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, we show that a proof tableau for partial correctness can be transformed into an inference sequence of rewriting induction for constrained rewriting. We also show that the resulting sequence is a valid proof for an inductive theorem corresponding to the Hoare triple if the constrained rewriting system obtained from the program is terminating. Such a valid proof with termination of the constrained rewriting system implies total correctness of the program w.r.t. the Hoare triple. 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.

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) [17], both of which are called *implicit induction principles*. Frameworks based on the RI principle (RI frameworks, for short) consist of inference rules to prove that given equations are inductive theorems. On the other hand, 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, 19], conditional and constrained TRSs [2], $\mathbb{Z}$-TRSs [6], and logically constrained TRSs [11]) have been developed [2, 20, 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, 21, 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 computation 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. Some RI frameworks succeed in proving equivalence of an imperative program and its functional specification such that a proof based on Hoare logic needs a loop invariant (cf. [8]). From such experiences, we are interested in differences between RI frameworks and other verification methods.

In this paper, we show that a *proof tableau* of Hoare logic can be transformed 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. We also show that the resulting inference sequence is a valid proof for an inductive theorem

H. Cirstea, D. Sabel (Eds.): Fourth International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE 2017)
EPTCS 265, 2018, pp. 35–51, doi:10.4204/EPTCS.265.4

corresponding to the Hoare triple for the proof tableau if the LCTRS obtained from the program is terminating.

Given a *while* program P and a proof tableau T_P of a Hoare triple $\{\varphi_P\}\ P\ \{\psi_P\}$ for *partial correctness*, we proceed as follows:

1. We transform P into an equivalent LCTRS $\mathcal{R}_P$, and we prove termination of the LCTRS $\mathcal{R}_P$.
2. We prepare rewrite rules $\mathcal{R}_{check}$ to verify the post-condition ψ_P in the proof tableau.
3. We prepare a constrained equation e_P corresponding to the Hoare triple $\{\varphi_P\}\ P\ \{\psi_P\}$.
4. Starting with the equation e_P, we transform the proof tableau into an inference sequence $(\{e_P\},\emptyset) \vdash_{RI} \cdots \vdash_{RI} (\emptyset,\mathcal{H})$ of RI in a top-down fashion, where we do not prove termination in constructing the inference sequence of RI.

In addition to the above transformation, we show that termination of the LCTRS $\mathcal{R}_P$ implies termination of the LCTRS $\mathcal{R}_P \cup \mathcal{R}_{check} \cup \mathcal{H}$. Termination of the LCTRS $\mathcal{R}_P \cup \mathcal{R}_{check} \cup \mathcal{H}$ ensures that the resulting inference sequence $(\{e_P\},\emptyset) \vdash_{RI} \cdots \vdash_{RI} (\emptyset,\mathcal{H})$ is a valid proof of RI—the equation e_P is an inductive theorem of the LCTRS $\mathcal{R}_P$—and thus, the *while* program P is *totally correct* w.r.t. φ_P and ψ_P.

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

This paper is organized as the follows. In Section 2, we briefly recall LCTRSs, *while* programs, and a conversion of *while* programs to LCTRSs. In Section 3, we recall proof tableaux of Hoare logic, and in Section 4, we recall the framework of rewriting induction for LCTRSs. In Section 5, we show that a proof tableau can be transformed into an inference sequence of RI, and the resulting inference sequence is a valid proof for total correctness if the LCTRS obtained from the proof tableau is terminating. In Section 6, we conclude this paper and describe future direction of this research.

2 Preliminaries

In this section, we recall LCTRSs, 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, 16] 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*—function symbols $a : \iota$ such that $\mathcal{J}$ gives a bijective mapping from $\mathcal{V}al_\iota$ to $\mathcal{I}_\iota$—for each sort ι occurring in Σ_{theory}. We require that $\Sigma_{terms} \cap \Sigma_{theory} \subseteq \mathcal{V}al = \bigcup_{\iota \in \mathcal{S}} \mathcal{V}al_\iota$. The sorts occurring in Σ_{theory} are called *theory sorts*, and the symbols *theory symbols*. Symbols in $\Sigma_{theory} \setminus \mathcal{V}al$ are *calculation symbols*. A term in $T(\Sigma_{theory}, \mathcal{V})$ is called a *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 $[\![s]\!] = [\![c]\!]$. We use infix notation for theory and calculation symbols.

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 $\mathsf{true}, \mathsf{false} : bool$, $\wedge, \vee, \Longrightarrow : bool \times bool \Rightarrow bool$, $\neg : bool \Rightarrow bool$, and, for all theory sorts ι, symbols $=_\iota, \neq_\iota : \iota \times \iota \Rightarrow bool$, and an evaluation function $\mathcal{J}$ that interprets these symbols as expected. We omit the sort subscripts from $=$ and $\neq$ when clear from context.

The standard integer signature Σ^{int}_{theory} is $\Sigma^{core}_{theory} \cup \{+, -, *, \mathsf{exp}, \mathsf{div}, \mathsf{mod} : int \times int \Rightarrow int\} \cup \{\geq, > : int \times int \Rightarrow bool\} \cup \{\mathsf{n} : int \mid n \in \mathbb{Z}\}$ with values true, false, and n for all integers $n \in \mathbb{Z}$. Thus, we use n (in sans-serif font) as the function symbol for $n \in \mathbb{Z}$ (in *math* font). We define $\mathcal{J}$ in the natural way, except: since all $\mathcal{J}(f)$ must be total functions, we set $\mathcal{J}(\mathsf{div})(n, 0) = \mathcal{J}(\mathsf{mod})(n, 0) = \mathcal{J}(\mathsf{exp})(n, k) = 0$ for all n and all $k < 0$. When constructing LCTRSs from, e.g., *while* programs, we can add explicit error checks for, e.g., "division by zero", to constraints (cf. [8]).

A *constrained rewrite rule* is a triple $\ell \to r\ [\varphi]$ such that ℓ and r are terms of the same sort, φ is a constraint, and ℓ has the form $f(\ell_1, \ldots, \ell_n)$ and contains at least one symbol in $\Sigma_{terms} \setminus \Sigma_{theory}$ (i.e., ℓ is not a 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 $\{f(x_1, \ldots, x_n) \to y\ [y = f(x_1, \ldots, x_n)] \mid f : \iota_1 \times \cdots \times \iota_n \Rightarrow \iota \in \Sigma_{theory} \setminus \mathcal{V}al\}$. 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. An LCTRS $\mathcal{R}$ is said to be *left-linear* if for every rule in $\mathcal{R}$, the left-hand side is linear. $\mathcal{R}$ is said to be *non-overlapping* if for every term s and rule $\ell \to r\ [\varphi]$ such that s reduces with $\ell \to r\ [\varphi]$ at the root position: (a) there are no other rules $\ell' \to r'\ [\varphi']$ such that s reduces with $\ell' \to r'\ [\varphi']$ at the root position, and (b) if s reduces with any rule at a non-root position q, then q is not a position of ℓ. $\mathcal{R}$ is said to be *orthogonal* if $\mathcal{R}$ is left-linear and non-overlapping. For $f(\ell_1, \ldots, \ell_n) \to r\ [\varphi] \in \mathcal{R}$, we call f a *defined symbol* of $\mathcal{R}$, and non-defined elements of Σ_{terms} and all values are called *constructors* of $\mathcal{R}$. Let $\mathcal{D}_{\mathcal{R}}$ be the set of all defined symbols and $\mathcal{C}_{\mathcal{R}}$ the set of constructors. A term in $T(\mathcal{C}_{\mathcal{R}}, \mathcal{V})$ is a *constructor term* of $\mathcal{R}$.

Example 2.1 ([8]) Let $\mathcal{S} = \{int, bool\}$, and $\Sigma = \Sigma_{terms} \cup \Sigma^{int}_{theory}$, where $\Sigma_{terms} = \{\ \mathsf{fact} : int \Rightarrow int\ \} \cup \{\ \mathsf{n} : int \mid n \in \mathbb{Z}\ \}$. Then both *int* and *bool* are theory sorts. We also define set and function interpretations,

i.e., $\mathcal{I}_{int} = \mathbb{Z}$, $\mathcal{I}_{bool} = \mathbb{B}$, and $\mathcal{J}$ is defined as above. Examples of 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., [18]).

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 $\mathbf{while} \mathbin{@} \zeta\,(\psi)\{c\}$ with ζ 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:

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

where $n \in \mathbb{Z}$, $v \in \mathcal{V}$, and we may omit brackets in the usual way. We use $\neq$, $\leq$, $>$, $\geq$, $\wedge$, $\implies$, etc, as syntactic sugars. We abbreviate $\mathbf{while} \mathbin{@} \mathsf{true}\,(\psi)\{c\}$ to $\mathbf{while}(\psi)\{c\}$. 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 $\mathcal{V}ar(P)$. Given an assignment θ for $\mathcal{V}ar(P)$, we write $\theta \Rightarrow_P \theta'$ if the execution of P starts with θ and halts with an assignment θ'. We abuse assignments for variables as substitutions for terms.

Example 2.2 The following, denoted by P_{sum}, is a *while* program 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 such that $\mathcal{V}ar(P) = \{x_1, \ldots, x_n\}$ and P has m lines without any assertion. We denote the sequence "$x_1, \ldots, x_n$" by $\vec{x}$. We prepare *state*, a sort 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 function symbols $\mathsf{state}_1, \ldots, \mathsf{state}_m$ with sort $\overbrace{\mathbb{Z} \times \cdots \times \mathbb{Z}}^{n} \Rightarrow \mathit{state}$. Instances of $\mathsf{state}_1, \ldots, \mathsf{state}_m$ represent *states* in executing P. Here, 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 stores i and $v_1, \ldots, v_n$ are assigned to $x_1, \ldots, x_n$, resp. For each statement in P, we generate constrained rewrite rules for $\mathsf{state}_1, \ldots, \mathsf{state}_m$ as follows:

- an assignment $\boxed{i \quad x_k := e;}$ is converted to the following rule:

$$\{\ \mathsf{state}_i(\vec{x}) \to \mathsf{state}_{i+1}(x_1, \ldots, x_{k-1}, e, x_{k+1}, \ldots, x_n)\ \}$$

- a "skip" statement $\boxed{i \quad \textbf{skip};}$ is converted to the following rule:

$$\{\ \mathsf{state}_i(\vec{x}) \to \mathsf{state}_{i+1}(\vec{x})\ \}$$

- an "if" statement $\boxed{\begin{array}{ll} i & \textbf{if}(\varphi)\{ \\ \vdots & \quad \cdots \\ j & \}\textbf{else}\{ \\ \vdots & \quad \cdots \\ k & \} \end{array}}$ is converted to the following rules:

$$\left\{ \begin{array}{lll} \mathsf{state}_i(\vec{x}) \to \mathsf{state}_{i+1}(\vec{x}) & [\ \varphi\ ] & \mathsf{state}_j(\vec{x}) \to \mathsf{state}_{k+1}(\vec{x}) \\ \mathsf{state}_i(\vec{x}) \to \mathsf{state}_{j+1}(\vec{x}) & [\neg\varphi] & \mathsf{state}_k(\vec{x}) \to \mathsf{state}_{k+1}(\vec{x}) \end{array} \right\}$$

- a "while" statement $\boxed{\begin{array}{ll} i & \textbf{while}\,@\,\zeta\,(\varphi)\{ \\ \vdots & \quad \cdots \\ j & \} \end{array}}$ is converted to the following rules:

$$\left\{ \begin{array}{lll} \mathsf{state}_i(\vec{x}) \to \mathsf{state}_{i+1}(\vec{x}) & [\ \varphi\ ] & \mathsf{state}_j(\vec{x}) \to \mathsf{state}_i(\vec{x}) \\ \mathsf{state}_i(\vec{x}) \to \mathsf{state}_{j+1}(\vec{x}) & [\neg\varphi] & \end{array} \right\}$$

For brevity, we replace state_m in the final result by end. By definition, it is clear that any LCTRS obtained from a *while* program by the above conversion is orthogonal.

Example 2.3 The program P_{sum} in Example 2.2 is converted to the following LCTRS:

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

$\mathcal{R}_{sum}$ is orthogonal (and thus, 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 e.g., Ctrl [13].

Theorem 2.4 ([9]) *Let $\mathcal{R}_P$ be the LCTRS obtained from P by the conversion in this section. For all assignments θ, θ' (for $\mathcal{V}ar(P)$), $\theta \Rightarrow_P \theta'$ iff $\mathsf{state}_1(\vec{x})\theta \to^*_{\mathcal{R}_P} \mathsf{end}(\vec{x})\theta'$.*

Note that the execution of P starting with θ does not halt iff $\mathsf{state}_1(\vec{x})\theta$ does not terminate on $\mathcal{R}_P$. It follows from Theorem 2.4 that if $\mathcal{R}_P$ is terminating, then any execution of P halts. On the other hand, the converse does not hold for all *while* programs, i.e., the conversion above does not preserve termination of P (see, e.g., [15]).[1]

3 Proof Tableaux of Hoare Logic

Hoare logic is a logic to prove a Hoare triple to hold (see e.g., [18]). A triple $\{\varphi\}\ P\ \{\psi\}$ for partial correctness is said to *hold* (or P is *partially correct* w.r.t. pre- and post-conditions φ, ψ) if for any initial state satisfying φ, the final state of the execution satisfies ψ whenever the execution from the initial state halts. A triple $[\varphi]\,P\,[\psi]$ for total correctness is said to *hold* (or P is *totally correct* w.r.t. pre- and post-conditions φ, ψ) if for any initial state satisfying φ, the execution from the initial state halts and the final state of the execution satisfies ψ. Note that total correctness is equivalent to partial correctness with termination of the program under the pre-condition.

In this section, we formalize proof tableaux of Hoare triples. The aim of this paper is to transform a proof tableau of a Hoare triple for partial correctness into an inference sequence of RI (shown in Section 4). For this reason, we consider proof tableaux for partial correctness and we do not focus on the construction of proof tableaux.

In the following, we consider *while* programs as sequences of commands connected by ";", and we write P as $C_1;C_2;\ldots;C_n$. Note that we consider ";" to implicitly exist at the end of "if" and "while" statements. Bodies of "if" and "while" statements are also considered sequences of commands.

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 last elements of the sequence are annotations, e.g., P is of the form $@\,\varphi;C_1;\ldots;C_n;@\,\psi$ ($n > 0$),*
- *for each subsequence $@\,\varphi;@\,\psi$ of annotations, the formula $\varphi \implies \psi$ is valid, and*
- *for each subsequence $C_1;C_2;C_3$, if C_2 is not an annotation, then the first and third elements 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 $\mathbf{if}(\psi)\{S'\}\mathbf{else}\{S''\}$ and C_1 is of the form $@\,\varphi$, then 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 both S' and S'' are equivalent, i.e., $C_1;C_2;C_3$ is of the form*

$$@\varphi;\ \mathbf{if}(\psi)\{@\varphi \wedge \psi;\ldots;\ @\xi\}\mathbf{else}\{@\varphi \wedge \neg\psi;\ldots;\ @\xi\};\ @\xi$$

and

[1] When replacing $x > i$ in P_{sum} by $x \neq i$, the constructed LCTRS $\mathcal{R}'_{sum}$ is the one obtained from $\mathcal{R}_{sum}$ by replacing $x > i$ by $x \neq i$. LCTRS $\mathcal{R}'_{sum}$ is not terminating because we have an infinite reduction sequence from, e.g., $\mathsf{state}_3(0,1,0)$.

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

$$\frac{}{\{\varphi\}\ \mathbf{skip}\ \{\varphi\}} \qquad \frac{\{\varphi\}\ C_1\ \{\xi\} \quad \{\xi\}\ C_2\ \{\psi\}}{\{\varphi\}\ C_1; C_2\ \{\psi\}}$$

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

Figure 1: basic inference rules of Hoare logic.

A1	$@\ x \geq 0;$	A8	$@\ z+i+1 = \frac{1}{2}(i+1)(i+2) \wedge x \geq i+1;$
A2	$@\ x \geq 0 \wedge 0 = 0;$	4	$z := z+i+1;$
1	$i := 0;$	A9	$@\ z = \frac{1}{2}(i+1)(i+2) \wedge x \geq i+1;$
A3	$@\ x \geq 0 \wedge i = 0;$	5	$i := i+1;$
A4	$@\ x \geq 0 \wedge i = 0 \wedge 0 = 0;$	A10	$@\ z = \frac{1}{2}i(i+1) \wedge x \geq i;$
2	$z := 0;$	6	$\}$
A5	$@\ x \geq 0 \wedge i = 0 \wedge z = 0;$	A11	$@\ z = \frac{1}{2}i(i+1) \wedge x \geq i \wedge \neg(x > i);$
A6	$@\ z = \frac{1}{2}i(i+1) \wedge x \geq i;$	A12	$@\ z = \frac{1}{2}x(x+1);$
3	$\mathbf{while}\ @z = \frac{1}{2}i(i+1) \wedge x \geq i\ (x > i)\{$	7	
A7	$\quad @\ z = \frac{1}{2}i(i+1) \wedge x \geq i \wedge x > i;$		

Figure 2: an annotated *while* program T_{sum} for P_{sum}.

– *if C_2 is of the form* $\mathbf{while}\ @\zeta\,(\varphi)\{S\}$, *then C_1 and the last element of the sequence S are $@\zeta$, and the head element of S is $@\,\zeta \wedge \varphi$, and C_3 is $@\,\zeta \wedge \neg\varphi$, i.e., $C_1;C_2;C_3$ is of the form*

$$@\zeta;\ \mathbf{while}\ @\zeta\,(\varphi)\{@\zeta \wedge \varphi; \ldots; @\zeta\};\ @\zeta \wedge \neg\varphi.$$

Note that a proof tableau is a tableau representation of an inference tree constructed by basic inference rules of Hoare logic illustrated in Figure 1 (see e.g., [18]).

Example 3.2 The annotated *while* program of Figure 2, denoted by T_{sum}, 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].

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 $s \approx t\ [\varphi]$.

As in [8], we restrict LCTRSs to be terminating and quasi-reductive. An RI-based method is to construct an *inference sequence* by applying the following basic inference rules to pairs $(\mathcal{E}, \mathcal{H})$ of finite sets $\mathcal{E}$ and $\mathcal{H}$ of constrained equations and rewrite rules, resp.:

EXPANSION

$$(\mathcal{E} \uplus \{s \simeq t\ [\varphi]\}, \mathcal{H}) \vdash_{RI} (\mathcal{E} \cup \mathit{Expd}_{\mathcal{R}}(s \approx t\ [\varphi], p), \mathcal{H} \cup \{s \to t\ [\varphi]\})$$

where

- p is a *basic* position of s,[2]
- $\mathcal{R} \cup \mathcal{H} \cup \{s \to t\ [\varphi]\}$ is terminating, and
- $\mathit{Expd}_{\mathcal{R}}(s \approx t\ [\varphi], p)$ is the set of constrained equation $s' \approx t'\ [\varphi']$ such that $s\gamma \approx t\gamma\ [\varphi\gamma \wedge \psi\gamma] \to_{1.p,\ell \to r\ [\psi]} s' \approx t'\ [\varphi']$ for some renamed variant $\ell \to r\ [\psi]$ of a rule in $\mathcal{R}$ (i.e., $\mathcal{V}ar(\ell, r, \psi) \cap \mathcal{V}ar(s, t, \varphi) = \emptyset$) and a most general unifier γ of $s|_p$ and ℓ.

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.

In addition to the above, we use the following inference rules:

CASESPLITTING

$$(\mathcal{E} \uplus \{s \simeq t\ [\varphi]\}, \mathcal{H}) \vdash_{RI} (\mathcal{E} \cup \mathit{Expd}_{\mathcal{R}}(s \approx t\ [\varphi], p), \mathcal{H})$$

where p is a *basic* position of s. Note that CASESPLITTING is a variant of EXPANSION without adding $s \to t\ [\varphi]$ to $\mathcal{H}$.

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 $(\emptyset, \mathcal{H})$, then 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 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 $\mathcal{R}$.*

[2] 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.

5 Transforming a Proof Tableau into an Inference Sequence of RI

In this section, using the proof tableau T_{sum}, we first illustrate a transformation of a proof tableau into an inference sequence of RI, and then formalize the transformation.

5.1 Overview

Let us recall the LCTRS $\mathcal{R}_{sum}$ in Example 2.2 and the proof tableau T_{sum} in Figure 2. To verify the post-condition after the execution of P_{sum}, we prepare the following rules with a new symbol $\mathsf{chk} : \mathit{state} \Rightarrow \mathit{bool}$:

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

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

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

It is clear that $\mathcal{R}_1$ is quasi-reductive.

From now on, we transform the proof tableau T_{sum} into an inference sequence of RI for $\mathcal{R}_1$ in a *top-down* fashion. 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 T_{sum} 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 the fact that T_{sum} is a proof tableau. Using the validity, we can generalize (A1) by applying GENERALIZATION to the above process:

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

Let us recall the inference rule of assignment in Hoare logic (Figure 1). 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 T_{sum} 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:

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

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

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

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

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

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

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

Line 3 of T_{sum} 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:

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

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 T_{sum} is an assertion and we can generalize (A7) by applying GENERALIZATION:

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

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

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

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

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

Line 6 of T_{sum} 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 (A10):

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

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 rule (A6) $\mathsf{chk}(\mathsf{state}_3(x,i,z)) \to \mathsf{true}\ [z = \frac{1}{2}i(i+1) \wedge x \geq i]$:

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

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

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

The remaining equation (A11) represents the state after exiting the loop. The last line of T_{sum} is an assertion corresponding to the post-condition. 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:

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

The constraints of (B3) and the post-condition of T_{sum} are equivalent and we can apply the first rule of $\mathcal{R}'_{check}$ to the left-hand side of (B3) in order to verify the post-condition. Thus, we can simplify (B3) by applying SIMPLIFICATION with rule $\mathsf{chk}(\mathsf{end}(x,i,z)) \to \mathsf{true}\ [z = \frac{1}{2}x(x+1)]$ in $\mathcal{R}'_{check}$ to the above process:

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

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

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

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 CASESPLITTING instead of EXPANSION.

Finally, we show that $\mathcal{R}_1 \cup \{\ \text{(A6)}\ \}$ is terminating. Since any term with sort *state* or *bool* does not appear in $\mathcal{R}_{sum}$ as a proper subterm, $\mathcal{R}'_{check} \cup \{\ \text{(A6)}\ \}$ does not introduce non-termination into $\mathcal{R}_{sum}$. As described before, $\mathcal{R}_{sum}$ is terminating and hence $\mathcal{R}_1 \cup \{\ \text{(A6)}\ \}$ is so.

5.2 Formalization

In this section, we formalize the idea illustrated in the previous section. In the following, we consider

- a *while* program P such that $\mathcal{V}ar(P) = \{x_1, \ldots, x_n\}$,
- a proof tableau T_P for a Hoare triple $\{\varphi_P\}\ P\ \{\psi_P\}$,[3] and
- the LCTRS $\mathcal{R}_P$ obtained from P by the conversion in Section 2.3.

We denote the sequence $x_1, \ldots, x_n$ by $\vec{x}$. Unlike previous sections, we specify line numbers for T_P, and reuse them in converting P to $\mathcal{R}_P$. For this reason, the function symbol to represent initial states is not state_1 but state_{i_0} for some $i_0 > 1$. Notice that the pre-condition φ_P is on line 1 of T_P as an assertion. For readability, we use *start* as a meta symbol that stands for state_{i_0}.

To check whether the final state of the execution of P satisfies the post-condition ψ_P, we prepare the following rules:

$$\mathcal{R}_{check} = \left\{ \begin{array}{ll} \mathsf{chk}(\mathsf{end}(\vec{x})) \to \mathsf{true} & [\ \psi_P\ ] \\ \mathsf{chk}(\mathsf{end}(\vec{x})) \to \mathsf{false} & [\neg\psi_P] \end{array} \right\}$$

where $\mathsf{chk} : \mathit{state} \Rightarrow \mathit{bool}$. Then, to verify the Hoare triple $\{\varphi_P\}\ P\ \{\psi_P\}$, we prepare the following constrained equation:

$$\mathsf{chk}(\mathit{start}(\vec{x})) \approx \mathsf{true}\ [\varphi_P]$$

In the following, we denote the above equation by e_P.

By definition, $\mathcal{R}_P \cup \mathcal{R}_{check}$ has the following properties.

Lemma 5.1 *All of the following hold:*

(a) $\mathcal{R}_P \cup \mathcal{R}_{check}$ *is orthogonal.*

(b) If $\mathcal{R}_P$ *is terminating, then* $\mathcal{R}_P \cup \mathcal{R}_{check}$ *is so.*

Proof. We first prove (a). As described in Section 2.3, $\mathcal{R}_P$ is orthogonal. By definition, $\mathcal{R}_{check}$ is orthogonal. $\mathcal{R}_{check}$ has no defined symbol of $\mathcal{R}_P$ and thus, $\mathcal{R}_{check}$ does not generate any overlap with $\mathcal{R}_P$. Therefore, $\mathcal{R}_P \cup \mathcal{R}_{check}$ is orthogonal.

Next, we prove (b). Assume that $\mathcal{R}_P$ is terminating but $\mathcal{R}_P \cup \mathcal{R}_{check}$ is not. Then, there exists an infinite reduction sequence of $\mathcal{R}_P \cup \mathcal{R}_{check}$. Due to the sort of chk, the infinite reduction sequence starts with a term of the form $\mathsf{state}_i(\vec{t})$, and any rule of $\mathcal{R}_{check}$ is not used in the reduction sequence. This means that the infinite reduction sequence is caused by $\mathcal{R}_P$. This contradicts the assumption. □

The equation e_P has the following property.

Theorem 5.2 *If* e_P *is an inductive theorem of* $\mathcal{R}_P \cup \mathcal{R}_{check}$*, then* $\{\varphi_P\}\ P\ \{\psi_P\}$ *holds.*

Proof. Let θ be an assignment for $\mathcal{V}ar(P)$. Assume that $\varphi_P\theta$ holds and the execution of P starting with θ halts with an assignment θ', i.e., $\theta \Rightarrow_P \theta'$. Then, it follows from Theorem 2.4 that $\mathit{start}(\vec{x})\theta \to^*_{\mathcal{R}_P} \mathsf{end}(\vec{x})\theta'$, and hence $\mathsf{chk}(\mathit{start}(\vec{x}))\theta \to^*_{\mathcal{R}_P} \mathsf{chk}(\mathsf{end}(\vec{x}))\theta'$. Since $\varphi_P\theta$ holds and e_P is an inductive theorem of $\mathcal{R}_P \cup \mathcal{R}_{check}$, we have that $\mathsf{chk}(\mathit{start}(\vec{x}))\theta \leftrightarrow^*_{\mathcal{R}_P \cup \mathcal{R}_{check}} \mathsf{true}$. Since $\mathcal{R}_P \cup \mathcal{R}_{check}$ is orthogonal (i.e., confluent) by Lemma 5.1 (a), we have that $\mathsf{chk}(\mathit{start}(\vec{x}))\theta \to^*_{\mathcal{R}_P \cup \mathcal{R}_{check}} \mathsf{true}$ and hence $\mathsf{chk}(\mathsf{end}(\vec{x}))\theta'$ has to reduce to true. This means that $\psi_P\theta'$ holds. Therefore, $\{\varphi_P\}\ P\ \{\psi_P\}$ holds. □

[3] Note that P is the same as the *while* program obtained from T_P by removing assertions.

Theorem 5.2 enables us to prove $\{\varphi_P\}\ P\ \{\psi_P\}$ to hold by showing that e_P is an inductive theorem of $\mathcal{R}_P \cup \mathcal{R}_{check}$. Note that the converse of Theorem 5.2 holds if P is terminating.

Next, we formalize the transformation shown in Section 5.1. We first prepare a function $Trans_1$ that takes a suffix T of proof tableau T_P and finite sets $\mathcal{E}$ and $\mathcal{H}$ of equations and rewrite rules, resp., and returns a suffix T' of T, and finite sets $\mathcal{E}'$ and $\mathcal{H}'$ of equations and rewrite rules, resp.: $Trans_1(T,\mathcal{E},\mathcal{H}) = (T',\mathcal{E}',\mathcal{H}')$. For readability, we use visualized notations for suffixes of proof tableaux, e.g.,

$$\boxed{\begin{array}{ll} i & @\varphi; \\ i+1 & @\psi; \\ \vdots & \vdots \end{array}}$$

for $@\varphi; @\psi; \ldots$ such that the first element $@\varphi$ is located on line i. We assume that any equation in $\mathcal{E}$ is of the form $\mathsf{chk}(\mathsf{state}_j(\vec{t})) \approx \mathsf{true}\ [\varphi]$ or $\mathsf{chk}(\mathsf{end}(\vec{t})) \approx \mathsf{true}\ [\varphi]$, and then we define $Trans_1$ so as to make $\mathcal{E}'$ a set of such equations. Following the definition of proof tableaux, the function $Trans_1$ is defined as follows:

- (two continuous assertions)

$$Trans_1(\boxed{\begin{array}{ll} i & @\varphi; \\ i+1 & @\psi; \\ \vdots & \vdots \end{array}}, \{\ \mathsf{chk}(\mathsf{state}_j(\vec{x})) \approx \mathsf{true}\ [\varphi]\ \} \uplus \mathcal{E}, \mathcal{H})$$
$$= (\boxed{\begin{array}{ll} i+1 & @\psi; \\ \vdots & \vdots \end{array}}, \{\ \mathsf{chk}(\mathsf{state}_j(\vec{x})) \approx \mathsf{true}\ [\psi]\ \} \cup \mathcal{E}, \mathcal{H})$$

  Note that $i > j$. This case corresponds to the application of GENERALIZATION to $(\{\ \mathsf{chk}(\mathsf{state}_j(\vec{x})) \approx \mathsf{true}\ [\psi]\ \} \uplus \mathcal{E}, \mathcal{H})$.

- (assignments)

$$Trans_1(\boxed{\begin{array}{ll} i & @\varphi; \\ i+1 & x_k := e; \\ i+2 & @\psi; \\ \vdots & \vdots \end{array}}, \{\ \mathsf{chk}(\mathsf{state}_{i+1}(\vec{x})) \approx \mathsf{true}\ [\varphi]\ \} \uplus \mathcal{E}, \mathcal{H})$$
$$= (\boxed{\begin{array}{ll} i+2 & @\psi; \\ \vdots & \vdots \end{array}}, \{\ \mathsf{chk}(\mathsf{state}_j(\vec{x})) \approx \mathsf{true}\ [\psi]\ \} \cup \mathcal{E}, \mathcal{H})$$

  where $\mathsf{state}_{i+1}(\ldots,x_k,\ldots) \to \mathsf{state}_j(\ldots,e,\ldots) \in \mathcal{R}_P$. Note that $\varphi = \psi\{x_k \mapsto e\}$, $j > i+1$, and $\mathsf{state}_{i+1}(\vec{x})\,[\varphi] \to_{\mathcal{R}_P} \mathsf{state}_j(\vec{x})\,[\psi]$.[4] This case corresponds to the application of SIMPLIFICATION to $(\{\ \mathsf{chk}(\mathsf{state}_{i+1}(\vec{x})) \approx \mathsf{true}\ [\varphi]\ \} \uplus \mathcal{E}, \mathcal{H})$.

- (the beginning of "while" statements)

$$Trans_1(\boxed{\begin{array}{ll} i & @\zeta; \\ i+1 & \textbf{while}@\,\xi\,(\varphi)\{ \\ i+2 & \quad @\zeta \wedge \varphi; \\ \vdots & \vdots \end{array}}, \{\ \mathsf{chk}(\mathsf{state}_{i+1}(\vec{x})) \approx \mathsf{true}\ [\zeta]\ \} \uplus \mathcal{E}, \mathcal{H})$$
$$= (\boxed{\begin{array}{ll} i+2 & @\zeta \wedge \varphi; \\ \vdots & \vdots \end{array}}, \left\{\begin{array}{l} \mathsf{chk}(\mathsf{state}_j(\vec{x})) \approx \mathsf{true}\ [\ \zeta \wedge \varphi\ ] \\ \mathsf{chk}(\mathsf{state}_k(\vec{x})) \approx \mathsf{true}\ [\zeta \wedge \neg\varphi] \end{array}\right\} \cup \mathcal{E},$$
$$\{\ \mathsf{chk}(\mathsf{state}_{i+1}(\vec{x})) \to \mathsf{true}\ [\zeta]\ \} \cup \mathcal{H})$$

[4] This is because $\mathsf{state}_{i+1}(\vec{x})\,[\varphi] = \mathsf{state}_{i+1}(\vec{x})\,[\psi\{x_k \mapsto e\}] \to_{\mathtt{base}} \mathsf{state}_j(x_1,\ldots,x_{k-1},e,x_{k+1},\ldots,x_n)\,[\psi\{x_k \mapsto e\}] \sim \mathsf{state}_j(\vec{x})\,[\psi]$.

where $\mathsf{state}_{i+1}(\vec{x}) \to \mathsf{state}_j(\vec{x})\ [\varphi], \mathsf{state}_{i+1}(\vec{x}) \to \mathsf{state}_k(\vec{x})\ [\neg\varphi] \in \mathcal{R}_P$. Note that $i+1 < j < k$. This case corresponds to the application of EXPANSION to $(\{\ \mathsf{chk}(\mathsf{state}_{i+1}(\vec{x})) \approx \mathsf{true}\ [\zeta]\ \} \uplus \mathcal{E}, \mathcal{H})$.

- (the end of "while" statements)

$$\mathit{Trans}_1\left(\boxed{\begin{array}{ll} i & @\zeta; \\ i+1 & \} \\ i+2 & @\zeta \wedge \neg\varphi; \\ \vdots & \vdots \end{array}}, \{\ \mathsf{chk}(\mathsf{state}_{i+1}(\vec{x})) \approx \mathsf{true}\ [\zeta]\ \} \uplus \mathcal{E}, \mathcal{H}\right) = \left(\boxed{\begin{array}{ll} i+2 & @\zeta \wedge \neg\varphi; \\ \vdots & \vdots \end{array}}, \mathcal{E}, \mathcal{H}\right)$$

where $\mathsf{state}_{i+1}(\vec{x}) \to \mathsf{state}_j(\vec{x}) \in \mathcal{R}_P$, $\mathsf{chk}(\mathsf{state}_j(\vec{x})) \to \mathsf{true}\ [\zeta] \in \mathcal{H}$, and $j < i+1$. This case corresponds to the application of SIMPLIFICATION with rule $\mathsf{state}_{i+1}(\vec{c}) \to \mathsf{state}_j(\vec{x}) \in \mathcal{R}_P$, SIMPLIFICATION with rule $\mathsf{chk}(\mathsf{state}_j(\vec{x})) \to \mathsf{true}\ [\zeta] \in \mathcal{H}$, and DELETION:

$$\begin{array}{ll} (\{\mathsf{chk}(\mathsf{state}_{i+1}(\vec{x})) \approx \mathsf{true}\ [\zeta]\} \cup \mathcal{E}, \mathcal{H}) & \vdash_{RI} (\{\mathsf{chk}(\mathsf{state}_j(\vec{x})) \approx \mathsf{true}\ [\zeta]\} \cup \mathcal{E}, \mathcal{H}) \\ & \vdash_{RI} (\{\mathsf{true} \approx \mathsf{true}\ [\zeta]\} \cup \mathcal{E}, \mathcal{H}) \\ & \vdash_{RI} (\mathcal{E}, \mathcal{H}) \end{array}$$

- (the beginning of "if" statements)

$$\mathit{Trans}_1\left(\boxed{\begin{array}{ll} i & @\varphi; \\ i+1 & \mathbf{if}(\psi)\{ \\ i+2 & @\varphi \wedge \psi; \\ \vdots & \vdots \end{array}}, \{\ \mathsf{chk}(\mathsf{state}_{i+1}(\vec{x})) \approx \mathsf{true}\ [\varphi]\ \} \uplus \mathcal{E}, \mathcal{H}\right)$$

$$= \left(\boxed{\begin{array}{ll} i+2 & @\varphi \wedge \psi; \\ \vdots & \vdots \end{array}}, \left\{\begin{array}{l} \mathsf{chk}(\mathsf{state}_j(\vec{x})) \approx \mathsf{true}\ [\ \varphi \wedge \psi\ ] \\ \mathsf{chk}(\mathsf{state}_k(\vec{x})) \approx \mathsf{true}\ [\varphi \wedge \neg\psi] \end{array}\right\} \cup \mathcal{E}, \mathcal{H}\right)$$

where $\mathsf{state}_{i+1}(\vec{x}) \to \mathsf{state}_j(\vec{x})\ [\psi], \mathsf{state}_{i+1}(\vec{x}) \to \mathsf{state}_k(\vec{x})\ [\neg\psi] \in \mathcal{R}_P$. Note that $i+1 < j < k$. This case corresponds to the application of CASESPLITTING to $(\{\ \mathsf{chk}(\mathsf{state}_{i+1}(\vec{x})) \approx \mathsf{true}\ [\varphi]\ \} \uplus \mathcal{E}, \mathcal{H})$.

- (the beginning of "else" statements)

$$\mathit{Trans}_1\left(\boxed{\begin{array}{ll} i & @\xi; \\ i+1 & \}\mathbf{else}\{ \\ i+2 & @\varphi \wedge \neg\psi; \\ \vdots & \vdots \end{array}}, \left\{\begin{array}{l} \mathsf{chk}(\mathsf{state}_{i+1}(\vec{x})) \approx \mathsf{true}\ [\ \ \xi\ \ ] \\ \mathsf{chk}(\mathsf{state}_j(\vec{x})) \approx \mathsf{true}\ [\varphi \wedge \neg\psi] \end{array}\right\} \uplus \mathcal{E}, \mathcal{H}\right)$$

$$= \left(\boxed{\begin{array}{ll} i+2 & @\varphi \wedge \neg\psi; \\ \vdots & \vdots \end{array}}, \left\{\begin{array}{l} \mathsf{chk}(\mathsf{state}_k(\vec{x})) \approx \mathsf{true}\ [\ \ \xi\ \ ] \\ \mathsf{chk}(\mathsf{state}_j(\vec{x})) \approx \mathsf{true}\ [\varphi \wedge \neg\psi] \end{array}\right\} \cup \mathcal{E}, \mathcal{H}\right)$$

where $\mathsf{state}_{i+1}(\vec{x}) \to \mathsf{state}_k(\vec{x}) \in \mathcal{R}_P$. Note that $i+2 < j < k$. This case corresponds to the application of SIMPLIFICATION to $(\{\ \mathsf{chk}(\mathsf{state}_{i+1}(\vec{x})) \approx \mathsf{true}\ [\xi\,],\ \mathsf{chk}(\mathsf{state}_j(\vec{x})) \approx \mathsf{true}\ [\varphi \wedge \neg\psi]\ \} \uplus \mathcal{E}, \mathcal{H})$.

- (the end of "if" statements)

$$\mathit{Trans}_1\left(\boxed{\begin{array}{ll} i & @\xi; \\ i+1 & \} \\ i+2 & @\xi; \\ \vdots & \vdots \end{array}}, \left\{\begin{array}{l} \mathsf{chk}(\mathsf{state}_k(\vec{x})) \approx \mathsf{true}\ [\xi\,] \\ \mathsf{chk}(\mathsf{state}_{i+1}(\vec{x})) \approx \mathsf{true}\ [\xi\,] \end{array}\right\} \uplus \mathcal{E}, \mathcal{H}\right)$$

$$= \left(\boxed{\begin{array}{ll} i+2 & @\xi; \\ \vdots & \vdots \end{array}}, \{\ \mathsf{chk}(\mathsf{state}_k(\vec{x})) \approx \mathsf{true}\ [\xi\,]\ \} \cup \mathcal{E}, \mathcal{H}\right)$$

where $\mathsf{state}_{i+1}(\vec{x}) \to \mathsf{state}_k(\vec{x}) \in \mathcal{R}_P$ and $i+1 < k$. This case corresponds to the application of SIMPLIFICATION to $(\{\ \mathsf{chk}(\mathsf{state}_k(\vec{x})) \approx \mathsf{true}\ [\xi\,],\ \mathsf{chk}(\mathsf{state}_{i+1}(\vec{x})) \approx \mathsf{true}\ [\xi\,]\ \} \uplus \mathcal{E}, \mathcal{H})$.

- (the end of tableaux)

$$\mathit{Trans}_1(\boxed{i \quad @\varphi;}\ , \{\ \mathsf{chk}(\mathsf{end}(\vec{x})) \approx \mathsf{true}\ [\varphi]\ \} \uplus \mathcal{E}, \mathcal{H}) = (\varepsilon, \mathcal{E}, \mathcal{H})$$

Note that the last element of T_P is $@\psi_P$ and thus, $\varphi = \psi_P$. Note also that $\mathsf{chk}(\mathsf{end}(\vec{x})) \to \mathsf{true}\ [\psi_P] \in \mathcal{R}_{check}$. This case corresponds to the application of SIMPLIFICATION and DELETION:

$$(\{\mathsf{chk}(\mathsf{end}(\vec{x})) \approx \mathsf{true}\ [\varphi]\} \cup \mathcal{E}, \mathcal{H}) \vdash_{RI} (\{\mathsf{true} \approx \mathsf{true}\ [\varphi]\} \cup \mathcal{E}, \mathcal{H}) \vdash_{RI} (\mathcal{E}, \mathcal{H})$$

By by the definition of proof tableaux and Trans_1, Trans_1 satisfies the following properties.

Lemma 5.3 *If $\mathit{Trans}_1(T, \mathcal{E}, \mathcal{H}) = (T', \mathcal{E}', \mathcal{H}')$, then (a) $(\mathcal{E}, \mathcal{H}) \vdash_{RI}^* (\mathcal{E}', \mathcal{H}')$, and (b) if $T' \neq \varepsilon$, then Trans_1 is applicable to $(T', \mathcal{E}', \mathcal{H}')$.*

Next, we define a function *Trans* that applies Trans_1 to $(T_P, \{e_P\}, \emptyset)$ as much as possible, returning a list of RI processes:

- $\mathit{Trans}(\varepsilon, \mathcal{E}, \mathcal{H}) = (\mathcal{E}, \mathcal{H})$, and
- $\mathit{Trans}(T, \mathcal{E}, \mathcal{H}) = (\mathcal{E}, \mathcal{H}), \mathit{Trans}(T', \mathcal{E}', \mathcal{H}')$ where $T \neq \varepsilon$ and $\mathit{Trans}_1(T, \mathcal{E}, \mathcal{H}) = (T', \mathcal{E}', \mathcal{H}')$.[5]

By definition and Lemma 5.3, *Trans* satisfies the following properties.

Lemma 5.4 *$\mathit{Trans}(T_P, \{e_P\}, \emptyset)$ returns a finite sequence of RI processes.*

Proof. The first argument of *Trans* is a proof tableau and the length is decreasing when *Trans* is recursively called. It follows from Lemma 5.3 (b) that *Trans* calls Trans_1 until the first argument (suffixes of T_P) becomes ε. Therefore, *Trans* halts, returning a finite sequence of RI processes. □

Lemma 5.5 *Let the result of $\mathit{Trans}(T_P, \{e_P\}, \emptyset)$ be a sequence $(\mathcal{E}_1, \mathcal{H}_1), (\mathcal{E}_2, \mathcal{H}_2), \ldots, (\mathcal{E}_n, \mathcal{H}_n)$. Then, $(\{e_P\}, \emptyset) = (\mathcal{E}_1, \mathcal{H}_1) \vdash_{RI}^* (\mathcal{E}_2, \mathcal{H}_2) \vdash_{RI}^* \cdots \vdash_{RI}^* (\mathcal{E}_n, \mathcal{H}_n) = (\emptyset, \mathcal{H}_n)$.*

Proof. By definition, it is clear that the head of the resulting sequence is $(\{e_P\}, \emptyset)$. The last call of *Trans* takes ε as the first argument, and thus the last call of Trans_1 returns $(\varepsilon, \mathcal{E}_n, \mathcal{H}_n)$. In the case of the beginning of "while" or "if" statements, Trans_1 adds an equation to the second argument, and in the case of the end of "while" or "if" statements, Trans_1 removes an equation from the second argument. This means that in the case of the end of T_P, the number of remaining equations is one, i.e., $|\mathcal{E}_{n-1}| = 1$. It follows from the last application $\mathit{Trans}_1(\ldots, \mathcal{E}_{n-1}, \mathcal{H}_{n-1}) = (\varepsilon, \mathcal{E}_n, \mathcal{H}_n)$ that $\mathcal{E}_{n-1} = \{\mathsf{chk}(\mathsf{end}(\vec{x})) \approx \mathsf{true}\ [\psi_P]\}$ and $\mathcal{E}_n = \emptyset$. It follows from Lemma 5.3 (a) that $(\mathcal{E}_i, \mathcal{H}_i) \vdash_{RI}^* (\mathcal{E}_{i+1}, \mathcal{H}_{i+1})$ for all $1 \leq i < n$. Therefore, this lemma holds. □

Finally, we show that termination of $\mathcal{R}_P$ implies both termination of $\mathcal{R}_P \cup \mathcal{R}_{check} \cup \mathcal{H}$ and total correctness of P w.r.t. φ_P and ψ_P. Let $\mathit{Trans}(T_P, \{e_P\}, \emptyset) = (\{e_P\}, \emptyset), \ldots, (\emptyset, \mathcal{H})$. We have already shown that termination of $\mathcal{R}_P$ implies termination of $\mathcal{R}_P \cup \mathcal{R}_{check}$ (Lemma 5.1 (b)). Thus, we show that termination of $\mathcal{R}_P$ implies termination of $\mathcal{R}_P \cup \mathcal{R}_{check} \cup \mathcal{H}$. Since the right-hand sides of oriented equations in $\mathcal{H}$ are always true, $\mathcal{H}$ is always terminating and does not introduce non-termination into $\mathcal{R}_P \cup \mathcal{R}_{check}$. This means that if $\mathcal{R}_P \cup \mathcal{R}_{check}$ is terminating, then so is $\mathcal{R}_P \cup \mathcal{R}_{check} \cup \mathcal{H}$.

[5] The result of $\mathit{Trans}(T, \mathcal{E}, \mathcal{H})$ is a sequence "$(\mathcal{E}, \mathcal{H}), \mathit{Trans}(T', \mathcal{E}', \mathcal{H}')$" that has $(\mathcal{E}, \mathcal{H})$ as its head element.

Theorem 5.6 *If $\mathcal{R}_P$ is terminating, then $\mathcal{R}_P \cup \mathcal{R}_{check} \cup \mathcal{H}$ is so.*

As a consequence of Lemma 5.5 and Theorem 5.6, we have the following result.

Theorem 5.7 *If $\mathcal{R}_P$ is terminating, then $(\{e_P\}, \emptyset) \vdash_{RI}^* \cdots \vdash_{RI}^* (\emptyset, \mathcal{H})$ is valid, and thus, $[\varphi_P]\, P\, [\psi_P]$ holds (i.e., P is totally correct w.r.t. φ_P and ψ_P).*

Theorem 5.7 means that if $\{\varphi_P\}\ P\ \{\psi_P\}$ is proved to hold (via T_P), then (1) there exists an inference sequence of RI, and (2) if $\mathcal{R}_P$ is terminating, then $[\varphi_P]\, P\, [\psi_P]$ can be proved to hold without using inference rules for proving total correctness.

6 Conclusion

In this paper, we showed that a proof tableau for partial correctness can be transformed into an inference sequence of RI, and also showed that if the corresponding LCTRS is terminating, then the inference sequence is valid and the program is totally correct w.r.t. the specified pre- and post-conditions. Our result indicates that if we can prove partial correctness of a program by Hoare logic, then there exists a way to prove it by RI. However, this does not mean that RI is better than Hoare logic. 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 $\mathcal{R}_1$.

Hoare logic often requires appropriate loop invariants, but once finding such invariants, we can construct a proof tableau in a deterministic way. On the other hand, there must be several inference sequences of RI, and for automation, RI requires an appropriate strategy for the application of inference rules. In addition to the strategy, to apply GENERALIZATION in this paper, we have to, given a constraint φ, find an appropriate formula ψ such that $\varphi \implies \psi$ is valid and ψ makes the later inference succeed. In Section 5.1, we had the proof tableau T_{sum} with an appropriate loop invariant, and thus, we could apply GENERALIZATION, succeeding in transforming T_{sum} into a valid inference sequence of RI. However, this is not always possible. For this reason, it is worth improving tools for RI so as to directly prove (A1) an inductive theorem of $\mathcal{R}_1$.

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. However, it is not clear how to use ranking functions to prove termination of the corresponding LCTRS. Recall that termination of programs is not preserved by the conversion to LCTRSs. For this reason, there is a program such that there exists a ranking function to ensure termination of the program but the corresponding LCTRS is not terminating. 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.

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 of our first submission 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 Alessandro Armando, Peter Baumgartner & Gilles Dowek, editors: *Proceedings of the 4th International Joint Conference on Automated Reasoning*, *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 Andrei Voronkov, editor: *Proceedings of the 19th International Conference on Rewriting Techniques and Applications*, *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 Renate A. Schmidt, editor: *Proceedings of the 22nd International Conference on Automated Deduction*, *Lecture Notes in Computer Science* 5663, Springer, pp. 277–293, doi:10.1007/978-3-642-02959-2_22.

[6] Stephan Falke & Deepak Kapur (2012): *Rewriting Induction + Linear Arithmetic = Decision Procedure*. In Bernhard Gramlich, Dale Miller & Uli Sattler, editors: *Proceedings of the 6th International Joint Conference on Automated Reasoning*, *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 Ralf Treinen, editor: *Proceedings of the 20th International Conference on Rewriting Techniques and Applications*, *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 Transactions on Computational Logic* 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 Transactions on Programming* 1(2), pp. 100–121. In Japanese (a translated summary is available from `http://www.trs.css.i.nagoya-u.ac.jp/crisys/`).

[10] Gérard Huet & Jean-Marie Hullot (1982): *Proof by Induction in Equational Theories with Constructors*. *Journal of Computer and System Science* 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 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.

[12] Cynthia Kop & Naoki Nishida (2014): *Automatic Constrained Rewriting Induction towards Verifying Procedural Programs*. In Jacques Garrigue, editor: *Proceedings of the 12th Asian Symposium on Programming Languages and Systems*, *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 Martin Davis, Ansgar Fehnker, Annabelle McIver & Andrei Voronkov, editors: *Proceedings of the 20th International Conference on Logic*

for Programming, Artificial Intelligence, and Reasoning, *Lecture Notes in Computer Science* 9450, Springer, pp. 549–557, doi:10.1007/978-3-662-48899-7_38.

[14] David R. Musser (1980): *On Proving Inductive Properties of Abstract Data Types*. In Paul W. Abrahams, Richard J. Lipton & Stephen R. Bourne, editors: *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pp. 154–162, doi:10.1145/567446.567461.

[15] Naoki Nishida & Takumi Kataoka (2014): *On Improving Termination Preservability of Transformations from Procedural Programs into Rewrite Systems by Using Loop Invariants*. In Carsten Fuhs, editor: *Proceedings of the 14th International Workshop on Termination*, pp. 1–5.

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

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

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

[19] Tsubasa Sakata, Naoki Nishida & Toshiki Sakabe (2011): *On Proving Termination of Constrained Term Rewrite Systems by Eliminating Edges from Dependency Graphs*. In Herbert Kuchen, editor: *Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming*, *Lecture Notes in Computer Science* 6816, Springer, pp. 138–155, doi:10.1007/978-3-642-22531-4_9.

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

[21] Germán Vidal (2012): *Closed Symbolic Execution for Verifying Program Termination*. In: *Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation*, IEEE Computer Society, pp. 34–43, doi:10.1109/SCAM.2012.13.