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 \( x \mapsto \sum_{i=1}^{x} i \).

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

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

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

(a)  
(b)  
(c)

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

We briefly introduce LCTRSs and a simplified version of Rewriting Induction as defined in [J][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 \( f : [t_1 \times \cdots \times t_n] \Rightarrow \kappa \), indicating that \( f \) takes \( n \) arguments of sorts \( t_1, \ldots, t_n \) respectively, and \( f(s_1, \ldots, s_n) \) has type \( \kappa \). We denote \( \terms{\Sigma, \var{Var}} \) for the set of (well-sorted) terms built from symbols in \( \Sigma \) and variables in \( \var{Var} \). In this paper, we usually let the set of sorts be \( \{ \text{bool}, \text{int}, \text{unit} \} \), and use symbols \( x, y, z, x_1, y_1, z_1 \) for variables.

We assume given two many-sorted signatures \( \Sigma_{\text{terms}} \) and \( \Sigma_{\text{theory}} \). Elements of \( \terms{\Sigma_{\text{theory}}, \var{Var}} \) are called logical terms: intuitively, these are terms like \( x + y \) or \( (14 \times x - 12 > y) \land (z \neq 0) \), which have a meaning in some underlying theory. For each sort \( t \) occurring in \( \Sigma_{\text{theory}} \), called theory sort, we assume given a corresponding set \( \var{I}_t \); and we assume given a map \( \var{J} \) which assigns to every function symbol \( f : [t_1 \times \cdots \times t_n] \Rightarrow \kappa \) in \( \Sigma_{\text{theory}} \) a function from \( \var{I}_{t_1} \times \cdots \times \var{I}_{t_n} \) to \( \var{I}_\kappa \). For example, we could map a symbol \( + : [\text{int} \times \text{int}] \Rightarrow \text{int} \) to the plus operator in the theory of the integers, or \( \land \) to logical conjunction. For every theory sort \( t \), we let the set \( \var{Val}_t \) of values of type \( t \) contain exactly the theory symbols of type \( t \); i.e., \( \var{Val}_t := \{ f \in \Sigma_{\text{theory}} \mid f : t \} \). We require that \( \var{J} \) is a bijection between \( \var{Val}_t \) and \( \var{I}_t \). Let \( \var{Val} \) be the set of all values in any \( \var{Val}_t \). The symbols in \( \Sigma_{\text{theory}} \) that are not values are called calculation symbols.

For ground logical terms, define \( \llbracket f(s_1, \ldots, s_n) \rrbracket = \var{J}(f)(\llbracket s_1 \rrbracket, \ldots, \llbracket s_n \rrbracket) \) and \( \llbracket s \rrbracket = \var{J}(s) \) if \( s \in \var{Val} \). A constraint is a logical term \( c : \text{bool} \), where \( \var{I}_{\text{bool}} = B = \{ \top, \bot \} \). A constraint \( c \) is valid if \( \llbracket c \rrbracket = \top \) for all substitutions \( \gamma \) that map all the variables in \( c \) to values. A substitution \( \gamma \) respects a constraint \( \varphi \) if all variables in \( \varphi \) are substituted by values and \( \llbracket \varphi \gamma \rrbracket = \top \). We typically assume \( \Sigma_{\text{theory}} \supseteq \Sigma_{\text{core}} \), where \( 
abla \Sigma_{\text{theory}} \) contains \text{true}, \text{false} : \text{bool}, \lor, \land : [\text{bool} \times \text{bool}] \Rightarrow \text{bool}, \neg : [\text{bool}] \Rightarrow \text{bool} \). We use infix notation for the binary operators in \( \Sigma_{\text{theory}} \).

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

A rewriting rule is a triple \( \ell \rightarrow r [\varphi] \), where \( \ell \) and \( r \) are terms and \( \varphi \) a constraint. We write \( \ell \rightarrow r \) if \( \varphi = \text{true} \). The root of \( \ell \) must be in \( \Sigma_{\text{terms}} \backslash \Sigma_{\text{theory}} \) and \( \ell \) and \( r \) must have the same sort. Note that it is allowed to have variables in \( r \) which do not occur in \( \ell \) (which we will use extensively in our method). The set of logic variables of a rule \( \ell \rightarrow r [\varphi] \), notation \( \var{LVar}(\ell \rightarrow r [\varphi]) \), contains all variables in \( \var{Val}(\varphi) \) and all variables in \( \var{Val}(\var{r}) \backslash \var{Val}(\ell) \). Define \( \var{R}_{\text{calc}} \) to be \( \{ f(x) \rightarrow y \mid f(x) = y) \mid f \in \Sigma_{\text{theory}} \} \backslash \var{Val} \). Given a set of rules \( \var{R} \), the rewrite relation \( \rightarrow_R \) is defined by: \( \var{C}(\ell) \rightarrow_R \var{C}(r) \) if \( \ell \rightarrow r [\varphi] \in (\var{R} \cup \var{R}_{\text{calc}}) \), \( \llbracket \varphi \rrbracket = \top \) and \( \gamma(x) \in \var{Val} \) for all \( x \in \var{LVar}(\ell \rightarrow r [\varphi]) \). That is, to apply a rule, the logic variables must be instantiated with values, and the constraint satisfied. For a rule \( f(l_1, \ldots, l_n) \rightarrow r [\varphi] \in \var{R} \) we call \( f \) a defined symbol. Let \( \var{D} \) be the set of defined symbols; symbols in \( \Sigma \backslash \var{D} \) are called constructors. A logically constrained term rewriting system (LCTRS) is the abstract rewriting system \( (\terms{\Sigma, \var{Var}}, \rightarrow_R) \).

Example 2 Simple programs may be represented using an LCTS (see, e.g., [2] for a translation method from simple C code). For example, the function from figure [15] may be represented by the LCTRS where \( \Sigma_{\text{theory}} = \Sigma_{\text{core}} \cup \{ +, \geq, < \} \cup \{ n \mid n \in \mathbb{Z} \} \) (with types and \( \var{J} \) as expected; \( \var{I}_{\text{int}} = \mathbb{Z} \)), and \( \Sigma_{\text{terms}} \) contains
A substitution versions of the most fundamental deduction rules from \([2]\). We do this by example, using the LCTRS 1. We return unit and \( : \text{int} \times \text{int} \times \text{int} \Rightarrow \text{unit} \), and \( \mathcal{R} \) consists of the rules:

\[
\begin{align*}
(\text{R1}) \quad & \text{sum1}(x) \rightarrow u(x, 1, 0) \\
(\text{R2}) \quad & u(x, i, z) \rightarrow u(x, i+1, z+i) \quad [i \leq x] \\
(\text{R3}) \quad & u(x, i, z) \rightarrow \text{return}(z) \quad [i > x]
\end{align*}
\]

Here, \( \text{LVar}(\text{R1}) = \emptyset \) and \( \text{LVar}(\text{R2}) = \text{LVar}(\text{R3}) = \{ x, i \} \) (the variables occurring in the constraints); this means that to apply these rules, \( x \) and \( i \) must be instantiated by values. Hence, \( \text{sum1}(1+1) \rightarrow_\mathcal{R} u(1+1, 1, 0) \) by rule (R1), but this term cannot be reduced by (R2). We can, however, reduce it using the calculation rule \( x+y \rightarrow z \ [x+y = z] \). Hence, \( u(1+1, 1, 0) \rightarrow_\mathcal{R} u(2, 1, 0) \rightarrow_\mathcal{R} u(2, 1+1, 0+1) \rightarrow_\mathcal{R} u(2, 2, 1) \rightarrow_\mathcal{R} u(2, 2+1, 2+1) \rightarrow_\mathcal{R} u(2, 3, 3) \rightarrow_\mathcal{R} \text{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., \( f(x) [x \geq 0] \equiv f(z) [2-3 < z] \). We will write \( s [\varphi] \rightarrow_\mathcal{R} t [\psi] \) if each instance of \( s \) that satisfies \( \varphi \) reduces in one step to an instance of \( t \) that satisfies \( \psi \), and all such instances of \( t \) can be obtained in this way. Formally, for a constrained term \( s [\varphi] \) and rule \( \rho = \ell \rightarrow_\mathcal{R} \psi \), define \( s [\varphi] \rightarrow_\rho t [\psi] \) if for some context \( C \) and substitution \( \gamma \) we have \( s = C[\ell \gamma], t = C[r \gamma], \varphi \Rightarrow \psi \gamma \) is valid and for all \( x \in \text{LVar}(\rho) \) we have that \( \gamma(x) \) is either a value or a variable in \( \text{Var}(\varphi) \).

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

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

### 2.2 Rewriting Induction

**Restrictions.** We limit interest to LCTRSs \( \mathcal{R} \) where: (1) \( \Sigma_{\text{theory}} \supseteq \Sigma_{\text{core}}; \) (2) \( \mathcal{R} \) is terminating, i.e. there are no infinite reductions \( s_1 \rightarrow_\mathcal{R} s_2 \rightarrow_\mathcal{R} s_3 \rightarrow_\mathcal{R} \ldots \); (3) \( \mathcal{R} \) is quasi-reductive, i.e. every ground term is either a constructor term or reduces; and (4) every sort in \( \Sigma \) 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 \( \varphi \) a constraint. A substitution \( \gamma \) respects \( s \approx t [\varphi] \) if \( \gamma \) respects \( \varphi \) and \( \text{Var}(s) \cup \text{Var}(t) \subseteq \text{Dom}(\gamma) \). An equation \( s \approx t [\varphi] \) is an inductive theorem if \( s \gamma \leftrightarrow_\mathcal{R} t \gamma \) for all substitutions \( \gamma \) that respect this equation and which map all variables in \( s, t \) to ground constructor terms. Intuitively, \( f(\bar{x}) \approx g(\bar{x}) [\varphi] \) is an inductive theorem if \( f \) and \( g \) agree on all \( \bar{x} \) for which \( \varphi(\bar{x}) \) 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 \((E, H)\) where \( E \) is a set of equations and \( H \) is a set of induction hypotheses: rewriting rules which have been derived during the rewriting induction. A proofstate \((E_1, H_1)\) might be transformed into another proofstate \((E_2, H_2)\), notation \( (E_1, H_1) \vdash_{\mathcal{R}_1} (E_2, H_2) \), by using one of the deduction rules which we explain below. The transitive, reflexive closure of \( \vdash_{\mathcal{R}_1} \) is denoted by \( \vdash^*_{\mathcal{R}_1} \).

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

**Theorem 1** \([2]\) If \((E, H) \vdash^*_{\mathcal{R}_1} (\emptyset, H)\) for some set \( H \), then every equation in \( E \) is an inductive theorem.

For readability, we will often write \( E_1 \vdash_{\mathcal{R}_1} E_2 \), omitting the \( H \) 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 \( \mathcal{R} \).
Lemma Generation Methods for RI on LCTRSs

that contains rules (R1) · · · (R3) from example 2 and the following rules corresponding to Figure 1b:

(R4) \( \text{sum2}(x) \rightarrow \text{add}(x, \text{sum2}(x-1)) \) [\( x > 0 \)]
(R5) \( \text{sum2}(x) \rightarrow \text{return}(0) \) [\( x \leq 0 \)]

(R6) \( \text{add}(x, \text{return}(y)) \rightarrow \text{return}(x+y) \)

If we aim to derive that \( \text{sum1}(x) \approx \text{sum2}(x) \) is an inductive theorem, then according to Theorem 1 we need to show that there is a deduction sequence \((\mathcal{E}, \theta) \vdash^{*}_{\text{r1}} (\theta, \mathcal{H})\), where \( \{\text{sum1}(x) \approx \text{sum2}(x)\} \subseteq \mathcal{E} \).

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 \( \text{sum1}(x) \) with (R1) gives

\[
\left\{ \text{sum1}(x) \approx \text{sum2}(x) \right\} \vdash_{\text{r1}} \{ \text{u}(x, 1, 0) \approx \text{sum2}(x) \}
\]

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 \( s \in \text{s} \approx t \mid [\varphi] \) is the subject of an expansion and \( \mathcal{R} \cup \mathcal{H} \cup \{ s \rightarrow t \mid [\varphi] \} \) is terminating, then the rewrite rule \( s \rightarrow t \mid [\varphi] \) is added to \( \mathcal{H} \).

If \( t \) is the subject of expansion, then \( t \rightarrow s \mid [\varphi] \) is added instead (if \( \mathcal{R} \cup \mathcal{H} \cup \{ t \rightarrow s \mid [\varphi] \} \) is terminating).

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

\[
\left\{ \text{u}(x, 1, 0) \approx \text{sum2}(x) \right\} \vdash_{\text{r1}} \left\{ \begin{array}{l}
\text{u}(x, 1, 0) \approx \text{add}(x, \text{sum2}(x-1)) \mid [x > 0], \\
\text{u}(x, 1, 0) \approx \text{return}(0) \mid [x \leq 0]
\end{array} \right\}
\]

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

\[
\left\{ \begin{array}{l}
\text{u}(x, 1, 0) \approx \text{add}(x, \text{sum2}(x-1)) \mid [x > 0], \\
\text{u}(x, 1, 0) \approx \text{return}(0) \mid [x \leq 0]
\end{array} \right\} \vdash_{\text{r1}} \left\{ \begin{array}{l}
\text{u}(x, 2, 1) \approx \text{add}(x, \text{sum2}(x-1)) \mid [x > 0], \\
\text{return}(0) \approx \text{return}(0) \mid [x \leq 0]
\end{array} \right\}
\]

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

\[
\left\{ \begin{array}{l}
\text{u}(x, 2, 1) \approx \text{add}(x, \text{sum2}(x-1)) \mid [x > 0], \\
\text{return}(0) \approx \text{return}(0) \mid [x \leq 0]
\end{array} \right\} \vdash_{\text{r1}} \left\{ \begin{array}{l}
\text{u}(x, 2, 1) \approx \text{add}(x, \text{u}(x-1, 0)) \mid [x > 0], \\
\text{return}(0) \approx \text{return}(0) \mid [x \leq 0]
\end{array} \right\}
\]

DELETION. An equation \( s \approx t \mid [\varphi] \) 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 \( \text{u}(x-1, 1, 0) \). 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:

\[
\left\{ \text{u}(x, 2, 1) \approx \text{add}(x, \text{u}(x-1, 1, 0)) \mid [x > 0] \right\} \vdash_{\text{r1}} \left\{ \begin{array}{l}
\text{u}(x, 2, 1) \approx \text{add}(x, \text{u}(x_1, 1, 0)) \mid [x > 0 \land x_1 = x-1]
\end{array} \right\}
\]

At this point we do another EXPANSION step, now on the left side of the equation. This causes (H2) \( \text{u}(x, 2, 1) \rightarrow \text{add}(x, \text{u}(x_1, 1, 0)) \mid [x > 0 \land x_1 = x-1] \) 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).

\[
\vdash_{\text{r1}} \left\{ \begin{array}{l}
\text{u}(x, 2+1, 1+2) \approx \text{add}(x, \text{u}(x_1, 1, 0)) \mid [x > 1 \land x_1 = x-1], \\
\text{return}(1) \approx \text{add}(x, \text{u}(x_1, 1, 0)) \mid [x = 1 \land x_1 = x-1]
\end{array} \right\}
\]
We apply SIMPLIFICATION in the second equation on the right side with rules \((R3), (R6)\) and \(R_{calc}\).

\[
\begin{align*}
\{ \text{return}(1) &\approx \text{add}(x,u(x_1, 1, 0)) \ [x = 1 \land x_1 = x - 1] \} \\
\vdash_{\text{r}_1} \{ \text{return}(1) &\approx \text{add}(x, \text{return}(0)) \ [x = 1 \land x_1 = x - 1] \} \\
\vdash_{\text{r}_1} \{ \text{return}(1) &\approx \text{return}(x + 0) \ [x = 1] \} \vdash_{\text{r}_1} \{ \text{return}(1) = \text{return}(x) \ [x = 1] \}
\end{align*}
\]

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 \(\varphi\). 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 \(\text{return}(1) \approx \text{return}(x) [x = 1 \land \neg (x = 1)]\), to which DELETION can be applied. After some simplifications to the remaining equation, we are left with:

\[
u(x, 3, 3) \approx \text{add}(x,u(x_1, 2, 1)) \ [x > 1 \land x_1 = x - 1]
\]

**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:

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

\[
\mathcal{H} = \begin{cases}
\text{sum2}\,(x) \to u(x, 1, 0), \\
\text{u}(x, 2, 1) \to \text{add}(x, u(x_1, 1, 0)) \ [x > 0 \land x_1 = x - 1], \\
\text{u}(x, 3, 3) \to \text{add}(x, u(x_1, 2, 1)) \ [x > 1 \land x_1 = x - 1], \\
\text{u}(x, 4, 6) \to \text{add}(x, u(x_1, 3, 3)) \ [x > 2 \land x_1 = x - 1]
\end{cases}
\]

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

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

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 \( f(x) \rightarrow g(x + 2, 1) \) is replaced by \( f(x) \rightarrow g(x + 2, a_0) \) \[ a_0 = 1 \]

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

\[
(R1') \quad \text{sum1}(x) \rightarrow u(x, a_1, a_0) \quad [a_1 = 1 \land a_0 = 0]
\]

(The other rules remain the same.) Aiming once again to prove that \( \text{sum1}(x) \approx \text{sum2}(x) \) 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:

\[
u(x, i_2, z_2) \approx \text{add} \left(x, u(x_1, i_1, z_1)\right)
\]

\[
[i_2 = i_1 + 1 \land z_2 = z_1 + i_1 \land x_1 = x - 1 \land x \geq i_1 \land i_1 = a_1 + 1 \land z_1 = a_0 + a_1 \land x > 0 \land a_1 = 1 \land a_1 = 0]
\]

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

\[
u(x, i_2, z_2) \rightarrow \text{add} \left(x, u(x_1, i_1, z_1)\right)
\]

\[
[i_2 = i_1 + 1 \land z_2 = z_1 + i_1 \land x_1 = x - 1 \land x \geq i_1 \land i_1 = a_1 + 1 \land z_1 = a_1 + i_0 \land x > 0]
\]

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 respectively. 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 LCTRS representations of the programs given in figure 1b and figure 1c.
(R4) \( \text{sum}2(x) \rightarrow \text{return}(0) \) \([x \leq 0]\)  
(R5) \( \text{sum}2(x) \rightarrow \text{add} \left( x, \text{sum}2(x-1) \right) \) \([x > 0]\)  
(R6) \( \text{add} \left( x, \text{return}(y) \right) \rightarrow \text{return}(x+y) \)

Aiming to show that \( \text{sum}2(x) \approx \text{sum}3(x) \) is an inductive theorem using the strategy in Section 2.2, we observe the following divergence pattern (without superfluous constraints): 

\[ E = \{ v(x_3,z_4) \approx \text{add} \left( x, v(x_5,y_4) \right) \mid x_5 = x_4 - 1 \land z_4 = 5x - 10 \land y_4 = 4x - 10 \land x > 4 \} \]

\[ H = \{ \begin{array}{l}
\text{sum}2(x) \rightarrow v(x,0), \\
v(x_1,z_0) \rightarrow \text{add} \left( x, v(x_1,0) \right) \mid x_1 = x - 1 \land z_0 = x \land x > 0, \\
v(x_2,z_1) \rightarrow \text{add} \left( x, v(x_2,1) \right) \mid x_2 = x - 2 \land z_1 = 2x - 1 \land x > 1, \\
v(x_3,z_2) \rightarrow \text{add} \left( x, v(x_3,y_2) \right) \mid x_3 = x - 3 \land z_2 = 3x - 3 \land y_2 = 2x - 3 \land x > 2, \\
v(x_4,z_3) \rightarrow \text{add} \left( x, v(x_4,y_3) \right) \mid x_4 = x - 4 \land z_3 = 4x - 6 \land y_3 = 3x - 4 \land x > 3 \end{array} \}
\]

Using InGen, we obtain a rule \( (R7') \) \( \text{sum}3(x) \rightarrow v(x,y_0) \) \([y_0 = 0]\). Dropping the initialization \( y_0 = 0 \) before the second expansion yields an equation \( v(x_1,z_0) \approx \text{add} \left( x, v(x_1,y_0) \right) \mid z_0 = y_0 + x \land x_1 = x - 1 \land x > 0 \). 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 \( \text{sum}2(x) \approx \text{sum}3(x) \).¹

**Lemma 1**  
\[ v(x_1,z_0) \approx \text{add} \left( x, v(x_1,y_0) \right) \mid z_0 = y_0 + x \land x > 0 \]

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 \( \text{sum}2(x) \approx \text{sum}3(x) \) also leads to lemma ¹

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

\[ \text{Terms}_{\text{init}} = \{ r_i \in \text{Terms}(\Sigma_{\text{theory}}, \text{Var}) \mid f(\ell_1,\ldots,\ell_n) \rightarrow C(g(r_1,\ldots,r_m)) \mid [\varphi] \in \mathcal{R}, f > g, g \in \mathcal{D} \} \]

**Example 7** In example 6 \( \geq \) is the transitive reflexive closure of \( \text{sum}2 \geq \text{add} \geq \text{return}, \text{sum}3 \geq \text{v} \geq \text{return} \). Hence, \( \text{sum}2 > \text{add} > \text{return}, \text{sum}3 > \text{v} > \text{return} \). There is one initialization \( x \) in \( \text{sum}2(x) \rightarrow \text{add} \left( x, \text{sum}2(x-1) \right) \) \([x > 0]\), and initializations \( x \) and \( 0 \) in \( \text{sum}3(x) \rightarrow v(x,0) \).

**Definition 2** Given \( \mathcal{R} \), fix sets \( \text{Var}_{\text{drop}}, \text{Var}_{\text{keep}} \) of variables not occurring in \( \mathcal{R} \) such that \( \text{Var}_{\text{drop}} \cap \text{Var}_{\text{keep}} = \emptyset \). The initialization-free counterpart \( \mathcal{R}' \) of \( \mathcal{R} \) is obtained by stepwise replacing every rule \( \ell \rightarrow C[s] \mid [\varphi] \) in \( \mathcal{R} \), with \( s \in \text{Terms}_{\text{init}} \) by the following, until no such rules remain:

1. \( \ell \rightarrow C[a] \mid [\varphi \land a = s] \) with \( a \in \text{Var}_{\text{drop}} \) fresh, if \( s \in \text{Val} \).
2. \( \ell \rightarrow C[I] \mid [\varphi \land I = s] \) with \( I \in \text{Var}_{\text{keep}} \) fresh, if \( s \notin \text{Val} \).

¹If we follow the rules exactly, the lemma needs a bit more modification since a rule \( v(x_1,z_0) \rightarrow \text{add}(x,v(x_1,y_0)) \mid [z_0 = y_0 + x \land x > 0] \) 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') \( \text{sum2}(x) \rightarrow \text{add}(l_0, \text{sum2}(x - 1)) \) \( [x > 0 \land l_0 = x] \)

(R7') \( \text{sum3}(x) \rightarrow \nu(l_1, a_0) \) \( [l_1 = x \land a_0 = 0] \)

Performing the rewriting induction \( \text{sum2}(x) \approx \text{sum3}(x) \) with this altered LCTRS quickly yields a state with \( \mathcal{E} = \{ \text{add}(l_0, \nu(l_1 - 1, a_0)) \approx \nu(l_1 - 1, a_0 + l_1) \} \) \( [l_0 = x \land x > 0 \land l_1 = x \land a_0 = 0] \)

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 \( s \approx t \phi \in \mathcal{E} \), where \( s \) has the form \( C[r] \) with \( r \in \mathcal{T} \text{erms}(\Sigma_{\text{theory}}, \text{Var}_{\text{keep}}) \) and \( r \) not a variable. Then \( s \approx t \phi \) may be rewritten to \( C[a] \approx t \phi \land \text{a} \) for \( a \in \text{Var}_{\text{drop}} \) 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: \( \text{add}(l_0, \nu(l_1 - 1, a_0)) \approx \nu(l_1 - 1, a_0 + l_1) \) \( [l_0 = x \land x > 0 \land l_1 = x \land a_0 = 0 \land a_1 = l_1 - 1] \)

As \( a_1 = l_1 - 1 \) already occurs in the constraint, we do not introduce a fresh variable for the occurrence of \( l_1 - 1 \) on the right-hand, but rather follow the strategy of [24] and simplify this directly to \( a_1 \). We also simplify \( a_0 + l_1 \), but since \( a_0 \notin \text{Var}_{\text{keep}} \), we here do not introduce an initialization variable. We obtain:

\( \text{add}(l_0, \nu(a_1, a_0)) \approx \nu(a_1, z_1) \) \( [l_0 = x \land x > 0 \land l_1 = x \land a_0 = 0 \land a_1 = l_1 - 1 \land z_1 = a_0 + l_1] \)

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

\( \text{add}(l_0, \nu(a_1, a_0)) \approx \nu(a_1, z_1) \) \( [l_0 = x \land x > 0 \land l_1 = x \land z_1 = a_0 + l_1] \)

4.3 Extending the rewrite system

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

\[ \mathcal{E} = \{ u(x, 6, 15) \approx \nu(x_5, z_4) \} \]

\[ \begin{align*}
\nu(x, 0) &\rightarrow u(x, 1, 0), \\
\nu(x_1, x) &\rightarrow u(x, 2, 1) \quad [x_1 = x - 1 \land x > 0], \\
\nu(x_2, z_1) &\rightarrow u(x, 3, 3) \quad [x_2 = x - 2 \land z_1 = 2x - 1 \land x > 1], \\
\nu(x_3, z_2) &\rightarrow u(x, 4, 6) \quad [x_3 = x - 3 \land z_2 = 3x - 2 \land x > 2], \\
\nu(x_4, z_3) &\rightarrow u(x, 5, 10) \quad [x_4 = x - 4 \land z_3 = 4x - 3 \land x > 3]
\end{align*} \]

Observe that the divergence pattern contains both \( u \) and \( \nu \), symbols unique to the LCTRS representation of \( \text{sum1} \) and \( \text{sum3} \) 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:

\[ \nu(x, i_1, z_1) \approx \nu(x - 1, z_1') \] \( [i_1 = i_0 + 1 \land z_1 = z_0 + i_0 \land z_1' = x + z_0'] \)
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**

\[ u(x, i, z) \approx v(x', z') \quad [i + x' = x + 1 \land z - z' = (i - 1)(x' - i - 1) \land x, i > 0 \land x' \geq 0] \]

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**

\[ v(x, z) \approx \text{add}(a, v(x, z')) \quad [z = z' + a \land a > 0] \]

This lemma is formulated in an extended LCTRS, where a symbol add is added to \( \Sigma_{\text{terms}} \). With this lemma, we can indeed prove that \( \text{sum1}(x) \approx \text{sum3}(x) \): the proof of this lemma initially diverges, but the divergence pattern contains symbols from only \( \text{sum1} \), 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 \( \text{sum2} \)? 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 \( v(x, z) \rightarrow v(x, z + x) \) 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 \( \text{add}(x, v(i, z)) \approx v(i, z + x) \) with \( \text{add}(x, \text{return}(z)) \rightarrow \text{return}(z + x) \). Lemma 3 is obtained from this by adding constraints to support termination. Slightly more generally:

**Definition 4** Suppose \( f \in \Sigma_{\text{terms}} \) is defined by two rules: (1) \( f(\vec{x}, y, \vec{z}) \rightarrow f(\vec{x}, q, \vec{z}) [\varphi] \) and (2) \( f(\vec{x}, y, \vec{z}) \rightarrow g(\vec{y}^2, u, \vec{r}^2) [\psi] \), with \( y \notin \text{Var}(\vec{x}, \vec{z}, \vec{y}, \vec{r}) \) and \( f \geq g \) following Def. 1. Then for a fresh symbol, introduce the rule \( h(\vec{x}, g(\vec{x}', y, \vec{z}'), \vec{z}) \rightarrow g(\vec{x}', q, \vec{z}') \) and consider a lemma of the form \( h(\vec{x}, f(\vec{x}', y, \vec{z}'), \vec{z}) \approx f(\vec{x}', q, \vec{z}') \).

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

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, \( \text{sum2} \) is such an intermediate function between \( \text{sum1} \) and \( \text{sum3} \).) 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 Def. 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 \( \nabla'(x, z) \rightarrow \text{add}(\nabla'(x - 1, z), x) [x > 0] \) and \( \nabla'(x, z) \rightarrow \text{return}(z) [x \leq 0] \). 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 \( \Sigma_{\sum_{i=1}^{n}} \). 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 \( \text{sum1}(x) \approx \text{sum2}(x) \)) 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 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