description of $\mathcal{I}$ and $\mathcal{J}$ if these are not clear from context. 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 $\mathcal{R}$.

Example 2.1 ([8]) Let $\mathcal{S} = \{\mathit{int}, \mathit{bool}\}$, and $\Sigma = \Sigma_{\mathit{terms}} \cup \Sigma^{\mathit{int}}_{\mathit{theory}}$, where $\Sigma_{\mathit{terms}} = \{\ \mathsf{fact} : \mathit{int} \Rightarrow \mathit{int}\ \} \cup \{\ \mathsf{n} : \mathit{int} \mid n \in \mathbb{Z}\ \}$. Then both *int* and *bool* are theory sorts. We also define set and function interpretations, i.e., $\mathcal{I}_{\mathit{int}} = \mathbb{Z}$, $\mathcal{I}_{\mathit{bool}} = \mathbb{B}$, and $\mathcal{J}$ is defined as above. With $=$ for $=_{\mathit{int}}$ and infix notation, examples of logical terms are $0 = 0 + -1$ and $x + 3 \geq y + -42$ that are constraints. $5 + 9$ is also a (ground) logical term, but not a constraint. Expected starting terms are, e.g., $\mathsf{fact}(42)$ or $\mathsf{fact}(\mathsf{fact}(-4))$. To implement an LCTRS calculating the *factorial* function, we use the signature Σ above and the following rules: $\mathcal{R}_{\mathsf{fact}} = \{\ \mathsf{fact}(x) \to 1\ [x \leq 0],\ \ \mathsf{fact}(x) \to x \times \mathsf{fact}(x-1)\ [\neg(x \leq 0)]\ \}$. Using calculation steps, a term $3 - 1$ reduces to 2 in one step with the calculation rule $x - y \to z\ [z = x - y]$, and $3 \times (2 \times (1 \times 1))$ reduces to 6 in three steps. Using the constrained rewrite rules in $\mathcal{R}_{\mathsf{fact}}$, $\mathsf{fact}(3)$ reduces in ten steps to 6.

A *constrained term* is a pair $s\,[\varphi]$ of a term s and a constraint φ. We say that $s\,[\varphi]$ and $t\,[\psi]$ are *equivalent*, written by $s\,[\varphi] \sim t\,[\psi]$, if for all substitutions γ which respect φ, there is a substitution δ which respects ψ such that $s\gamma = t\delta$, and vice versa. Intuitively, a constrained term $s\,[\varphi]$ represents all terms $s\gamma$ where γ respects φ, and can be used to reason about such terms. For this reason, equivalent constrained terms represent the same set of terms. For a rule $\rho := \ell \to r\ [\psi] \in \mathcal{R} \cup \mathcal{R}_{\mathtt{calc}}$ and position q, we let $s\,[\varphi] \to_{\rho,q} t\,[\varphi]$ if there exists a substitution γ such that $s|_q = \ell\gamma$, $t = s[r\gamma]_q$, $\gamma(x)$ is either a value or a variable in $\mathcal{V}ar(\varphi)$ for all $x \in \mathcal{LV}ar(\ell \to r\ [\psi])$, and $\varphi \implies (\psi\gamma)$ is valid. We write $s\,[\varphi] \to_{\mathtt{base}} t\,[\varphi]$ for $s\,[\varphi] \to_{\rho,q} t\,[\varphi]$ with some ρ, q. The relation $\to_{\mathcal{R}}$ on constrained terms is defined as $\sim \cdot \to_{\mathtt{base}} \cdot \sim$.

2.2 While Programs

In this section, we recall the syntax of *while programs* (see e.g., [17]).

We deal with a simple class of *while* programs over the integers, which consist of assignments, skip, sequences, "if" statements, and "while" statements with loop invariants: a "while" statement is of the

form **while** $@\,\varphi\,(\psi)\{c\}$ with φ a loop invariant. To deal with proof tableaux, we allow to write assertions of the form $@\varphi$ as annotations. An *annotated while program* is defined by the following BNF:

$$\begin{array}{lcl} P & ::= & v := E \mid \textbf{skip} \mid P;P \mid @\,B \mid \textbf{if}(B)\{P\}\textbf{else}\{P\} \mid \textbf{while}\,@\,B\,(B)\{P\} \\ E & ::= & n \mid v \mid (E+E) \mid (E-E) \mid (E*E) \mid (E\,/\,E) \\ B & ::= & \mathsf{true} \mid \mathsf{false} \mid E=E \mid E<E \mid (\neg B) \mid (B \vee B) \end{array}$$

where $n \in \mathbb{Z}$, $v \in \mathcal{V}$, and we may omit brackets in the usual way. We use $\neq$, $\leq$, $>$, $\geq$, $\wedge$, $\Longrightarrow$, etc, as syntactic sugars. We abbreviate **while** $@\,\mathsf{true}\,(\psi)\{c\}$ to **while**$(\psi)\{c\}$. For page limitation, we do not introduce the semantics of *while* programs, and they are evaluated in the usual way—in evaluating *while* programs, we ignore loop invariants and assertions, while they are taken into account in considering proof tableaux. For a *while* program P, we denote the set of variables appearing in P by $\mathcal{V}ar(P)$.

Example 2.2 The following is a *while* program, denoted by P_{sum}, with $\mathcal{V}ar(P_{sum}) = \{x,i,z\}$, which computes the summation from 0 to x if $x \geq 0$.

```
1  i := 0;
2  z := 0;
3  while(x > i){
4      z := z + i + 1;
5      i := i + 1;
6  }
7
```

We write a line number for each statement, and write a blank line at the end of the program, which is used to simplify a conversion of *while* programs to LCTRSs.

2.3 Converting *while* Programs to LCTRSs

In this section, we briefly introduce a conversion of *while* programs to LCTRSs (see e.g., [8]).

Let P be a *while* program and $\mathcal{V}ar(P) = \{x_1,\ldots,x_n\}$. We denote the sequence "$x_1,\ldots,x_n$" by $\vec{x}$. We prepare a sort *state* for tuples of integers. We assume that there is no blank line in P with line numbers, except for the last line m e.g., line 7 of P_{sum}. We first prepare $m+1$ function symbols $\mathsf{state}_1,\ldots,\mathsf{state}_m,\mathsf{end}$ with sort $\overbrace{\mathbb{Z}\times\cdots\times\mathbb{Z}}^{n} \Rightarrow \mathit{state}$. For brevity, we use state_m, but we identify state_m and end, replacing state_m by end in the final result. Instances of $\mathsf{state}_1,\ldots,\mathsf{state}_m,\mathsf{end}$ represent *states* in executing P—a state consists of a program counter and an assignment to variables in the program (see e.g., [3]). For example, $\mathsf{state}_i(v_1,\ldots,v_n)$ represents a state such that the program counter is i and $v_1,\ldots,v_n$ are assigned to $x_1,\ldots,x_n$, resp. For each statement in P, following Table 1, we generate constrained rewrite rules for $\mathsf{state}_1,\ldots,\mathsf{state}_m,\mathsf{end}$.

Example 2.3 The program P_{sum} in Example 2.2 is converted to the LCTRS $\mathcal{R}_{sum}$ in Figure 1. $\mathcal{R}_{sum}$ is non-overlapping (and thus, locally confluent), *quasi-reductive* (i.e., every ground term with a defined symbol is reducible), and terminating. Note that termination of $\mathcal{R}_{sum}$ can be proved by Ctrl [13].

3 Proof Tableaux of Hoare Logic

Hoare logic is a logic to prove a Hoare triple $\{\varphi\}\ P\ \{\psi\}$ to hold (see e.g., [17]). In this paper, we consider proof tableaux for *partial correctness* only. A triple $\{\varphi\}\ P\ \{\psi\}$ is said to *hold* (or P is *partially correct* w.r.t. precondition φ and postcondition ψ) if for any initial state satisfying φ, the final state of

Table 1: conversion of statements to constrained rewrite rules.

statement		generated constrained rewrite rules
i	$x_k := e$;	$\mathsf{state}_i(\vec{x}) \to \mathsf{state}_{i+1}(x_1,\ldots,x_{k-1},e,x_{k+1},\ldots,x_n)$
i	**skip**;	$\mathsf{state}_i(\vec{x}) \to \mathsf{state}_{i+1}(\vec{x})$
i	**if**$(\varphi)\{$	
$\vdots$	$\ldots$	$\mathsf{state}_i(\vec{x}) \to \mathsf{state}_{i+1}(\vec{x})$ $[\ \varphi\ ]$
j	$\}$**else**$\{$	$\mathsf{state}_i(\vec{x}) \to \mathsf{state}_{j+1}(\vec{x})$ $[\neg\varphi]$
		$\mathsf{state}_j(\vec{x}) \to \mathsf{state}_{k+1}(\vec{x})$
$\vdots$	$\ldots$	$\mathsf{state}_k(\vec{x}) \to \mathsf{state}_{k+1}(\vec{x})$
k	$\}$	
i	**while** $@\,\psi\,(\varphi)\{$	$\mathsf{state}_i(\vec{x}) \to \mathsf{state}_{i+1}(\vec{x})$ $[\ \varphi\ ]$
$\vdots$	$\ldots$	$\mathsf{state}_i(\vec{x}) \to \mathsf{state}_{j+1}(\vec{x})$ $[\neg\varphi]$
j	$\}$	$\mathsf{state}_j(\vec{x}) \to \mathsf{state}_i(\vec{x})$

$$\mathcal{R}_{sum} = \left\{ \begin{array}{ll} \mathsf{state}_1(x,i,z) \to \mathsf{state}_2(x,0,z) & \\ \mathsf{state}_2(x,i,z) \to \mathsf{state}_3(x,i,0) & \\ \mathsf{state}_3(x,i,z) \to \mathsf{state}_4(x,i,z) & [\ \ x > i\ \ ] \\ \mathsf{state}_3(x,i,z) \to \mathsf{end}(x,i,z) & [\neg(x > i)] \\ \mathsf{state}_4(x,i,z) \to \mathsf{state}_5(x,i,z+i+1) & \\ \mathsf{state}_5(x,i,z) \to \mathsf{state}_6(x,i+1,z) & \\ \mathsf{state}_6(x,i,z) \to \mathsf{state}_3(x,i,z) & \end{array} \right\}$$

Figure 1: the LCTRS $\mathcal{R}_{sum}$ obtained from P_{sum}.

the execution satisfies ψ whenever the execution from the initial state terminates. The aim of this paper is to transform a proof tableau of Hoare logic into an inference sequence of RI (shown in Section 4). For this reason, we do not focus on the construction of proof tableaux. In this section, we formalize proof tableaux of Hoare triple. We consider *while* programs as sequences of commands connected by ";", and we write P as $C_1;C_2;\ldots;C_n$. Bodies of "if" and "while" statements are also considered sequences of commands. Note that we consider ";" to implicitly exist at the end of "if" and "while" statements.

Definition 3.1 *An annotated* while *program P is called a* proof tableau *if all of the following hold:*

- *every longest command-(sub)sequence in P has the length more than two and the head and the last element of the sequence are annotations, e.g., P is of the form $@\,\varphi;C_1;\ldots;C_n;@\,\psi$ ($n > 0$) with the* precondition φ *and the* postcondition ψ*,*
- *for each subsequence $C_1;C_2$ where C_1 and C_2 are annotations $@\,\varphi$ and $@\,\psi$, resp., $\varphi \implies \psi$ is valid, and*
- *for each subsequence $C_1;C_2;C_3$ where C_2 is not an annotation, C_1,C_3 are annotations such that*
  - *if C_2 is an assignment $x := e$, then C_1 is $C_3\{x \mapsto e\}$,*
  - *if C_2 is* **skip***, then C_1 and C_3 are equivalent,*
  - *if C_2 is of the form* **if**$(\psi)\{S'\}$**else**$\{S''\}$*, then C_1 is $@\,\varphi$, the head of S' is $@\,\varphi \wedge \psi$, the head of S'' is $@\,\varphi \wedge \neg\psi$, and C_3 and the last elements of S' and S'' are equivalent, and*
  - *if C_2 is of the form* **while** $@\,\varphi\,(\psi)\{S\}$*, then C_1 is φ, the head of S is $@\ \varphi \wedge \psi$, C_3 is $@\,\varphi \wedge \neg\psi$, and the last element of the sequence S is $@\varphi$.*

In other words, a proof tableau is a tableau version of an inference tree constructed by basic inference rules of Hoare logic illustrated in Figure 2 (see e.g., [17]).

$$\frac{\varphi \implies \varphi' \text{ is valid} \quad \{\varphi'\}\ C\ \{\psi'\} \quad \psi' \implies \psi \text{ is valid}}{\{\varphi\}\ C\ \{\psi\}} \qquad \frac{}{\{\varphi\{v \mapsto e\}\}\ v := e\ \{\varphi\}} \qquad \frac{}{\{\varphi\}\ \mathbf{skip}\ \{\varphi\}}$$

$$\frac{\{\varphi\}\ C_1\ \{\xi\} \quad \{\xi\}\ C_2\ \{\psi\}}{\{\varphi\}\ C_1; C_2\ \{\psi\}} \qquad \frac{\{\varphi \wedge \psi\}\ C_1\ \{\xi\} \quad \{\varphi \wedge \neg\psi\}\ C_2\ \{\xi\}}{\{\varphi\}\ \mathbf{if}(\psi)\{C_1\}\mathbf{else}\{C_2\}\ \{\xi\}} \qquad \frac{\{\varphi \wedge \psi\}\ C\ \{\varphi\}}{\{\varphi\}\ \mathbf{while}\ @\ \varphi\,(\psi)\{C\}\ \{\varphi \wedge \neg\psi\}}$$

Figure 2: basic inference rules of Hoare logic.

```
A1    @ x ≥ 0;
A2    @ x ≥ 0 ∧ 0 = 0;
1     i := 0;
A3    @ x ≥ 0 ∧ i = 0;
A4    @ x ≥ 0 ∧ i = 0 ∧ 0 = 0;
2     z := 0;
A5    @ x ≥ 0 ∧ i = 0 ∧ z = 0;
A6    @ z = 1/2 i(i+1) ∧ x ≥ i;
3     while @ z = 1/2 i(i+1) ∧ x ≥ i (x > i){
A7        @ z = 1/2 i(i+1) ∧ x ≥ i ∧ x > i;
A8        @ z+i+1 = 1/2 (i+1)(i+2) ∧ x ≥ i+1;
4         z := z+i+1;
A9        @ z = 1/2 (i+1)(i+2) ∧ x ≥ i+1;
5         i := i+1;
A10       @ z = 1/2 i(i+1) ∧ x ≥ i;
6     }
A11   @ z = 1/2 i(i+1) ∧ x ≥ i ∧ ¬(x > i);
A12   @ z = 1/2 x(x+1);
7
```

Figure 3: an annotated *while* program P_{tab} for P_{sum}.

Example 3.2 The annotated *while* program of Figure 3, denoted by P_{tab}, is a proof tableau for the Hoare triple $\{x \geq 0\}\ P_{sum}\ \{z = \frac{1}{2}x(x+1)\}$, where the original line numbers for P_{sum} are left.

4 Rewriting Induction on LCTRSs

In this section, we recall the framework of *rewriting induction* (RI) for LCTRSs [8], showing a simpler version. As in [8], we restrict LCTRSs to be terminating and quasi-redutive.

A *constrained equation* is a triple $s \approx t\ [\varphi]$. We may simply write $s \approx t$ instead of $s \approx t\ [\varphi]$ if φ is true. We write $s \simeq t\ [\varphi]$ to denote either $s \approx t\ [\varphi]$ or $t \approx s\ [\varphi]$. A substitution γ is said to *respect* $s \approx t\ [\varphi]$ if γ respects φ and $\mathcal{V}ar(s) \cup \mathcal{V}ar(t) \subseteq \mathcal{D}om(\gamma)$, and to be a *ground constructor substitution* if all $\gamma(x)$ with $x \in \mathcal{D}om(\gamma)$ are ground constructor terms. An equation $s \approx t\ [\varphi]$ is called an *inductive theorem* of an LCTRS $\mathcal{R}$ if $s\gamma \leftrightarrow^*_{\mathcal{R}} t\gamma$ for any ground constructor substitution γ that respects $s \approx t\ [\varphi]$.

An RI-based method is to construct an *inference sequence* by applying the following basic inference rules to pairs of finite sets $\mathcal{E}$ and $\mathcal{H}$ of constrained equations and rewrite rules:

- EXPANSION $(\mathcal{E} \uplus \{s \simeq t\ [\varphi]\}, \mathcal{H}) \vdash_{RI} (\mathcal{E} \cup Expd_{\mathcal{R}}(s \approx t\ [\varphi], p), \mathcal{H} \cup \{s \to t\ [\varphi]\})$ where p is a *basic* position of s,[1] $Expd_{\mathcal{R}}(s \approx t\ [\varphi], p) = \{s' \approx t'\ [\varphi'] \mid s\gamma \approx t\gamma\ [\varphi\gamma \wedge \psi\gamma] \to_{1.p,\ell\to r\ [\psi]} s' \approx t'\ [\varphi'],\ \ell \to r\ [\psi]$ is a renamed variant of a rule in $\mathcal{R}$, γ is a most general unifier of $s|_p$ and $\ell\}$, and $\mathcal{R} \cup \mathcal{H} \cup \{s \to t\ [\varphi]\}$ is terminating. Note that $\approx$ is considered a binary function symbol in constrained rewriting.
- SIMPLIFICATION $(\mathcal{E} \uplus \{s \simeq t\ [\varphi]\}, \mathcal{H}) \vdash_{RI} (\mathcal{E} \cup \{u \approx t\ [\psi]\}, \mathcal{H})$ where $s\,[\varphi] \to_{\mathcal{R} \cup \mathcal{H}} u\,[\psi]$.
- DELETION $(\mathcal{E} \uplus \{s \approx t\ [\varphi]\}, \mathcal{H}) \vdash_{RI} (\mathcal{E}, \mathcal{H})$ where $s = t$ or φ is not satisfiable.
- GENERALIZATION $(\mathcal{E} \uplus \{s \approx t\ [\varphi]\}, \mathcal{H}) \vdash_{RI} (\mathcal{E} \cup \{s \approx t\ [\psi]\}, \mathcal{H})$ where $\varphi \implies \psi$ is valid. Note that this is a simpler version of the original one in [8].

A pair $(\mathcal{E}, \mathcal{H})$ is called a *process* of RI. Starting with $(\mathcal{E}, \emptyset)$, we apply the inference rules above to processes of RI. If we get $(\emptyset, \mathcal{H})$, all the equations in $\mathcal{E}$ are proved to be inductive theorems of $\mathcal{R}$.

Next, we revisit the role of termination in the RI method. When we apply EXPANSION to $(\mathcal{E}_i, \mathcal{H}_i)$, we prove termination of $\mathcal{R} \cup \mathcal{H}_i \cup \{s \to t\ [\varphi]\}$. This is necessary to avoid both constructing an incorrect

[1] A position of p of term s is *basic* if $s|_p$ is of the form $f(s_1, \ldots, s_n)$ with f a defined symbol and $s_1, \ldots, s_n$ constructor terms.

inference sequence and applying SIMPLIFICATION infinitely many times. However, from theoretical viewpoint, it suffices to prove termination of $\mathcal{R} \cup \mathcal{H}$ after constructing an inference sequence $(\mathcal{E}, \emptyset) \vdash_{RI} \cdots \vdash_{RI} (\emptyset, \mathcal{H})$. In this paper, we drop termination of $\mathcal{R} \cup \mathcal{H} \cup \{s \to t\ [\varphi]\}$ from the side condition of EXPANSION. Due to this relaxation, a constructed inference sequence does not always ensure that $\mathcal{E}$ is a set of inductive theorems of $\mathcal{R}$. For this reason, we introduce the notion of *valid inference sequences*. An inference sequence $(\mathcal{E}, \emptyset) \vdash_{RI} \cdots \vdash_{RI} (\emptyset, \mathcal{H})$ is called *valid* if $\mathcal{R} \cup \mathcal{H}$ is terminating.

Theorem 4.1 ([8]) *Let $\mathcal{R}$ be an LCTRS and $\mathcal{E}$ a finite set of equations. If we have a valid inference sequence $(\mathcal{E}, \emptyset) \vdash_{RI} \cdots \vdash_{RI} (\emptyset, \mathcal{H})$, then every equation in $\mathcal{E}$ is an inductive theorem of $\mathcal{R}$.*

5 Transforming a Proof Tableau into an Inference Sequence of RI

In this section, using the proof tableau P_{tab}, we illustrate a construction of an inference sequence of RI.

To verify the postcondition after the execution of statements, we prepare the following rules with a new symbol check with sort *state* $\Rightarrow$ *bool*:

$$\mathcal{R}_{check} = \left\{ \begin{array}{ll} \mathsf{check}(\mathsf{end}(x,i,z)) \to \mathsf{true} & [\ \ z = \frac{1}{2}x(x+1)\ \ ] \\ \mathsf{check}(\mathsf{end}(x,i,z)) \to \mathsf{false} & [\neg(z = \frac{1}{2}x(x+1))] \end{array} \right\}$$

We let $\mathcal{R}_1 = \mathcal{R}_{sum} \cup \mathcal{R}_{check}$. To prove the Hoare triple $\{x \geq 0\}\ P_{sum}\ \{z = \frac{1}{2}x(x+1)\}$, it suffices to consider initial states satisfying the precondition $x \geq 0$, and thus, we prove the following equation an inductive theorem of $\mathcal{R}_1$:

$$\text{(A1)} \quad \mathsf{check}(\mathsf{state}_1(x,i,z)) \approx \mathsf{true}\ [x \geq 0]$$

It is clear that $\mathcal{R}_1$ is quasi-reductive and $\mathcal{R}_{check}$ is terminating. Since any term with sort *state* or *bool* does not appear in $\mathcal{R}_{sum}$ as a proper subterm, $\mathcal{R}_{check}$ does not arise non-termination with $\mathcal{R}_{sum}$. As described before, $\mathcal{R}_{sum}$ is terminating and hence $\mathcal{R}_1$ is so.

From now on, we transform the proof tableau P_{tab} into an inference sequence of RI for $\mathcal{R}_1$ *from top to bottom*. The construction is independent of termination of $\mathcal{R}_1$ with generated rules, and thus the construction itself does not ensure validity of the resulting inference sequence.

We start with the initial process $(\{\ \text{(A1)}\ \}, \emptyset)$. Line A2 of P_{tab} is an assertion $@\, x \geq 0 \wedge 0 = 0$ and the validity of $x \geq 0 \implies x \geq 0 \wedge 0 = 0$ is guaranteed by P_{tab}. Using the validity, we can generalize (A1) by applying GENERALIZATION to the above process:

$$(\{\ \text{(A2)} \quad \mathsf{check}(\mathsf{state}_1(x,i,z)) \approx \mathsf{true}\ [x \geq 0 \wedge 0 = 0]\}, \emptyset)$$

Let us recall the inference rule of assignment in Hoare logic (Figure 2). For an assignment $x_k := e$ on line j, a rewrite rule $\mathsf{state}_j(\vec{x}) \to \mathsf{state}_{j+1}(x_1, \ldots, x_{k-1}, e, x_{k+1}, \ldots, x_n)$ is generated, and thus, we have the derivation $\mathsf{state}_j(\vec{x})\,[\varphi\{x_k \mapsto e\}] \to_{\mathcal{R}} \mathsf{state}_{j+1}(\vec{x})\,[\varphi]$ because $\mathsf{state}_j(\vec{x})\,[\varphi\{x_k \mapsto e\}] \to_{\mathtt{base}} \mathsf{state}_{j+1}(x_1, \ldots, x_{k-1}, e, x_{k+1}, \ldots, x_n)\,[\varphi\{x_k \mapsto e\}] \sim \mathsf{state}_{j+1}(\vec{x})\,[\varphi]$. Line 1 of P_{tab} is an assignment $i := 0$, and hence, $\mathsf{state}_1(x,i,z)\,[x \geq 0 \wedge 0 = 0] \to_{\mathcal{R}_1} \mathsf{state}_2(x,i,z)\,[x \geq 0 \wedge i = 0]$. Thus, we can simplify (A2) by applying SIMPLIFICATION to the above process:

$$(\{\ \text{(A3)} \quad \mathsf{check}(\mathsf{state}_2(x,i,z)) \approx \mathsf{true}\ [x \geq 0 \wedge i = 0]\}, \emptyset)$$

Line A4 of P_{tab} is $@\, x \geq 0 \wedge i = 0 \wedge 0 = 0$ and we can generalize (A3) by applying GENERALIZATION:

$$(\{\ \text{(A4)} \quad \mathsf{check}(\mathsf{state}_2(x,i,z)) \approx \mathsf{true}\ [x \geq 0 \wedge i = 0 \wedge 0 = 0]\}, \emptyset)$$

Line 2 of P_{tab} is an assignment $z := 0$, and we can simplify (A4) by applying SIMPLIFICATION:

$$(\{\ \text{(A5)} \quad \mathsf{check}(\mathsf{state}_3(x,i,z)) \approx \mathsf{true}\ [x \geq 0 \wedge i = 0 \wedge z = 0]\}, \emptyset)$$

Line A6 of P_{tab} is $@\, z = \frac{1}{2}i(i+1) \wedge x \geq i$ and we can generalize (A5) by applying GENERALIZATION:

$$\left(\left\{\ \text{(A6)}\quad \mathsf{check}(\mathsf{state}_3(x,i,z)) \approx \mathsf{true}\ [z = \tfrac{1}{2}i(i+1) \wedge x \geq i]\right\}, \emptyset\right)$$

Line 3 of P_{tab} is a "while" statement. At this point, we have two branches: the one entering the loop (i.e., executing the body of the loop) and the other exiting the loop. For the case analysis, we apply EXPANSION to (A6), getting the following two equations and one oriented equation:

$$\left(\left\{\begin{array}{lll} \text{(A7)} & \mathsf{check}(\mathsf{state}_4(x,i,z)) \approx \mathsf{true} & [\ \ z = \frac{1}{2}i(i+1) \wedge x \geq i \wedge x > i\ \ ] \\ \text{(A11)} & \mathsf{check}(\mathsf{end}(x,i,z)) \approx \mathsf{true} & [z = \frac{1}{2}i(i+1) \wedge x \geq i \wedge \neg(x > i)] \end{array}\right\}, \{\ \text{(A6)}\ \}\right)$$

where (A6) is oriented from left to right. The first equation represents the case where the loop body is executed, and the second one represents the case where we exit from the loop.

Line A8 of P_{tab} is an assertion and we can generalize (A7) by applying GENERALIZATION:

$$\left(\left\{\ \text{(A8)}\quad \mathsf{check}(\mathsf{state}_4(x,i,z)) \approx \mathsf{true}\quad [z+i+1 = \tfrac{1}{2}(i+1)(i+2) \wedge x \geq i+1],\ \text{(A11)}\ \right\}, \{\ \text{(A6)}\ \}\right)$$

Line 4 of P_{tab} is an assignment $z := z+i+1$ and we can simplify (A8) by applying SIMPLIFICATION:

$$\left(\left\{\ \text{(A9)}\quad \mathsf{check}(\mathsf{state}_5(x,i,z)) \approx \mathsf{true}\quad [z = \tfrac{1}{2}(i+1)(i+2) \wedge x \geq i+1],\ \text{(A11)}\ \right\}, \{\ \text{(A6)}\ \}\right)$$

Line 5 of P_{tab} is an assignment $i := i+1$ and we can simplify (A9) by applying SIMPLIFICATION:

$$\left(\left\{\ \text{(A10)}\quad \mathsf{check}(\mathsf{state}_6(x,i,z)) \approx \mathsf{true}\quad [z = \tfrac{1}{2}i(i+1) \wedge x \geq i],\ \text{(A11)}\ \right\}, \{\ \text{(A6)}\ \}\right)$$

Line 6 of P_{tab} is the end of the loop and we can apply the rule $\mathsf{state}_6(x,i,z) \to \mathsf{state}_3(x,i,z)$ that makes the left-hand side of (A10) go back to the beginning of the loop. Thus, we can simplify (A11):

$$\left(\left\{\ \text{(B1)}\quad \mathsf{check}(\mathsf{state}_3(x,i,z)) \approx \mathsf{true}\quad [z = \tfrac{1}{2}i(i+1) \wedge x \geq i],\ \text{(A11)}\ \right\}, \{\ \text{(A6)}\ \}\right)$$

The equation (B1) means that we reach the beginning of the loop after the one execution of the body. Moreover, (B1) is the same as (A6) due to the loop invariant, and hence the induction hypothesis (A6) is applicable to (B1). Thus, we can simplify (B1) by applying SIMPLIFICATION to the above process with the rule $\mathsf{check}(\mathsf{state}_3(x,i,z)) \to \mathsf{true}\ [z = \frac{1}{2}i(i+1) \wedge x \geq i]$:

$$\left(\left\{\ \text{(B2)}\quad \mathsf{true} \approx \mathsf{true}\quad [z = \tfrac{1}{2}i(i+1) \wedge x \geq i],\ \text{(A11)}\ \right\}, \{\ \text{(A6)}\ \}\right)$$

The both sides of (B2) are equivalent and we can delete (B2) by applying DELETION:

$$\left(\left\{\ \text{(A11)}\quad \mathsf{check}(\mathsf{end}(x,i,z)) \approx \mathsf{true}\quad [z = \tfrac{1}{2}i(i+1) \wedge x \geq i \wedge \neg(x > i)]\ \right\}, \{\ \text{(A6)}\ \}\right)$$

The remaining equation (A11) represents the state after exiting the loop. The last line of P_{tab} is an assertion corresponding to the postcondition. Due to the validity of $z = \frac{1}{2}i(i+1) \wedge x \geq i \wedge \neg(x > i) \implies z = \frac{1}{2}x(x+1)$, we can generalize (A11) by applying GENERALIZATION:

$$\left(\left\{\ \text{(B3)}\quad \mathsf{check}(\mathsf{end}(x,i,z)) \approx \mathsf{true}\quad [z = \tfrac{1}{2}x(x+1)]\ \right\}, \{\ \text{(A6)}\ \}\right)$$

The constraints of (B3) and the postcondition of P_{tab} are equivalent and we can apply the first rule of $\mathcal{R}_{check}$ to verify the postcondition. Thus, we can simplify (B3) by applying SIMPLIFICATION to the above process with the rule of $\mathsf{check}(\mathsf{end}(x,i,z)) \to \mathsf{true}\ [z = \frac{1}{2}x(x+1)]$:

$$\left(\left\{\ \text{(B4)}\quad \mathsf{true} \approx \mathsf{true}\quad [z = \tfrac{1}{2}x(x+1)]\ \right\}, \{\ \text{(A6)}\ \}\right)$$

The both sides of (B4) are equivalent and we can delete (B4) by applying DELETION:

$$(\emptyset, \{\ (\mathrm{A6})\ \})$$

In this way, we constructed an inference sequence of RI. In the above illustration, we did not show the case of "if" statements. However, the missing case is a simpler one of "while" statements, where we use EXPANSION without orienting equations. For page limitation, we do not formalize the above construction, but the above illustration is almost formal because it does not depend on the detail of the example. For this reason, the formalization would be straightforward.

Next, we show that the constructed inference sequence above is valid. To this end, it suffices to show that $\mathcal{R}_1 \cup \{\ (\mathrm{A6})\ \}$ is terminating. Since the right-hand sides of oriented equations in $\mathcal{H}$ are always true, $\mathcal{H}$ is always terminating and does not arise non-termination of the original LCTRS $\mathcal{R} \cup \mathcal{R}_{check}$. This means that if $\mathcal{R}$ is terminating, then so is $\mathcal{R} \cup \mathcal{R}_{check} \cup \mathcal{H}$. As described before, $\mathcal{R}_1$ is terminating, and hence, $\mathcal{R}_1 \cup \{\ (\mathrm{A6})\ \}$ is terminating. Therefore, the constructed inference sequence is valid, and the equation (A1) is an inductive theorem of $\mathcal{R}_1$. This observation implies that given a proof tableau P, if $\mathcal{R}$ obtained from P is terminating, then we can construct a valid inference sequence of RI from P and $\mathcal{R}$.

Since $\mathcal{R}_1$ is non-overlapping and thus confluent, every ground instance of $\mathsf{check}(\mathsf{state}_1(x,i,z))\,[x \geq 0]$ reduces to true. As the proof tableau shows, this means that for any initial state satisfying the precondition $x \geq 0$, the final state of the terminating execution of P_{sum} satisfies the postcondition $z = \frac{1}{2}x(x+1)$.

6 Discussion

In the previous section, we transformed a proof tableau for *partial correctness* into a valid inference sequence of RI. It would be possible to transform a proof tableaux for *total correctness*, which includes *ranking functions* in loop invariants, into an inference sequence of RI. It is, however, not clear how to use ranking functions to prove termination of the corresponding LCTRS. On the other hand, to prove validity of the converted inference sequence of RI, we can use techniques for proving termination of LCTRSs, which are based on techniques developed well for term rewriting. The transformation of proof tableaux for partial correctness into inference sequences of RI enables us to use such techniques instead of finding appropriate ranking functions for all loops in given programs. The use of techniques to prove termination is one of the advantages of the transformation.

From the idea of the transformation, we may apply RI to the initial equation such as (A1) instead of constructing a proof tableau for a given Hoare triple. Unfortunately, Ctrl [13], an RI tool for LCTRSs, did not succeed in automatically proving (A1) an inductive theorem of $\mathcal{R}_1$. It is worth improving tools for RI so as to directly prove (A1) an inductive theorem of $\mathcal{R}_1$.

As future work, we will transform some inference sequences of RI into proof tableaux of Hoare logic in order to compare RI with Hoare logic. For inference sequences of RI, we sometimes need a lemma equation that is helpful to use induction, but it is not easy to find an appropriate lemma equation. For this reason, we expect the transformation between proof tableaux of Hoare logic and inference sequences of RI to help us to develop and improve a technique for lemma generation.

Acknowledgement We thank the anonymous reviewers for their useful comments to improve this paper, and to encourage us to continue this work.

References

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

[2] Adel Bouhoula & Florent Jacquemard (2008): *Automated Induction with Constrained Tree Automata*. In: *Proc. IJCAR 2008*, *Lecture Notes in Computer Science* 5195, Springer, pp. 539–554, doi:10.1007/978-3-540-71070-7_44.

[3] Aaron R. Bradley & Zohar Manna (2007): *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, doi:10.1007/978-3-540-74113-8.

[4] Stephan Falke & Deepak Kapur (2008): *Dependency Pairs for Rewriting with Built-In Numbers and Semantic Data Structures*. In: *Proc. RTA 2008*, *Lecture Notes in Computer Science* 5117, Springer, pp. 94–109, doi:10.1007/978-3-540-70590-1_7.

[5] Stephan Falke & Deepak Kapur (2009): *A Term Rewriting Approach to the Automated Termination Analysis of Imperative Programs*. In: *Proc. CADE 2009*, *Lecture Notes in Computer Science* 5663, Springer, pp. 277–293, doi:10.1007/978-3-642-02959-2_22.

[6] Stephan Falke & Deepak Kapur (2012): *Rewriting Induction + Linear Arithmetic = Decision Procedure*. In: *Proc. IJCAR 2012*, *Lecture Notes in Computer Science* 7364, Springer, pp. 241–255, doi:10.1007/978-3-642-31365-3_20.

[7] Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp & Stephan Falke (2009): *Proving Termination of Integer Term Rewriting*. In: *Proc. RTA 2009*, *Lecture Notes in Computer Science* 5595, Springer, pp. 32–47, doi:10.1007/978-3-642-02348-4_3.

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

[9] 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 Trans. Program.* 1(2), pp. 100–121. In Japanese (a translated summary is available from `http://www.trs.css.i.nagoya-u.ac.jp/crisys/`).

[10] Gérard Huet & Jean-Marie Hullot (1982): *Proof by Induction in Equational Theories with Constructors*. *J. Comput. Syst. Sci.* 25(2), pp. 239–266, doi:10.1016/0022-0000(82)90006-X.

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

[12] Cynthia Kop & Naoki Nishida (2014): *Automatic Constrained Rewriting Induction towards Verifying Procedural Programs*. In: *Proc. APLAS 2014*, *Lecture Notes in Computer Science* 8858, Springer, pp. 334–353, doi:10.1007/978-3-319-12736-1_18.

[13] Cynthia Kop & Naoki Nishida (2015): *Constrained Term Rewriting tooL*. In: *Proc. LPAR-20*, *Lecture Notes in Computer Science* 9450, Springer, pp. 549–557, doi:10.1007/978-3-662-48899-7_38.

[14] David R. Musser (1980): *On Proving Inductive Properties of Abstract Data Types*. In: *Proc. POPL 1980*, pp. 154–162, doi:10.1145/567446.567461.

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

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

[17] John C. Reynolds (1998): *Theories of Programming Languages*. Cambridge University Press, doi:10.1017/CBO9780511626364.

[18] Tsubasa Sakata, Naoki Nishida & Toshiki Sakabe (2011): *On Proving Termination of Constrained Term Rewrite Systems by Eliminating Edges from Dependency Graphs*. In: *Proc. WFLP 2011*, *Lecture Notes in Computer Science* 6816, Springer, pp. 138–155, doi:10.1007/978-3-642-22531-4_9.

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

[20] Germán Vidal (2012): *Closed Symbolic Execution for Verifying Program Termination*. In: *Proc. SCAM 2012*, IEEE Computer Society, pp. 34–43, doi:10.1109/SCAM.2012.13.