Extending a Lemma Generation Approach for Rewriting Induction on Logically Constrained Term Rewriting Systems

Wouter Florian Brozius
VU Amsterdam, Netherlands
w.f.brozius@vu.nl

Kasper Hagens
RU Nijmegen, Netherlands
kasper.hagens2@ru.nl

Cynthia Kop
RU Nijmegen, Netherlands
c.kop@cs.ru.nl

When transforming a program into a more efficient version (for example by translating recursive functions into iterative ones), it is important that the input-output-behavior remains the same. One approach to assure this uses Logically Constrained Term Rewriting Systems (LCTRSs). Two versions of a program are translated into LCTRSs and compared in a proof system (Rewriting Induction). Proving their equivalence in this system often requires the introduction of a lemma. In this paper we review a lemma generation approach by Fuhs, Kop and Nishida and propose two possible extensions.

1 Introduction

During program development, it is common to do meaning-preserving transformations, for example for optimization purposes or to refactor in preparation for later updates. When doing so, it is important that the input-output-behavior remains the same. In this paper, we explore a method to prove that this holds.

The core of this method is to abstract both versions of the program into Logically Constrained Term Rewriting Systems (LCTRSs) and then proving equivalence of those LCTRSs using a proof system called Rewriting Induction (RI). Although equivalence is undecidable in general, it is often possible to automatically find a proof in this way, using various strategies and lemma generation techniques. In this paper, we build on one such approach, called *Initialization Generalization* [2].

This is *work in progress*: we have developed several ideas that would strengthen this method, but they are not fully formalized or automated, and we have thus far only tested them on a small benchmark set.

Paper overview. In section 2 and 3 we present an overview of the required literature. Section 4 contains our new ideas. We evaluate our ideas and present possible future work in section 5. Throughout this paper we will use the programs in figure 1 as leading examples, all of them implementations of $x \mapsto \sum_{i=1}^{x} i$.

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

(a)

```
int sum2(int x) {
    if (x > 0)
        return x + sum2(x-1);
    else
        return 0;
}
```

(b)

```
int sum3(int x) {
    z = 0;
    while (x > 0)
        z += x;
        x = x-1;
    return z;
}
```

(c)

Figure 1: Three (equivalent) implementations of the function $x \mapsto \sum_{i=1}^{x} i$ on $\mathbb{Z}$.

Submitted to:
WPTE 2022

2 Preliminaries

We briefly introduce LCTRSs and a simplified version of Rewriting Induction as defined in [1, 2].

2.1 Logically Constrained Term Rewriting

We assume familiarity with basic notions of many-sorted term rewriting, such as function symbols, variables, terms, substitutions and contexts. The type of a function symbol f is denoted $\mathsf{f} : [\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa$, indicating that f takes n arguments of sorts $\iota_1, \ldots, \iota_n$ respectively, and $\mathsf{f}(s_1, \ldots, s_n)$ has type κ. We denote $\mathcal{T}erms(\Sigma, \mathcal{V}ar)$ for the set of (well-sorted) terms built from symbols in Σ and variables in $\mathcal{V}ar$. In this paper, we usually let the set of sorts be $\{ \mathsf{bool}, \mathsf{int}, \mathsf{unit} \}$, and use symbols x, y, z, x_i, y_i, z_i for variables.

We assume given two many-sorted signatures Σ_{terms} and Σ_{theory}. Elements of $\mathcal{T}erms(\Sigma_{theory}, \mathcal{V}ar)$ are called *logical terms*: intuitively, these are terms like $x + y$ or $(14 * x - 12 > y) \wedge (z \neq 0)$ which have a *meaning* in some underlying theory. For each sort ι occurring in Σ_{theory}, called *theory sort*, we assume given a corresponding set $\mathcal{I}_\iota$; and we assume given a map $\mathcal{J}$ which assigns to every function symbol $\mathsf{f} : [\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa$ in Σ_{theory} a function from $\mathcal{I}_{\iota_1} \times \cdots \times \mathcal{I}_{\iota_n}$ to $\mathcal{I}_\kappa$. For example, we could map a symbol $+ : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int}$ to the plus operator in the theory of the integers, or $\wedge$ to logical conjunction. For every theory sort ι, we let the set $\mathcal{V}al_\iota$ of *values* of type ι contain exactly the theory symbols of type ι; i.e., $\mathcal{V}al_\iota := \{\mathsf{f} \in \Sigma_{theory} \mid \mathsf{f} : \iota\}$. We require that $\mathcal{J}$ is a bijection between $\mathcal{V}al_\iota$ and $\mathcal{I}_\iota$. Let $\mathcal{V}al$ be the set of all values in any $\mathcal{V}al_\iota$. The symbols in Σ_{theory} that are not values are called *calculation symbols*.

For ground logical terms, define $[\![f(s_1, \ldots, s_n)]\!] = \mathcal{J}(f)([\![s_1]\!], \ldots, [\![s_n]\!])$ and $[\![s]\!] = \mathcal{J}(s)$ if $s \in \mathcal{V}al$. A *constraint* is a logical term $c : \mathsf{bool}$, where $\mathcal{I}_{\mathsf{bool}} = \mathbb{B} = \{ \top, \bot \}$. A constraint c is *valid* if $[\![c\gamma]\!] = \top$ for all substitutions γ that map all the variables in c to values. A substitution γ *respects* a constraint φ if all variables in φ are substituted by values and $[\![\varphi\gamma]\!] = \top$. We typically assume $\Sigma_{theory} \supseteq \Sigma^{core}_{theory}$, where Σ^{core}_{theory} contains $\mathsf{true}, \mathsf{false} : \mathsf{bool}, \vee, \wedge : [\mathsf{bool} \times \mathsf{bool}] \Rightarrow \mathsf{bool}, \neg : [\mathsf{bool}] \Rightarrow \mathsf{bool}$. with $\mathcal{I}_{\mathsf{bool}} = \mathbb{B}$. and where $\mathcal{J}$ maps these symbols to their usual interpretation. We use infix notation for the binary operators in Σ_{theory}.

Example 1 Consider $\Sigma_{terms} = \{ \mathsf{sum1} : [\mathsf{int}] \Rightarrow \mathsf{unit},\ \mathsf{max} : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int} \}$ and $\Sigma_{theory} = \Sigma^{core}_{theory} \cup \{ \mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z} \}$. Note that unit does not occur in Σ_{theory}. Let $\mathcal{I}(\mathsf{bool}) = \mathbb{B}$ and $\mathcal{I}(\mathsf{int}) = \mathbb{Z}$ with $\mathcal{J}$ mapping all symbols to their usual interpretation. Then false, true and all $\mathsf{n} : \mathsf{int}$ are values. Examples of logical terms are $t_1 = \mathsf{true}$, $t_2 = \neg(x \wedge y)$, $t_3 = x \vee \neg x$, $t_4 = \mathsf{true} \vee \mathsf{false}$. The interpretation of t_4 is: $[\![t_4]\!] = (\top$ "or" $\bot) = \top$. All t_i are constraints; t_1, t_3 and t_4 are valid. An example of a non logical term is $\mathsf{max}(x, 0)$.

A rewriting rule is a triple $\ell \to r\,[\varphi]$, where ℓ and r are terms and φ a constraint. We write $\ell \to r$ if $\varphi = \mathsf{true}$. The root of ℓ must be in $\Sigma_{terms} \backslash \Sigma_{theory}$ and ℓ and r must have the same sort. Note that it is allowed to have variables in r which do not occur in ℓ (which we will use extensively in our method). The set of *logic variables* of a rule $\ell \to r\,[\varphi]$, notation $LVar(\ell \to r\,[\varphi])$, contains all variables in $\mathcal{V}ar(\varphi)$ and all variables in $\mathcal{V}ar(r) \setminus \mathcal{V}ar(\ell)$. Define $\mathcal{R}_{\mathtt{calc}}$ to be $\{ f(\vec{x}) \to y\,[f(\vec{x}) = y] \mid f \in \Sigma_{theory} \backslash \mathcal{V}al \}$.

Given a set of rules $\mathcal{R}$, the rewrite relation $\to_{\mathcal{R}}$ is defined by: $C[\ell\gamma] \to_{\mathcal{R}} C[r\gamma]$ if $\ell \to r\,[\varphi] \in (\mathcal{R} \cup \mathcal{R}_{\mathtt{calc}})$, $[\![\varphi\gamma]\!] = \top$ and $\gamma(x) \in \mathcal{V}al$ for all $x \in LVar(\ell \to r\,[\varphi])$. That is, to apply a rule, the logic variables must be instantiated with values, and the constraint satisfied. For a rule $f(l_1, \ldots, l_n) \to r\,[\varphi] \in \mathcal{R}$ we call f a *defined symbol*. Let $\mathcal{D}$ be the set of defined symbols; symbols in $\Sigma \setminus \mathcal{D}$ are called *constructors*. A *logically constrained term rewriting system* (LCTRS) is the abstract rewriting system $(\mathcal{T}erms(\Sigma, \mathcal{V}ar), \to_{\mathcal{R}})$.

Example 2 Simple programs may be represented using an LCTRS (see, e.g., [2] for a translation method from simple C code). For example, the function from figure 1b may be represented by the LCTRS where $\Sigma_{theory} = \Sigma^{core}_{theory} \cup \{+, \geq, <\} \cup \{\mathsf{n} \mid n \in \mathbb{Z}\}$ (with types and $\mathcal{J}$ as expected; $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$), and Σ_{terms} contains

$\mathsf{sum1}, \mathsf{return} : [\mathsf{int}] \Rightarrow \mathsf{unit}$ and $\mathsf{u} : [\mathsf{int} \times \mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{unit}$, and $\mathcal{R}$ consists of the rules:

$$(R1)\quad \mathsf{sum1}(x) \;\to\; \mathsf{u}(x,1,0) \qquad (R2)\quad \mathsf{u}(x,i,z) \;\to\; \mathsf{u}(x,i+1,z+i) \;\; [i \le x]$$
$$(R3)\quad \mathsf{u}(x,i,z) \;\to\; \mathsf{return}(z) \;\; [i > x]$$

Here, $\mathit{LVar}((R1)) = \emptyset$ and $\mathit{LVar}((R2)) = \mathit{LVar}((R3)) = \{ x, i \}$ (the variables occurring in the constraints); this means that to apply these rules, x and i must be instantiated by values. Hence, $\mathsf{sum1}(1+1) \to_{\mathcal{R}} \mathsf{u}(1+1,1,0)$ by rule $(R1)$, but this term cannot be reduced by $(R2)$. We *can*, however, reduce it using the calculation rule $x+y \to z\ [x+y=z]$. Hence, $\mathsf{u}(1+1,1,0) \to_{\mathcal{R}} \mathsf{u}(2,1,0) \to_{\mathcal{R}} \mathsf{u}(2,1+1,0+1) \to_{\mathcal{R}} \mathsf{u}(2,2,0+1) \to_{\mathcal{R}} \mathsf{u}(2,2,1) \to_{\mathcal{R}} \mathsf{u}(2,2+1,2+1) \to_{\mathcal{R}} \mathsf{u}(2,3,2+1) \to_{\mathcal{R}} \mathsf{u}(2,3,3) \to_{\mathcal{R}} \mathsf{return}(3)$.

To reason about sets of terms, we can also rewrite *constrained terms* $s\ [\varphi]$. Essentially, this represents the set of terms $s\gamma$ such that $\varphi\gamma$ is valid. We say that two constrained terms are *equivalent* if they represent the same set; e.g., $\mathsf{f}(x)\ [x \ge 0]$ is equivalent to $\mathsf{f}(z)\ [2-3 < z]$. We will write $s\ [\varphi] \to_{\mathcal{R}} t\ [\psi]$ if each instance of s that satisfies φ reduces in one step to an instance of t that satisfies ψ, and all such instances of t can be obtained in this way. Formally, for a constrained term $s\ [\varphi]$ and rule $\rho = \ell \to r\ [\psi]$, define $s\ [\varphi] \to_\rho t\ [\varphi]$ if for some context C and substitution γ we have $s = C[\ell\gamma]$, $t = C[r\gamma]$, $\varphi \Rightarrow \psi\gamma$ is valid and for all $x \in \mathit{LVar}(\rho)$ we have that $\gamma(x)$ is either a value or a variable in $\mathcal{V}ar(\varphi)$.

We rewrite constrained terms modulo term equivalence, which allows us to for instance rewrite $\mathsf{f}(a.a+1)\ [a>0]$ to $\mathsf{f}(a,b)\ [b>0 \wedge b = a+1]$: $\mathsf{f}(a,a+1)\ [a>0]$ is equivalent to $\mathsf{f}(a,a+1)\ [a>0 \wedge b = a+1]$, and we can then use the calculation rule $x+y \to z\ [x+y=z]$ with substitution $[x := a, y := 1, z := b]$.

Example 3 Let us rewrite a constrained term $\mathsf{u}(a,b,0)\ [a \ge 2 \wedge 2 \ge b]$ in the LCTRS of Example 2. We have $\mathsf{u}(a,b,0)\ [a \ge 2 \wedge 2 \ge b] \to_{\mathcal{R}} \mathsf{u}(a,b+1,0+b)\ [a \ge 2 \wedge 2 \ge b]$ by $(R2)$ (using $\gamma = [x := a, i := b]$), because if $a \ge 2 \wedge 2 \ge b$, holds, then $a = \gamma(x) \ge \gamma(i) = b$. By the calculation rule $x+y \to z\ [x+y=z]$ (and using equivalence), this $\to_{\mathcal{R}} \mathsf{u}(a,b',0+b\ [a \ge 2 \wedge 2 \ge b \wedge b' = b+1]$. By the same calculation rule, this $\to_{\mathcal{R}} \mathsf{u}(a,b',b)\ [a \ge 2 \wedge 2 \ge b \wedge b' = b+1]$. This constrained term cannot be reduced any further.

2.2 Rewriting Induction

Restrictions. We limit interest to LCTRSs $\mathcal{R}$ where: (1) $\Sigma_{\mathit{theory}} \supseteq \Sigma^{\mathit{core}}_{\mathit{theory}}$; (2) $\mathcal{R}$ is terminating, i.e. there are no infinite reductions $s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} s_3 \to_{\mathcal{R}} \ldots$; (3) $\mathcal{R}$ is quasi-reductive, i.e. every ground term is either a constructor term or reduces; and (4) every sort in Σ has at least one ground term.

Inductive theorems. An *equation* is a triple $s \approx t\ [\varphi]$ with s,t terms of the same type and φ a constraint. A substitution γ *respects* $s \approx t\ [\varphi]$ if γ respects φ and $\mathcal{V}ar(s) \cup \mathcal{V}ar(t) \subseteq \mathit{Dom}(\gamma)$. An equation $s \approx t\ [\varphi]$ is an *inductive theorem* if $s\gamma \;\leftrightarrow^*_{\mathcal{R}}\; t\gamma$ for all substitutions γ that respect this equation and which map all variables in s,t to ground constructor terms. Intuitively, $f(\vec{x}) \approx g(\vec{x})\ [\varphi]$ is an inductive theorem if f and g agree on all $\vec{x}$ for which $\varphi(\vec{x})$ holds. To prove that an equation is an inductive theorem we can use *rewriting induction*; a deduction system for manipulating proofstates.

A *proofstate* is a pair $(\mathcal{E},\mathcal{H})$ where $\mathcal{E}$ is a set of equations and $\mathcal{H}$ is a set of induction hypotheses: rewriting rules which have been derived during the rewriting induction. A proofstate $(\mathcal{E}_1,\mathcal{H}_1)$ might be transformed into another proofstate $(\mathcal{E}_2,\mathcal{H}_2)$, notation $(\mathcal{E}_1,\mathcal{H}_1) \;\vdash_{\mathtt{ri}}\; (\mathcal{E}_2,\mathcal{H}_2)$, by using one of the deduction rules which we explain below. The transitive, reflexive closure of $\vdash_{\mathtt{ri}}$ is denoted by $\vdash^*_{\mathtt{ri}}$.

The use of rewriting induction is shown by the following theorem:

Theorem 1 [2] *If* $(\mathcal{E},\emptyset) \;\vdash^*_{ri}\; (\emptyset,\mathcal{H})$ *for some set* $\mathcal{H}$*, then every equation in* $\mathcal{E}$ *is an inductive theorem.*

For readability, we will often write $\mathcal{E}_1 \;\vdash_{\mathtt{ri}}\; \mathcal{E}_2$, omitting the $\mathcal{H}$ component, and only state when it is changed. A proofstate can be manipulated using *deduction rules*. Below we shall present simplified versions of the most fundamental deduction rules from [2].We do this by example, using the LCTRS $\mathcal{R}$

that contains rules $(R1)\cdots(R3)$ from example 2 and the following rules corresponding to Figure 1b.

$$\begin{array}{llllll} (R4) & \mathsf{sum2}(x) & \to & \mathsf{add}(x,\mathsf{sum2}(x-1)) & [x>0] \\ (R5) & \mathsf{sum2}(x) & \to & \mathsf{return}(0) & [x\leq 0] \end{array} \qquad (R6)\quad \mathsf{add}(x,\mathsf{return}(y)) \to \mathsf{return}(x+y)$$

If we aim to derive that $\mathsf{sum1}(x) \approx \mathsf{sum2}(x)$ is an inductive theorem, then according to Theorem 1, we need to show that there is a deduction sequence $(\mathcal{E},\emptyset) \vdash_{\mathtt{ri}}^{*} (\emptyset,\mathcal{H})$, where $\{\mathsf{sum1}(x) \approx \mathsf{sum2}(x)\} \subseteq \mathcal{E}$.

`SIMPLIFICATION`. We may apply a rewrite rule from $\mathcal{R}$ or $\mathcal{H}$ to one side of an equation, where for rewriting purposes the equation is viewed as a single constrained term with $\approx$ a new function symbol.

Using `SIMPLIFICATION` on $\mathsf{sum1}(x)$ with $(R1)$ gives

$$\left\{ \underline{\mathsf{sum1}(x)} \approx \mathsf{sum2}(x) \right\} \quad \vdash_{\mathtt{ri}} \quad \{\ \mathsf{u}(x,1,0) \approx \mathsf{sum2}(x)\ \}$$

`EXPANSION`. We may do a case analysis on one side of an equation relative to the rules in $\mathcal{R}$. For every case, a new equation is added with corresponding constraint. Furthermore, if s in $s \approx t\ [\varphi]$ is the subject of an expansion and $\mathcal{R}\cup\mathcal{H}\cup\{s \to t\ [\varphi]\}$ is terminating, then the rewrite rule $s \to t\ [\varphi]$ is added to $\mathcal{H}$. If t is the subject of expansion, then $t \to s\ [\varphi]$ is added instead (if $\mathcal{R}\cup\mathcal{H}\cup\{t \to s\ [\varphi]\}$ is terminating).

Using `EXPANSION` on $\mathsf{sum2}(x)$ causes $(H1)$ $\mathsf{sum2}(x) \to \mathsf{u}(x,1,0)$ to be added to $\mathcal{H}$, and gives:

$$\left\{ \mathsf{u}(x,1,0) \approx \underline{\mathsf{sum2}(x)} \right\} \ \vdash_{\mathtt{ri}} \ \left\{ \begin{array}{l} \mathsf{u}(x,1,0) \approx \mathsf{add}(x,\mathsf{sum2}(x-1))\ [x>0], \\ \mathsf{u}(x,1,0) \approx \mathsf{return}(0)\ [x\leq 0] \end{array} \right\}$$

Now $(R2)$, $(R3)$ and $\mathcal{R}_{\mathtt{calc}}$ can be used to apply `SIMPLIFICATION` on the equations to get

$$\left\{ \begin{array}{l} \underline{\mathsf{u}(x,1,0)} \approx \mathsf{add}(x,\mathsf{sum2}(x-1))\ [x>0], \\ \underline{\mathsf{u}(x,1,0)} \approx \mathsf{return}(0)\ [x\leq 0] \end{array} \right\} \vdash_{\mathtt{ri}}^{*} \left\{ \begin{array}{l} \mathsf{u}(x,2,1) \approx \mathsf{add}(x,\mathsf{sum2}(x-1))\ [x>0], \\ \mathsf{return}(0) \approx \mathsf{return}(0)\ [x\leq 0] \end{array} \right\}$$

In addition, we now have access to the induction rule $(H1)$. We can use this to rewrite $\mathsf{sum2}(x-1)$:

$$\left\{ \begin{array}{l} \mathsf{u}(x,2,1) \approx \mathsf{add}(x,\underline{\mathsf{sum2}(x-1)})\ [x>0], \\ \mathsf{return}(0) \approx \mathsf{return}(0)\ [x\leq 0] \end{array} \right\} \vdash_{\mathtt{ri}}^{*} \left\{ \begin{array}{l} \mathsf{u}(x,2,1) \approx \mathsf{add}(x,\mathsf{u}(x-1,1,0))\ [x>0], \\ \mathsf{return}(0) \approx \mathsf{return}(0)\ [x\leq 0] \end{array} \right\}$$

`DELETION`. An equation $s \approx t\ [\varphi]$ may be deleted if $s = t$ or $[\![\varphi]\!] = \bot$.

Following this rule, we may remove the second equation in our running example from the proofstate.

The remaining equation contains the (sub)term $\mathsf{u}(x-1,1,0)$. To support the application of rewrite rules, it is usually convenient to replace a calculation (e.g. $x-1$) by a fresh variable (e.g. x_1), where we update the constraint with a corresponding equality (e.g. $x_1 = x-1$). This can be done using the `SIMPLIFICATION` rule. In this case, replacing $x-1$ by x_1 yields:

$$\{\ \mathsf{u}(x,2,1) \approx \mathsf{add}(x,\mathsf{u}(\underline{x-1},1,0))\ [x>0]\ \} \ \vdash_{\mathtt{ri}} \ \{\ \mathsf{u}(x,2,1) \approx \mathsf{add}(x,\mathsf{u}(x_1,1,0))\ [x>0 \wedge x_1 = x-1]\ \}$$

At this point we do another `EXPANSION` step, now on the left side of the equation. This causes $(H2)$ $\mathsf{u}(x,2,1) \to \mathsf{add}(x,\mathsf{u}(x_1,1,0))\ [x>0 \wedge x_1 = x-1]$ to be added to $\mathcal{H}$, and yields the following proofstate, where we simplify the constraints as much as possible (we usually do this implicitly).

$$\vdash_{\mathtt{ri}} \ \left\{ \begin{array}{l} \mathsf{u}(x,2+1,1+2) \approx \mathsf{add}(x,\mathsf{u}(x_1,1,0))\ [x>1 \wedge x_1 = x-1], \\ \mathsf{return}(1) \approx \mathsf{add}(x,\mathsf{u}(x_1,1,0))\ [x=1 \wedge x_1 = x-1] \end{array} \right\}$$

We apply SIMPLIFICATION in the second equation on the right side with rules $(R3)$, $(R6)$ and $\mathcal{R}_{\texttt{calc}}$.

$$\begin{aligned}
&\{\ \mathsf{return}(1) \approx \mathsf{add}(x, \underline{\mathsf{u}(x_1,1,0)})\ [x=1 \wedge x_1 = x-1]\ \} \\
\vdash_{\texttt{ri}}\ &\{\ \mathsf{return}(1) \approx \underline{\mathsf{add}(x, \mathsf{return}(0))}\ [x=1 \wedge x_1 = x-1]\ \} \\
\vdash_{\texttt{ri}}\ &\{\ \mathsf{return}(1) \approx \mathsf{return}(\underline{x+0})\ [x=1]\ \}\ \vdash_{\texttt{ri}}\ \{\ \mathsf{return}(1) \approx \mathsf{return}(x)\ [x=1]\ \}
\end{aligned}$$

In the resulting equation, we cannot apply DELETION, since the equation terms are not syntactically equal. To be able to remove this equation still, the following ancillary deduction rule is introduced.

EQ-DELETION Applying EQ-DELETION to an equation $C[s_1,\ldots,s_n] \approx C[t_1,\ldots,t_n]\ [\varphi]$, where all s_i, t_i are logical terms adds the negation $\neg(\bigwedge_{i=1}^{n} s_i = t_i)$ to the constraint φ. Intuitively, this rule allows us to delete equations whose left and right side are not syntactically equal, but their equivalence is implied by the constraint. Applying this rule creates an unsatisfiable constraint that is then subject to the DELETION rule.

Applying EQ-DELETION to the equation above yields $\mathsf{return}(1) \approx \mathsf{return}(x)\ [x=1 \wedge \neg(x=1)]$, to which DELETION can be applied. After some simplifications to the remaining equation, we are left with:

$$\mathsf{u}(x,3,3) \approx \mathsf{add}(x, \mathsf{u}(x_1,2,1))\ [x > 1 \wedge x_1 = x-1]$$

Automation. The rewriting induction process can be automated using a strategy: a priority selection on the deduction rules. The strategy used in [2] tries to apply deduction rules in the following order: EQ-DELETION, DELETION, SIMPLIFICATION, EXPANSION.

Divergence As often happens in practical cases, and also in our running example, we eventually keep expanding but cannot apply the induction rules. Consider the proofstate after a few more deductions:

$$\mathcal{E} = \{\ \mathsf{u}(x,5,10) \approx \mathsf{add}(x, \mathsf{u}(x_1,4,6))\ [x > 3 \wedge x_1 = x-1]\ \}$$

$$\mathcal{H} = \left\{ \begin{array}{l} \mathsf{sum2}(x) \to \mathsf{u}(x,1,0), \\ \mathsf{u}(x,2,1) \to \mathsf{add}(x, \mathsf{u}(x_1,1,0))\ [x > 0 \wedge x_1 = x-1], \\ \mathsf{u}(x,3,3) \to \mathsf{add}(x, \mathsf{u}(x_1,2,1))\ [x > 1 \wedge x_1 = x-1], \\ \mathsf{u}(x,4,6) \to \mathsf{add}(x, \mathsf{u}(x_1,3,3))\ [x > 2 \wedge x_1 = x-1] \end{array} \right\}$$

It is clear that no induction hypothesis will ever be applicable in an ongoing equation. What we encounter here is an instance of *divergence*. This is where our final rule comes in.

3 Generalization

As is often the case in mathematics, it may happen that proving a more general statement is easier than proving a particular instance of that statement. This can also be the case in rewriting induction, in the sense that an equation with a diverging proof may be a special case of a more general equation with a non-diverging proof. In our running example, the recurring equation in $\mathcal{E}$ is always an instance of

$$(H) \quad \mathsf{u}(x,i,z) \approx \mathsf{add}(x, \mathsf{u}(x-1, i-1, z-i+1))\ [x > i-2].$$

It turns out that equation H is an inductive theorem that can be solved using rewriting induction. This proof adds a useful rewriting rule to $\mathcal{H}$ that can be used to complete our original proof. In this setting, we call (H) a *lemma*. The question remains how such a lemma can be found automatically, without human reasoning. Much work has been done in this area (see e.g. [3, 4, 5, 6, 7, 8, 2]). Below we present one of these methods called *initialization generalization*. [2]

3.1 Initialization generalization [2]

In essence, initialization generalization (InGen) adapts an LCTRS by moving value instantiations to the constraints and drops these instantiations somewhere in the rewriting induction process.

Example 4 Rule $\mathsf{f}(x) \to \mathsf{g}(x+2,1)$ is replaced by $\mathsf{f}(x) \to \mathsf{g}(x+2,a_0)\ [a_0 = 1]$

Formally, for all rules $\ell \to r\ [\varphi]$: any value v in r that is the immediate subterm of a symbol in $\Sigma_{terms} \cap \mathcal{D}$ gets replaced by a 'fresh' variable a_i (i.e. a variable that does not already occur in the LCTRS) from some set $\mathcal{V}ar_{\texttt{init}}$. Then the equality $a_i = v$ is added to φ. (Since value 2 in example 4 above is an immediate subterm of $+$, which is not a symbol in Σ_{terms}, the value is *not* replaced by a variable.)

Secondly, we allow usage of a `GENERALIZATION` rule in the rewriting induction process that drops any value instantiations: equalities of the form $a = v$, where a is a variable from $\mathcal{V}ar_{\texttt{init}}$ and v is a value. For our strategy, we set the priority of `GENERALIZATION` between `SIMPLIFICATION` and `EXPANSION`. Lastly, the rules `SIMPLIFICATION` and `EXPANSION` are adapted to avoid simplifying these equalities away.

Example 5 Using InGen on our running example changes the rule $(R1)$ into:

$$(R1') \qquad \mathsf{sum1}(x) \to \mathsf{u}(x,a_1,a_0) \qquad [a_1 = 1 \wedge a_0 = 0]$$

(The other rules remain the same.) Aiming once again to prove that $\mathsf{sum1}(x) \approx \mathsf{sum2}(x)$ is an inductive theorem, we repeat the strategy from before until we arrive in the divergence pattern. As we aim to preserve the equalities $a_1 = 1$ and $a_0 = 0$, this yields the following (fully simplified) state:

$$\mathsf{u}(x,i_2,z_2) \approx \mathsf{add}\,(x,\mathsf{u}(x_1,i_1,z_1))$$
$$[i_2 = i_1 + 1 \wedge z_2 = z_1 + i_1 \wedge x_1 = x - 1 \wedge x \geq i_1 \wedge i_1 = a_1 + 1 \wedge z_1 = a_0 + a_1 \wedge x > 0 \wedge a_1 = 1 \wedge a_1 = 0]$$

We drop both value initializations and do `EXPANSION` on the left. This adds the following rule to $\mathcal{H}$:

$$\mathsf{u}(x,i_2,z_2) \to \mathsf{add}\,(x,\mathsf{u}(x_1,i_1,z_1))$$
$$[i_2 = i_1 + 1 \wedge z_2 = z_1 + i_1 \wedge x_1 = x - 1 \wedge x \geq i_1 \wedge i_1 = a_1 + 1 \wedge z_1 = a_1 + i_0 \wedge x > 0]$$

We now have an induction hypothesis that we *can* apply to an ongoing proofstate. After doing so, the inductive theorem can be proved. (Due to space requirements, we will not display the full proof here.)

4 Alternative generalization ideas

Of course, no generalization method is perfect; in general, equivalence is undecidable. Strategies using InGen allow us to prove more inductive theorems, but not all. We will see for example, that using InGen will not allow us to prove equivalence between sum2 and sum3, the LCTRS representations of the programs in figure 1b and 1c repectively. Proving the equivalence between sum1 and sum3 even harder.

In this section we present some new generalization techniques. We will show that the removal of a constraint is sometimes enough to obtain a valid, applicable lemma. In addition we propose a more general notion of an initialization, and an idea that involves extending the rewrite system.

This is *work in progress*. We will present ideas, not detailed methods. In particular, substantial steps will be needed to be able to formalize and automate these ideas.

4.1 Dropping constraints

Example 6 Consider the LCTRSs representations of the programs given in figure 1b and figure 1c:

$(R4)\ \mathsf{sum2}(x) \to \mathsf{return}(0)\ [x \leq 0]$
$(R5)\ \mathsf{sum2}(x) \to \mathsf{add}\,(x, \mathsf{sum2}(x-1))\ [x > 0]$
$(R6)\ \mathsf{add}\,(x, \mathsf{return}(y)) \to \mathsf{return}(x+y)$
$(R7)\ \mathsf{sum3}(x) \to \mathsf{v}(x,0)$
$(R8)\ \mathsf{v}(x,z) \to \mathsf{v}(x-1, z+x)\ [x > 0]$
$(R9)\ \mathsf{v}(x,z) \to \mathsf{return}(z)\ [x \leq 0]$

Aiming to show that $\mathsf{sum2}(x) \approx \mathsf{sum3}(x)$ is an inductive theorem using the strategy in Section 2.2 (without InGen) we observe the following divergence pattern (we removed superfluous constraints):
$\mathcal{E} = \{\mathsf{v}(x_5, z_4) \approx \mathsf{add}\,(x, \mathsf{v}(x_5, y_4))\ [x_5 = x_4 - 1 \wedge z_4 = 5x - 10 \wedge y_4 = 4x - 10 \wedge x > 4]\}$

$$\mathcal{H} = \left\{ \begin{array}{l} \mathsf{sum2}(x) \to \mathsf{v}(x,0), \\ \mathsf{v}(x_1, z_0) \to \mathsf{add}\,(x, \mathsf{v}(x_1, 0))\ [x_1 = x-1 \wedge z_0 = x \wedge x > 0], \\ \mathsf{v}(x_2, z_1) \to \mathsf{add}\,(x, \mathsf{v}(x_2, x_1))\ [x_2 = x-2 \wedge z_1 = 2x-1 \wedge x > 1], \\ \mathsf{v}(x_3, z_2) \to \mathsf{add}\,(x, \mathsf{v}(x_3, y_2))\ [x_3 = x-3 \wedge z_2 = 3x-3 \wedge y_2 = 2x-3 \wedge x > 2], \\ \mathsf{v}(x_4, z_3) \to \mathsf{add}\,(x, \mathsf{v}(x_4, y_3))\ [x_4 = x-4 \wedge z_3 = 4x-6 \wedge y_3 = 3x-6 \wedge x > 3] \end{array} \right\}$$

Using InGen, we obtain a rule $(R7')\ \mathsf{sum3}(x) \to \mathsf{v}(x, y_0)\ [y_0 = 0]$. Dropping the initialization $y_0 = 0$ before the second expansion yields an equation $\mathsf{v}(x_1, z_0) \approx \mathsf{add}\,(x, \mathsf{v}(x_1, y_0))\quad [z_0 = y_0 + x \wedge x_1 = x - 1 \wedge x > 0]$. This equation is not solvable using the strategy from before (Section 2.2), but if we drop the requirement $x_1 = x - 1$ then we obtain a lemma which *can* be used for proving $\mathsf{sum2}(x) \approx \mathsf{sum3}(x)$. [1]

Lemma 1 $\mathsf{v}(x_1, z_0) \approx \mathsf{add}\,(x, \mathsf{v}(x_1, y_0))\quad [z_0 = y_0 + x \wedge x > 0]$

This suggests that dropping constraints may be a sensible method for adjusting a non-useful equation produced by InGen. The question remains however, *which* constraints we should drop. Randomly dropping constraints and checking if the resulting lemma is solvable (and useful) is very inefficient since there are 2^n ways to choose a subset of a constraint consisting of n conjuncts.

4.2 Recognizing more initializations

To identify suitable constraints to drop, we extend InGen. Specifically, we recognize a broader range of (logical) terms as a kind of initializations, keep track of constraints derived from them, and consider these constraints as candidates for discarding. Applying this idea to $\mathsf{sum2}(x) \approx \mathsf{sum3}(x)$ also leads to lemma 1.

Definition 1 Let $\geq$ be the quasi-order on Σ_{terms} generated by $f \geq g$ if there is a rule $f(\ell_1, \ldots, \ell_n) \to r\,[\varphi] \in \mathcal{R}$ with g occurring in r. Define a corresponding strict order $> = (\geq \setminus \leq)$ and the set:

$$\mathcal{T}erms_{\mathtt{init}} = \left\{ r_i \in \mathcal{T}erms(\Sigma_{theory}, \mathcal{V}ar) \mid f(\ell_1, \ldots, \ell_n) \to C[g(r_1, \ldots, r_m)]\,[\varphi] \in \mathcal{R},\ f > g,\ g \in \mathcal{D} \right\}$$

Example 7 In example 6: $\geq$ is the transitive reflexive closure of $\mathsf{sum2} \geq \mathsf{add} \geq \mathsf{return}$, $\mathsf{sum3} \geq \mathsf{v} \geq \mathsf{return}$. Hence, $\mathsf{sum2} > \mathsf{add} > \mathsf{return}$, $\mathsf{sum3} > \mathsf{v} > \mathsf{return}$. There is one initialization x in $\mathsf{sum2}(x) \to \mathsf{add}\,(x, \mathsf{sum2}(x-1))\ [x > 0]$, and initializations x and 0 in $\mathsf{sum3}(x) \to \mathsf{v}\,(x, 0)$.

Definition 2 Given $\mathcal{R}$, fix sets $\mathcal{V}ar_{\mathtt{drop}}, \mathcal{V}ar_{\mathtt{keep}}$ of variables not occurring in $\mathcal{R}$ such that $\mathcal{V}ar_{\mathtt{drop}} \cap \mathcal{V}ar_{\mathtt{keep}} = \emptyset$. The initialization-free counterpart $\mathcal{R}'$ of $\mathcal{R}$ is obtained by stepwise replacing every rule $\ell \to C[s]\,[\varphi]$ in $\mathcal{R}$, with $s \in \mathcal{T}erms_{\mathtt{init}}$ by the following, until no such rules remain:

- $\ell \to C[a]\,[\varphi \wedge a = s]$ with $a \in \mathcal{V}ar_{\mathtt{drop}}$ fresh, if $s \in \mathcal{V}al$.
- $\ell \to C[I]\,[\varphi \wedge I = s]$ with $I \in \mathcal{V}ar_{\mathtt{keep}}$ fresh, if $s \notin \mathcal{V}al$.

[1] If we follow the rules exactly, the lemma needs a bit more modification since a rule $\mathsf{v}(x_1, z_0) \to \mathsf{add}(x, \mathsf{v}(x_1, y_0))[z_0 = y_0 + x \wedge x > 0]$ does not meet the termination requirement. However, this is easily repaired by adding $z_0 \geq 0$ to the constraint, which is certainly satisfied in the divergence.

Example 8 The initialization-free counterpart of the LCTRS in example 6 will only affect $(R5)$ and $(R7)$:

$$(R5')\ \mathsf{sum2}(x) \to \mathsf{add}(I_0, \mathsf{sum2}(x-1))\ [x>0 \wedge I_0 = x] \qquad (R7')\ \mathsf{sum3}(x) \to \mathsf{v}(I_1, a_0)\ [I_1 = x \wedge a_0 = 0]$$

Performing the rewriting induction $\mathsf{sum2}(x) \approx \mathsf{sum3}(\mathsf{x})$ with this altered LCTRS quickly yields a state with $\mathcal{E} = \{\mathsf{add}(I_0, \mathsf{v}(I_1-1, a_0)) \approx \mathsf{v}(I_1-1, a_0+I_1) \quad [I_0 = x \wedge x > 0 \wedge I_1 = x \wedge a_0 = 0]\}$

A first generalization-attempt would be to proceed similarly to InGen: dropping all initializations. However, in this example dropping non-value initializations $I = r$ makes the equation unsolvable: we lose crucial information on the relations between variables. Instead, we will keep track of constraints *derived* from non-value initializations using a calculation step. These constraints are candidates to be dropped.

Definition 3 Let $s \approx t\ [\varphi] \in \mathcal{E}$, where s has the form $C[r]$ with $r \in \mathcal{T}erms(\Sigma_{theory}, \mathcal{V}ar_{\mathtt{keep}})$ and r not a variable. Then $s \approx t\ [\varphi]$ may be rewritten to $C[a] \approx t\ [\varphi \wedge a = r]$ for $a \in \mathcal{V}ar_{\mathtt{drop}}$ fresh and suitably typed.

Note that this is just the result of applying one or more `SIMPLIFICATION` steps to r using calculation rules. Definition 3 simply allows us to give the fresh variable used in the constraint a special, recognizable name. This allows us to keep track of specific constraints derived from initializations.

Example 9 Consider the equation in example 8: $\mathsf{add}(I_0, \mathsf{v}(I_1-1, a_0)) \approx \mathsf{v}(I_1-1, a_0+I_1)\ [I_0 = x \wedge x > 0 \wedge I_1 = x \wedge a_0 = 0]$. We can use `SIMPLIFICATION` to move the first occurrence of $I_1 - 1$ into the constraint. Since $I_1 \in \mathcal{V}ar_{\mathtt{keep}}$, Definition 3 applies, so we can use a fresh variable in $\mathcal{V}ar_{\mathtt{drop}}$. This yields:

$$\mathsf{add}(I_0, \mathsf{v}(a_1, a_0)) \approx \mathsf{v}(I_1-1, a_0+I_1) \quad [I_0 = x \wedge x > 0 \wedge I_1 = x \wedge a_0 = 0 \wedge a_1 = I_1 - 1]$$

As $a_1 = I_1 - 1$ already occurs in the constraint, we do *not* introduce a fresh variable for the occurrence of $I_1 - 1$ on the right-hand, but rather follow the strategy of [2] and simplify this directly to a_1. We also simplify $a_0 + I_1$, but since $a_0 \notin \mathcal{V}ar_{\mathtt{keep}}$, we here do not introduce an initialization variable. We obtain:

$$\mathsf{add}(I_0, \mathsf{v}(a_1, a_0)) \approx \mathsf{v}(a_1, z_1) \quad [I_0 = x \wedge x > 0 \wedge I_1 = x \wedge a_0 = 0 \wedge a_1 = I_1 - 1 \wedge z_1 = a_0 + I_1]$$

If we drop all initializations $a_i = r_i$ with $a_i \in \mathcal{V}ar_{\mathtt{drop}}$ and $r_i \in \mathcal{T}erms(\Sigma_{theory}, \mathcal{V}ar_{\mathtt{keep}})$ from the constraint, we obtain the following equation, which is equivalent to lemma 1, so can be used to complete Example 8.

$$\mathsf{add}(I_0, \mathsf{v}(a_1, a_0)) \approx \mathsf{v}(a_1, z_1) \quad [I_0 = x \wedge x > 0 \wedge I_1 = x \wedge z_1 = a_0 + I_1]$$

4.3 Extending the rewrite system

We now know that $\mathsf{sum1}(x) \approx \mathsf{sum2}(x)$ and $\mathsf{sum2}(x) \approx \mathsf{sum3}(x)$ are inductive theorems and therefore by transitivity $\mathsf{sum1}(x) \approx \mathsf{sum3}(x)$ must be an inductive theorem as well. However, proving this directly using rewriting induction is not so straightforward. We obtain the following divergence:

$$\mathcal{E} = \{\mathsf{u}(x, 6, 15) \approx \mathsf{v}(x_5, z_4) \quad [x_5 = x_4 - 1 \wedge z_4 = z_3 + x_4 \wedge x > 4]\}$$

$$\mathcal{H} = \left\{ \begin{array}{ll} \mathsf{v}(x, 0) \to \mathsf{u}(x, 1, 0), & \\ \mathsf{v}(x_1, x) \to \mathsf{u}(x, 2, 1) & [x_1 = x - 1 \wedge x > 0], \\ \mathsf{v}(x_2, z_1) \to \mathsf{u}(x, 3, 3) & [x_2 = x - 2 \wedge z_1 = 2x - 1 \wedge x > 1], \\ \mathsf{v}(x_3, z_2) \to \mathsf{u}(x, 4, 6) & [x_3 = x - 3 \wedge z_2 = 3x - 3 \wedge x > 2], \\ \mathsf{v}(x_4, z_3) \to \mathsf{u}(x, 5, 10) & [x_4 = x - 4 \wedge z_3 = 4x - 6 \wedge x > 3] \end{array} \right\}$$

Observe that the divergence pattern contains both u and v, symbols unique to the LCTRS representation of sum1 and sum3 respectively, as we were not able to apply any of the induction hypotheses. When using InGen, we obtain an equation that cannot be automatically solved:

$$\mathsf{u}(x, i_1, z_1) \approx \mathsf{v}(x-1, z_1') \quad [i_1 = i_0 + 1 \wedge z_1 = z_0 + i_0 \wedge z_1' = x + z_0']$$

Dropping initializations earlier/later, or usage of the methods defined in this paper does not seem to help either. This problem arises in many examples when we compare two tail-recursive functions (obtained from iteration in the original C-code). The generalization method of [2], and our extensions in the previous sections, are primarily useful to compare an iterative function to a recursive one.

This does not mean that rewriting induction cannot be used. We solve the problem using the lemma:

Lemma 2 $\mathsf{u}(x,i,z) \approx \mathsf{v}(x',z') \quad [i+x' = x+1 \wedge z-z' = (i-1)(x'-i-1) \wedge x,i > 0 \wedge x' \geq 0]$

However, this lemma is very specific to the current problem of summation, and hard to find automatically. We aim to develop general methods. Hence, instead of trying to reproduce this lemma we will show that we can generate a lemma of a different shape:

Lemma 3 $\mathsf{v}(x,z) \approx \mathsf{add}(a,\mathsf{v}(x,z')) \quad [z = z'+a \wedge z' \geq 0 \wedge a > 0]$

This lemma is formulated in an *extended* LCTRS, where a symbol add is added to Σ_{terms}. With this lemma, we can indeed prove that $\mathsf{sum1}(x) \approx \mathsf{sum3}(x)$: the proof of this lemma initially diverges, but the divergence pattern contains symbols from only sum1, which allows us to solve it using InGen.

But how to find this lemma? And how would we know to include add and the corresponding rule, if we did not know about sum2? To answer these questions, we consider the rules of the LCTRS.

The second component z of v acts as an accumulator: the recursive v-rule has a form $\mathsf{v}(x,z) \to \mathsf{v}(s,z+x)$ where z does not occur in s, and the base rule returns z. Hence, we *externalize* the addition, and consider a lemma of the form $\mathsf{add}(x,\mathsf{v}(i,z)) \approx \mathsf{v}(i,z+x)$ with $\mathsf{add}(x,\mathsf{return}(z)) \to \mathsf{return}(z+x)$. Lemma 3 is obtained from this by adding constraints to support termination. Slightly more generally:

Definition 4 Suppose $\mathsf{f} \in \Sigma_{terms}$ is defined by two rules: (1) $\mathsf{f}(\vec{x},y,\vec{z}) \to \mathsf{f}(\vec{s},q,\vec{t})$ $[\varphi]$ and (2) $\mathsf{f}(\vec{x},y,\vec{z}) \to \mathsf{g}(\vec{s'},u,\vec{t'})$ $[\psi]$, with $y \notin \mathcal{V}ar(\vec{s},\vec{t},\vec{s'},\vec{t'})$ and $\mathsf{f} > \mathsf{g}$ following Def. 1. Then for h a fresh symbol, introduce the rule $\mathsf{h}(\vec{x},\mathsf{g}(\vec{x'},y,\vec{z'}),\vec{z}) \to \mathsf{g}(\vec{x'},q,\vec{z'})$ and consider a lemma of the form $\mathsf{h}(\vec{x},\mathsf{f}(\vec{x''},y,\vec{z''}),\vec{z}) \approx \mathsf{f}(\vec{x''},q,\vec{z''})$.

(In our example, $\mathsf{f} := \mathsf{v}$, $\mathsf{g} := \mathsf{return}$, $\mathsf{h} := \mathsf{add}$, $|\vec{x}| = |\vec{x''}| = 1$ and $|\vec{z}| = |\vec{x'}| = |\vec{z'}| = |\vec{z''}| = 0$.)

This approach is most likely to work when AC operators like $+$ and $*$ are involved, but also works for some others (e.g., $-$). As this is still work in progress, we have not yet explored the power of this method, and how best to formulate Def. 4 in a general way (e.g., to support symbols f with more than two cases).

Finding an intermediate function Another idea is that, if it is hard to prove $f(\vec{x}) \approx g(\vec{x})$ $[\varphi]$, then we look for an intermediate function h for which it is easier to prove equivalence, i.e. $f(\vec{x}) \approx h(\vec{x})$ $[\varphi]$ and $h(\vec{x}) \approx g(\vec{x})$ $[\varphi]$. (In our example, sum2 is such an intermediate function between sum1 and sum3). Then we either complete by transitivity, or use these two equivalences as lemma equations.

To find such an intermediate function, we could for instance use a similar approach as we did in Definition 4: if we detect that a parameter to a function symbol is used as an accumulator, we could try to remove that parameter and, essentially, replace iteration by recursion. For example, in place of the v-rules we would generate $\mathsf{v'}(x,z) \to \mathsf{add}(\mathsf{v'}(x-1,z),x)$ $[x > 0]$ and $\mathsf{v'}(x,z) \to \mathsf{return}(z)$ $[x \leq 0]$. Another approach may be to use a tool that modifies a function in the source language while preserving its input-output-behaviour. We can then try to prove equivalence for the modified version(s).

5 Conclusion

In this paper, we have shared some of our early findings on automatic lemma generation extending the method of [2] using three implementations of $\Sigma_{i=1}^{n} i$. In particular, we have seen that dropping a constraint from an equation may sometimes give a suitable lemma. We have given a definition that allows us to recognise certain constraints that may be useful to drop: derived initializations. Finally, we have presented sketches of possible lemma generation techniques that extend the signature and rewrite system.

Discussion An issue already present in [2] is that the strategy is not fail-save, and may require (costly) backtracking or fail altogether even when a proof exists. For example, dropping value initializations with InGen can sometimes be done before the first `EXPANSION`, but in other cases (like $\mathsf{sum1}(x) \approx \mathsf{sum2}(x)$) must be done later. In general, it is safe to drop them once we get into a repeating pattern – but it is not obvious how to detect such a pattern automatically. In addition, choices such as which side of a rule to apply an `EXPANSION` on, or which induction rule to use, can make a large difference.

As this is work in progress, our proposed lemmas were found manually. Lemma 2 in section 4.3 was found by solving a recurrence relation. In general, we will refrain from using this technique, since solving all recurrence relations is impossible and we prefer not to limit our method to linear arithmetic. For our other methods, it seems likely that they can be generalized and automated. However, our verification so far has focused on very simple examples, like the sum-examples in this paper. While we believe that these examples provide a good foundation, and that the ideas are more generally applicable, it is possible that the techniques are too tailored, and do not work in more complex cases.

Future Work To start, we intend to fully formalize the methods presented in this paper, and test them on a wide range of (both simple and more complex) benchmarks. Most of the calculations in this paper were done by hand, which is error-prone and time-intensive. Hence, we also intend to develop a new rewriting induction tool for LCTRSs, both to support the manual rewriting induction process and test new ideas, and to automate our new lemma generation techniques and strategies for rewriting induction.

Another question is how to choose which lemma generation technique to apply, and on which rules, or which side of an equation? For this, one could look at properties of the LCTRS: is there recursion or tail-recursion? Is there single or double recursion? How many 'base cases' are there? How many variables are in the recursion and how do they decrease?

Finally, we would like to explore different contexts for this methodology, for example for non-terminating programs / term rewriting systems (such as ongoing processes).

Acknowledgements We thank Femke van Raamsdonk and Jörg Endrullis for help and supervision.

References

[1] C. Kop and N. Nishida, "Term rewriting with logical constraints," in *Proc. FroCoS*, pp. 343–358, 2013.

[2] C. Fuhs, C. Kop, and N. Nishida, "Verifying procedural programs via constrained rewriting induction," *TOCL*, vol. 18, no. 2, pp. 1–50, 2017.

[3] A. Bundy, D. Basin, D. Hutter, and A. Ireland, *Rippling: meta-level guidance for mathematical reasoning*, vol. 56. Cambridge University Press, 2005.

[4] D. Kapur and M. Subramaniam, "Lemma discovery in automating induction," in *Proc. CADE*, pp. 538–552, 1996.

[5] D. Kapur and N. A. Sakhanenko, "Automatic generation of generalization lemmas for proving properties of tail-recursive definitions," in *Proc. TPHOLs*, pp. 136–154, 2003.

[6] N. Nakabayashi, N. Nishida, K. Kusakari, T. Sakabe, and M. Sakai, "Lemma generation method in rewriting induction for constrained term rewriting systems," *Computer Software*, vol. 28, no. 1, pp. 173–189, 2010.

[7] P. Urso and E. Kounalis, "Sound generalizations in mathematical induction," *TCS*, vol. 323, no. 1-3, pp. 443–471, 2004.

[8] T. Walsh, "A divergence critic for inductive proof," *JAIR*, vol. 4, pp. 209–235, 1996.