Matrix invariants for program equivalence in LCTRSs

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 present a new technique for generating invariants, using matrix calculations, and show how it can be combined with an existing method by Fuhs, Kop and Nishida to produce a lemma.

1 Introduction

Given two algorithms, can we provide a reasoning that ensures they produce the same output for every possible input? This question is known as *program equivalence*. It is a challenging problem that naturally arises in software development. For example, it is common to apply transformations on programming code, e.g. for optimization purposes or to refactor in preparation for later updates [1]. In order to guarantee preservation of reliability and functionality, such transformations are required to retain equivalence.

In principle, with human reasoning, one could manually show that two programs are equivalent. For practical usage, however, this is not convenient: real programs are often so big that this would be a too time-consuming task which is, moreover, very prone to mistakes. Therefore, we would like to automate the process as much as possible. Unfortunately, program equivalence in general is undecidable. Hence, the best we can hope for is developing a large collection of methods that can be used in specific cases. Some of them can be found in the field of formal verification, semantics, and logic [2].

The approach we will follow in this paper is based on term rewriting: a mathematical discipline that studies the step-by-step transformation of objects, called terms, by the application of rewrite rules. A term can, for example, represent the state of an algorithm on some particular instance during its execution. The application of a rewrite rule can then be thought of as a calculation step in the algorithm.

Programs can be automatically converted into Logically Constrained Term Rewriting Systems (LCTRSs) [3, 4], after which equivalence is shown using a proof system called Rewriting induction (RI). It is often possible to automatically find a proof, using various strategies and lemma generation techniques. Some lemma generation methods for LCTRSs were introduced in [3], most importantly *Initialization Generalization (InGen)*, which works fine in many cases but also has its limitations.

This paper proposes a new method for generating invariants. We show how these invariants can in turn be combined with SMT and InGen in cases where InGen on itself is unable to produce a lemma.

Related Work Related work potentially includes everything that discusses the generation of invariants on programs. One existing method uses widening operators [5, 6]. Another method is based on o-minimal invariants [7] on discrete dynamical systems. Perhaps the most related to our idea of matrix invariants is a method based on linear algebra [8, 9]. We still need to investigate how exactly these methods relate to our ideas and whether we can use them in our setting.

Submitted to:
WPTE 2023

2 Preliminaries

We assume familiarity with basic notions of many-sorted term rewriting, such as function symbols, variables, terms, substitutions and contexts. In this section, we briefly introduce LCTRSs and a simplified version of Rewriting Induction (RI) as defined for LCTRSs in [10, 3].

Logically Constrained Term Rewriting Assume a set of *sorts* (most commonly $\{\mathsf{bool}, \mathsf{int}, \mathsf{unit}\}$), a set $\mathcal{V}ar$ of sorted variables (notation $x : \iota$), and a set Σ of *function symbols*, each equipped with a *sort declaration* $[\iota_1 \times \cdots \times \iota_k] \Rightarrow \kappa$, indicating that f takes k arguments of sorts $\iota_1, \ldots, \iota_k$, and $\mathsf{f}(s_1, \ldots, s_k)$ has type κ. Let $\mathcal{T}erms(\Sigma, \mathcal{V}ar)$ be the set of well-sorted terms built from Σ and $\mathcal{V}ar$. We typically denote x, y, z, x_i, y_i, z_i for variables and $\mathsf{f}, \mathsf{g}, \mathsf{u}, \mathsf{v}$ or more suggestive notation for function symbols.

We assume $\Sigma = \Sigma_{terms} \cup \Sigma_{theory}$ where Σ_{theory} contains symbols such as $+, >, \wedge, 3, -50$ that have a *meaning* in some underlying theory, whereas Σ_{terms} contains symbols which behavior we want to define by the rewriting system. Each sort ι occurring in Σ_{theory}, called *theory sort*, is interpreted as set $\mathcal{I}_\iota$. For each $\mathsf{f} : [\iota_1 \times \cdots \times \iota_k] \Rightarrow \kappa$ in Σ_{theory} there is an interpretation function $\mathcal{J}(\mathsf{f}) : \mathcal{I}_{\iota_1} \times \cdots \times \mathcal{I}_{\iota_k} \to \mathcal{I}_\kappa$. For example, we usually define $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$ and $\mathcal{J}(+) : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ is usually defined as addition on $\mathbb{Z}$. Elements of $\mathcal{T}erms(\Sigma_{theory}, \mathcal{V}ar)$ are called *logical terms*. For every theory sort ι, define $\mathcal{V}al_\iota := \{\mathsf{f} \in \Sigma_{theory} \mid \mathsf{f} : \iota\}$ to be the set of values of sort ι. We require that $\mathcal{J}(f)$ is a bijection between $\mathcal{V}al_\iota$ and $\mathcal{I}_\iota$, for all $\mathsf{f} : \iota$. Let $\mathcal{V}al = \cup_\iota \mathcal{V}al_\iota$ be the set of all values. Symbols in $\Sigma_{theory} \setminus \mathcal{V}al$ 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 mapped to 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 $\mathcal{J}$ interprets all function symbols as expected. We use infix notation for the binary operators in Σ_{theory}.

Example 1 Consider $\Sigma_{terms} = \{\mathsf{max} : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int},\ \mathsf{return} : [\mathsf{int}] \Rightarrow \mathsf{bool}\}$ and $\Sigma_{theory} = \Sigma^{core}_{theory} \cup \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\}$. Let $\mathcal{I}_{\mathsf{bool}} = \mathbb{B}$, $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$ and $\mathcal{J}$ the function that interprets all symbols in Σ_{theory} as expected. Then $\mathsf{false} : \mathsf{bool}$, $\mathsf{true} : \mathsf{bool}$ and all $\mathsf{n} : \mathsf{int}$ are values. Examples of logical terms include $t_1 = \mathsf{true}$, $t_2 = \neg(x \wedge y)$, $t_3 = x \vee \neg x$, $t_4 = \mathsf{true} \vee \mathsf{false}$. The interpretation of t_4 is $[\![t_4]\!] = (\top$ "or" $\bot) = \top$. All t_i are constraints, but only 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 $\ell \to r\ [\varphi]$, with ℓ and r terms and φ a constraint. We write $\ell \to r$ if $\varphi = \mathsf{true}$. The root of ℓ must be in $\Sigma_{terms} \setminus \Sigma_{theory}$ and ℓ and r must have the same sort. Note that it is allowed to have variables in r, not occurring in ℓ. The set of *logic variables* of a rule $\ell \to r\ [\varphi]$, notation $LVar(\ell \to r\ [\varphi])$, is given by $\mathcal{V}ar(\varphi) \cup (\mathcal{V}ar(r) \setminus \mathcal{V}ar(\ell))$. Define $\mathcal{R}_{\mathtt{calc}} = \{f(\vec{x}) \to y\ [f(\vec{x}) = y] \mid f \in \Sigma_{theory} \setminus \mathcal{V}al\}$.

For 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 logical variables must be instantiated with values, and the constraint must be satisfied. For any $f(l_1, \ldots, l_n) \to r\ [\varphi] \in \mathcal{R}$ we call f a *defined symbol*. $\mathcal{D}$ is the set of defined symbols; symbols in $\mathcal{V}al \cup (\Sigma_{terms} \setminus \mathcal{D})$ are *constructors*. A *logically constrained term rewriting system* (LCTRS) is the abstract rewriting system $(\mathcal{T}erms(\Sigma, \mathcal{V}ar), \to_{\mathcal{R}})$.

Constrained Terms To handle program equivalence, we need to rewrite *constrained terms*. Essentially, a constrained term $s\ [\varphi]$ represents the set of terms $s\gamma$ such that $\varphi\gamma$ is valid. Two constrained terms are *equivalent* if they represent the same set; e.g., $\mathsf{f}(x)\ [x \geq 0]$ is equivalent to $\mathsf{f}(z)\ [2 - 3 < z]$. We 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 some substitution γ we have $s = C[\ell\gamma]$, $t = C[r\gamma]$, $\varphi \Rightarrow \psi\gamma$ is valid and for all $x \in LVar(\rho)$ we have $\gamma(x) \in \mathcal{V}al \cup \mathcal{V}ar(\varphi)$. The rewrite relation on constrained terms is modulo term equivalence. For example, $\mathsf{f}(a, a+1)\ [a > 0]$ is rewritten into $\mathsf{f}(a,b)\ [a > 0 \wedge b = a+1]$ because $\mathsf{f}(a,a+1)\ [a > 0]$ 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 $[x := a, y := 1, z := b]$.

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 $\mathcal{V}ar(s) \cup \mathcal{V}ar(t) \subseteq Dom(\gamma)$. 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 proof states. A *proof state* 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 proof state $(\mathcal{E}_1, \mathcal{H}_1)$ might be transformed into another proof state $(\mathcal{E}_2, \mathcal{H}_2)$, 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 in Section 3. The transitive, reflexive closure of $\vdash_{\mathtt{ri}}$ is denoted by $\vdash^*_{\mathtt{ri}}$.

Theorem 1 [3] *If $(\mathcal{E}, \emptyset) \vdash^*_{ri} (\emptyset, \mathcal{H})$ for some set $\mathcal{H}$, then all equations in $\mathcal{E}$ are inductive theorems.*

Validness of this theorem is, however, only guaranteed if we satisfy the following requirements: (1) $\Sigma_{theory} \supseteq \Sigma^{core}_{theory}$; (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 a constructor term or reduces; and (4) every sort in Σ has at least one ground term.

3 Program Equivalence by LCTRSs

Let us demonstrate how LCTRSs can be used for program equivalence. Figure 1 shows three implementations of $sum(x) = \sum_{i=1}^{x} i$. We use rewriting induction to prove their equivalence.

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

(c)

Figure 1: Three equivalent implementations of $sum(x) = \sum_{i=1}^{x} i$.

The code in Figure 1a is represented as an LCTRS with $\Sigma_{theory} = \Sigma^{core}_{theory} \cup \{+, \geq, <\} \cup \{\mathsf{n} \mid n \in \mathbb{Z}\}$ and $\Sigma_{terms} = \{\mathsf{sum1}, \mathsf{return} : [\mathsf{int}] \Rightarrow \mathsf{unit}, \mathsf{u} : [\mathsf{int} \times \mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{unit}\}$ and rewrite rules $\mathcal{R}_{\mathsf{sum1}}$, consisting of

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

Similarly, we can represent the function in Figure 1b by an LCTRS $\mathcal{R}_{\mathsf{sum2}}$, consisting of rules

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

Rewriting Induction The equivalence of `sum1` and `sum2` is shown by proving that $\mathsf{sum1}(x) \approx \mathsf{sum2}(x)$ is an inductive theorem. By Theorem 1, it suffices to show that there is deduction $(\mathcal{E},\emptyset) \vdash^{*}_{\mathtt{ri}} (\emptyset,\mathcal{H})$, for some sets $\mathcal{H}$ and $\mathcal{E}$ with $\{\mathsf{sum1}(x) \approx \mathsf{sum2}(x)\} \subseteq \mathcal{E}$. We demonstrate how to derive this by simultaneously explaining a simplified version of rewriting induction.

Let $\mathcal{R} = \mathcal{R}_{\mathsf{sum1}} \cup \mathcal{R}_{\mathsf{sum2}}$ and start the rewriting induction with proof state $(\mathcal{E},\mathcal{H}) := (\{\mathsf{sum1}(x) \approx \mathsf{sum2}(x)\},\emptyset)$. For readability, we will write a deduction step by $\mathcal{E}_1 \vdash_{\mathtt{ri}} \mathcal{E}_2$, omitting the $\mathcal{H}$ component.

`SIMPLIFICATION`. We may apply any rule from $\mathcal{R} \cup \mathcal{R}_{\mathtt{calc}} \cup \mathcal{H}$ to a subterm. For rewriting purposes, the equation is viewed as a single constrained term with a new function symbol $\approx$. `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 a subterm on one side of an equation $s \approx t\ [\varphi]$. For every rule $l \to r\ [\psi] \in \mathcal{R}$ this will try to find a substitution γ that matches l with the chosen subterm. For every successful case, a new equation is added where constraint $\psi\gamma$ is added to the constraint of the equation and `SIMPLIFICATION` is applied with the selected rule. Furthermore, if s is the subject of an expansion and $\mathcal{R} \cup \mathcal{R}_{\mathtt{calc}} \cup \mathcal{H} \cup \{s \to t\ [\varphi]\}$ 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 $\mathcal{R} \cup \mathcal{R}_{\mathtt{calc}} \cup \mathcal{H} \cup \{t \to s\ [\varphi]\}$ is terminating).

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

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

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

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

In addition, we can apply $(H1)$ to rewrite $\mathsf{sum2}(x-1)$.

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

`DELETION`. An equation $s \approx t\ [\varphi]$ may be deleted if $s = t$ or φ is unsatisfiable (that is, $[\![\varphi\gamma]\!] = \bot$ for any γ that maps the variables in φ to values). Following this rule, we may remove the second equation from the proof state. The remaining equation contains the subterm $\mathsf{u}(x-1,1,0)$. To support the application of rewrite rules, it is needed to replace a calculation (e.g. $x-1$) by a fresh variable (e.g. x'), where we update the constraint with a corresponding equality (e.g. $x' = x-1$). This can be done using the `SIMPLIFICATION` rule. In this case, replacing $x-1$ by x' yields:

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

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

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

We apply SIMPLIFICATION in the rhs of the second equation with $(R3)$, $(R6)$ and $\mathcal{R}_{\texttt{calc}}$ (for the moment, we do not denote the first equation, but it is of course still there).

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

In this last equation, we cannot apply DELETION, since the left and right side are not syntactically equal. Their equivalence is, however, implied via the constraint.

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 $\mathsf{return}(1) \approx \mathsf{return}(x)\ [x=1 \wedge \neg(x+0=1)]$ to which DELETION can be applied. After some few more simplifications, we obtain an equation $\mathsf{u}(x,3,3) \approx \mathsf{add}(x, \mathsf{u}(x',2,1))\ [x>1 \wedge x'=x-1]$

Automation. The rewriting induction process can be automated using a strategy: a priority selection on the deduction rules. The strategy used in [3] tries to apply deduction rules in the following order: EQ-DELETION, DELETION, SIMPLIFICATION, EXPANSION.

Divergence As often happens in practice, and also for $\mathsf{sum1}(x) \approx \mathsf{sum2}(x)$, we eventually keep expanding but we cannot apply any induction rule to the lhs. After some more steps we arrive at
$\mathcal{E} = \{\ \mathsf{u}(x,5,10) \approx \mathsf{add}(x, \mathsf{u}(x',4,6))\ [x>3 \wedge x'=x-1]\ \}$

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

It is clear that no induction hypothesis will ever be applicable to the left-hand side of the ongoing equation. What we encounter here is called *divergence*.

4 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 $\mathsf{sum1}(x) \approx \mathsf{sum2}(x)$, the recurring equation in $\mathcal{E}$ is always an instance of

$$(L): \quad \mathsf{u}(x,i_1,z_1) \approx \mathsf{add}\,(x, \mathsf{u}(x',i_0,z_0))$$
$$[i_1 = i_0+1 \wedge z_1 = z_0+i_0 \wedge x' = x-1 \wedge x \geq i_0 \wedge i_0 = i+1 \wedge z_0 = z+i \wedge x > 0]$$

It happens that (L) *is* an inductive theorem, provable using RI. The proof adds (L) to $\mathcal{H}$ as a rewriting rule, which in turn can be applied to solve the divergence. Hence, $(\{\mathsf{sum1}(x) \approx \mathsf{sum2}(x)\},\ \{(L)\}) \vdash^*_{\texttt{ri}} (\emptyset, \mathcal{H})$ for some $\mathcal{H}$, proving the equivalence. In this setting, we call (L) a *lemma*. Equalities/inequalities such as $i_1 = i_0 + 1$ and $x > 0$ are called *invariants* because they hold for every divergence rule in $\mathcal{H}$. The question remains how such a lemma can be found automatically. Much work has been done in this area (see e.g. [11, 12, 13, 14, 15, 16, 3]). Below we recall one of these methods called *initialization generalization*. [3]

Initialization generalization [3] Initialization generalization (InGen) adapts an LCTRS by assigning a fresh (red colored) variables to every value initialization. Using this altered (but equivalent) LCTRS, we start the RI and somewhere in this process the value assignments are removed. In the meantime, we obtained a bunch of equalities/inequalities involving these fresh variables that may give us a lemma.

Example 2 The InGen procedure will replace $(R1)$ $\mathsf{sum1}(x) \to \mathsf{u}(x,1,0)$ by $(R1')$ $\mathsf{sum1}(x) \to \mathsf{u}(x,i,z)$ $[i = 1 \wedge z = 0]$. Using $(R1')$ we start the rewriting induction on $\mathsf{sum1}(x) \approx \mathsf{sum2}(x)$ until we need to perform the second divergence-`EXPANSION`. At this point we have the following equation in $\mathcal{E}$:

$$\mathsf{u}(x,i_1,z_1) \approx \mathsf{add}\,(x,\mathsf{u}(x',i_0,z_0))$$
$$[i_1 = i_0 + 1 \wedge z_1 = z_0 + i_0 \wedge x' = x - 1 \wedge x \geq i_0 \wedge i_0 = i + 1 \wedge z_0 = z + i \wedge x > 0 \wedge i = 1 \wedge z = 0]$$

We drop the value initializations, followed by `EXPANSION` on the left. This adds the following rule to $\mathcal{H}$:

$$\mathsf{u}(x,i_1,z_1) \to \mathsf{add}\,(x,\mathsf{u}(x',i_0,z_0))$$
$$[i_1 = i_0 + 1 \wedge z_1 = z_0 + i_0 \wedge x' = x - 1 \wedge x \geq i_0 \wedge i_0 = i + 1 \wedge z_0 = z + i \wedge x > 0]$$

This induction rule *does* apply to an ongoing proof state, by which we can finish the proof. Conclude that

$$\mathsf{u}(x,i_1,z_1) \approx \mathsf{add}\,(x,\mathsf{u}(x',i_0,z_0))$$
$$[i_1 = i_0 + 1 \wedge z_1 = z_0 + i_0 \wedge x' = x - 1 \wedge x \geq i_0 \wedge i_0 = i + 1 \wedge z_0 = z + i \wedge x > 0]$$

is a valid lemma, which allows us to prove that $\mathsf{sum1}(x) \approx \mathsf{sum2}(x)$ is an inductive theorem.

5 Generalization with matrix invariants

InGen is a good starting point when looking for a lemma generation method, but it also has some flaws. First, not all equalities/inequalities produced by InGen are always invariants. In such cases we will not obtain a valid lemma. This for example happens when trying to prove the equivalence of Figure 1b and Figure 1c. The second, more serious, problem is that InGen is not always able to produce enough invariants. We provide a method for generating inequality invariants, by using matrix calculations. While its usefulness is limited when used on its own, we show various strategies on combining it with SMT (Section 5.2) to obtain a method that can handle a variety of systems that were previously out of reach.

5.1 Matrix invariants

Example 3 Consider the LCTRSs for 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, we obtain the following divergence:

$$\mathcal{H} = \left\{ \begin{array}{l} \mathsf{sum2}(x) \to \mathsf{v}(x,0), \\ \mathsf{v}(y_0,z_0') \to \mathsf{add}\,(x,\mathsf{v}(y_0,0)) \;\; [y_0 = x-1 \wedge z_0' = x \wedge x > 0], \\ \mathsf{v}(y_1,z_1') \to \mathsf{add}\,(x,\mathsf{v}(y_1,z_1)) \;\; [y_1 = x-2 \wedge z_1' = 2x-1 \wedge z_1 = x-1 \wedge x > 1], \\ \mathsf{v}(y_2,z_2') \to \mathsf{add}\,(x,\mathsf{v}(y_2,z_2)) \;\; [y_2 = x-3 \wedge z_2' = 3x-3 \wedge z_2 = 2x-3 \wedge x > 2], \\ \mathsf{v}(y_3,z_3') \to \mathsf{add}\,(x,\mathsf{v}(y_3,z_3)) \;\; [y_3 = x-4 \wedge z_3' = 4x-6 \wedge z_3 = 3x-6 \wedge x > 3] \end{array} \right\}$$

Using InGen, we obtain a rule $(R7')$ $\mathsf{sum3}(x) \to \mathsf{v}(x,z)$ $[z = 0]$. Dropping initialization $z = 0$ before the second expansion yields an equation $\mathsf{v}(y,z') \approx \mathsf{add}\,(x,\mathsf{v}(y,z))$ $[z' = z + x \wedge y = x - 1 \wedge x > 0]$. This is, however, not a valid lemma because $y = x - 1$ is not an invariant. If we remove this non-invariant we are very close to a valid lemma. The resulting equation is not provable by RI because we are not able to do the first `EXPANSION` step: this would add the non-terminating rule $\mathsf{v}(y,z') \to \mathsf{add}(x,\mathsf{v}(y,z))$ $[z' = z + x \wedge x > 0]$ to $\mathcal{H}$, which is not allowed. The problem can be solved by adding an invariant $z \geq 0$ to the constraint,

which is satisfied by all equations in the divergence. The resulting lemma can be easily proved with rewriting induction, which in turn allows us to prove $\mathsf{sum2}(x) \approx \mathsf{sum3}(x)$. Instead of randomly dropping constraints, and adding inequalities to protect termination, we present a systematic method that is able to detect non-invariants automatically, as well is able to generate invariants.

Simple divergence In Example 3, the rules in $\mathcal{H}$ (except for the first) obey a *simple* divergence pattern: every instance of the divergence has the same shape. Each instance contains the *divergence variables* y_i, z'_i and z_i, all dependent on x, which we will call an *initialization variable*. First, we need to distinguish between those two types of variables.

Definition 1 Let $\sqsupseteq$ be the quasi-order on Σ_{terms} generated by $f \sqsupseteq g$ 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 $\sqsupset = (\sqsupseteq \setminus \sqsubseteq)$ and the set of initializations $\mathcal{T}erms_{\mathtt{init}}$, where $\overline{\ell} = (\ell_1,\ldots,\ell_n)$:

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

Define $\mathcal{V}_{init} = \mathcal{V}ar(\mathcal{T}erms_{\mathtt{init}})$ to be the set of *initialization variables*.

Example 4 In Example 3: $\sqsupseteq$ is the transitive reflexive closure of $\{\mathsf{sum2} \sqsupseteq \mathsf{add},\ \mathsf{add} \sqsupseteq \mathsf{return},\ \mathsf{sum3} \sqsupseteq \mathsf{v},\ \mathsf{v} \sqsupseteq \mathsf{return}\}$. Hence, $\mathsf{sum2} \sqsupset \mathsf{add} \sqsupset \mathsf{return}$ and $\mathsf{sum3} \sqsupset \mathsf{v} \sqsupset \mathsf{return}$. There is one initialization x in $\mathsf{sum2}(x) \to \mathsf{add}(x,\mathsf{sum2}(x-1))\ [x>0]$, and there are initializations x and 0 in $\mathsf{sum3}(x) \to \mathsf{v}(x,0)$. So $\mathcal{T}erms_{\mathtt{init}} = \{x,0\}$ and $\mathcal{V}_{init} = \{x\}$.

Definition 2 A simple divergence pattern in n parameters is a sequence $(\rho_i)_{i\in\mathbb{N}}$ of rewrite rules together with a substitution χ, called the *diverge substitution*, variables $w_1,\ldots,w_n$, called the *divergence variables*, and terms ℓ, r, called the *divergence shape*, such that:

1. $\mathcal{V}_{div} \subseteq \mathcal{V}ar(\ell) \cup \mathcal{V}ar(r)$, where $\mathcal{V}_{div} = \{w_1,\ldots,w_n\}$ is the set of divergence variables.
2. Every ρ_i has a shape $\ell[w_1 := v_{i,1},\ldots,w_n := v_{i,n}] \to r[w_1 := v_{i,1},\ldots,w_n := v_{i,n}]\ [\varphi_i]$.
3. For every i and $1 \le j \le n$: φ_i contains an equality $v_{i,j} = e_{i,j}$ for some $e_{i,j} \in \mathcal{T}erms(\Sigma_{theory},\mathcal{V}_{init})$. We call $\vec{e}_i := (e_{i,1},\ldots,e_{i,n})$ the *divergence vector* for i.
4. For every i and $1 \le j \le n$: $e_{i+1,j} = \chi(w_j)[w_1 := e_{i,1},\ldots,w_n := e_{i,n}]$.

Define *divergence space* $\mathcal{D}iv = \{\vec{e}_i \mid i \in \mathbb{N}\} = \{\chi^i(\vec{w})[\vec{w} := \vec{e}_0] \mid i \in \mathbb{N}\}$.

Example 5 In Example 3, every rule is of the shape $\mathsf{v}(y_i,z'_i) \to \mathsf{add}(x,\mathsf{v}(y_i,z_i))$, which satisfies the divergence pattern with divergence shape $(\mathsf{v}(y,z'),\ \mathsf{add}(x,\mathsf{v}(y,z)))$ and divergence variables $\mathcal{V}_{div} = \{y,z,z'\}$. The divergence substitution is $\chi = [y := y-1,\ z' := z'+y,\ z := z+y]$ and we have $\mathcal{V}_{init} = \{x\}$. Hence $\vec{v}_0 = (y_0,z'_0,z_0)$ and $\vec{e}_0 = (x-1,x,0)$. We can now successively apply χ to compute divergence space

$$\mathcal{D}iv = \{\underbrace{(x-1,x,0)}_{\vec{e}_0},\ \underbrace{(x-2,2x-1,x-1)}_{\chi(\vec{w})[\vec{w}:=\vec{e}_0]},\ \underbrace{(x-3,3x-3,2x-3)}_{\chi^2(\vec{w})[\vec{w}:=\vec{e}_0]},\ \underbrace{(x-4,4x-6,3x-6)}_{\chi^3(\vec{w})[\vec{w}:=\vec{e}_0]},\ \ldots\}$$

Note that $\mathcal{D}iv \subseteq \mathbb{Z}[x]^3$. In general, for TRSs over the theory of integers, $\mathcal{D}iv \subseteq \mathbb{Z}[\mathcal{V}_{init}]^n$ where n is the number of parameters of the divergence pattern.

Definition 3 Let $n = length(\vec{e}_0)$, so $\mathcal{D}iv \subseteq \mathbb{Z}[\mathcal{V}_{init}]^n$. An *invariant* is a non-zero function $f : \mathbb{Z}[\mathcal{V}_{init}]^n \to \mathbb{Z}[\mathcal{V}_{init}]$ such that $f(\vec{e}_i) = 0$, for all $\vec{e}_i \in \mathcal{D}iv$.

Let us consider an affine invariant of the shape $f = a_0 + \sum_{v\in\mathcal{V}_{div}} a_v \cdot v$ with coefficients $a_0, a_v \in \mathbb{Z}[\mathcal{V}_{init}]$.

Example 6 In Example 5, an affine invariant is a polynomial $f(y,z',z) = a_0 + a_y \cdot y + a_{z'} \cdot z' + a_z \cdot z \in \mathbb{Z}[x][y,z',z]$, for some $a_0, a_y, a_{z'}, a_z \in \mathbb{Z}[x]$, such that $f(\vec{e}_0) = f(\vec{e}_1) = f(\vec{e}_2) = \ldots = 0$. Considering the first $n+1 = 4$ (the number of unknown coefficients) requirements yields a system of linear equations:

$$\underbrace{\begin{pmatrix} 1 & x-1 & x & 0 \\ 1 & x-2 & 2x-1 & x-1 \\ 1 & x-3 & 3x-3 & 2x-3 \\ 1 & x-4 & 4x-6 & 3x-6 \end{pmatrix}}_{\mathbf{L}} \underbrace{\begin{pmatrix} a_0 \\ a_y \\ a_{z'} \\ a_z \end{pmatrix}}_{\vec{a}} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Definition 4 Let $n = \mathit{length}(\vec{e}_0)$. The *affine invariant matrix* is defined by $\mathbf{L} = \begin{pmatrix} 1 & \vec{e}_0 \\ 1 & \vec{e}_1 \\ \vdots & \vdots \\ 1 & \vec{e}_n \end{pmatrix}$.

Matrix invariants Observe that an affine invariant corresponds to a vector $\vec{a} \in \mathbb{Z}[\mathcal{V}_{\mathit{init}}]$ such that $\mathbf{L}\vec{a} = \vec{0}$. In terminology of linear algebra, the collection of all such vectors is known as the *kernel* of $\mathbf{L}$, notation $\ker(\mathbf{L})$. So for any affine invariant $\vec{a}$ we necessarily have that $\vec{a} \in \ker(\mathbf{L}) \setminus \{\vec{0}\}$. Using Gaussian elimination (or in practice more optimized methods), computer algebra systems can compute a basis for the kernel of a symbolic matrix. Each basis vector gives us possibly an affine invariant: even though $\ker(\mathbf{L}) \setminus \{\vec{0}\}$ contains all affine invariants, the converse "every element in $\ker(\mathbf{L}) \setminus \{\vec{0}\}$ is a affine invariant", may not be true. The reason for this is that, in order to be able to compute the matrix $\mathbf{L}$, we restricted the full requirement "$f(\vec{e}_i) = 0$, for all i" to a finite initial part of length $n+1$.

Corollary 1 *Any affine invariant is contained in* $\ker(\mathbf{L}) \setminus \{\vec{0}\}$.

Remark. Some complications may arise due to the fact that we work with modules instead of vector spaces. E.g. in Example 6 we consider $\mathbb{Z}[x]^4$ as a $\mathbb{Z}[x]$-module. Although this is a free module (meaning that it has a basis) this unfortunately does not imply that every submodule is also free.

Example 7 In Example 6 we compute $\ker(\mathbf{L}) = \mathrm{span}\{(x,0,-1,1)\}$, corresponding to the invariant $f(y,z',z) = x \cdot 1 + 0 \cdot y + (-1) \cdot z' + 1 \cdot z$. Since $f(\vec{e}_i) = 0$, for all $\vec{e}_i \in \mathcal{D}\mathit{iv}$, this translates to $z' = x + z$ for all $\vec{e}_i = (y_i, z'_i, z_i) \in \mathcal{D}\mathit{iv}$. This gives us the equation $\mathsf{v}(y,z') \approx \mathsf{add}(x, \mathsf{v}(y,z))$ $[z' = x+z]$. As we saw in Example 3, this is not a lemma yet: we need some additional inequalities to guarantee termination.

Two questions remain. First, how to automatically compute the divergence substitution χ; and second, how to adapt the resulting lemma for termination. For the former question, we can use the following strategy: start with the divergence shape $\ell \to r$ in equational form $\ell \approx r$, using an empty constraint, and expand on the side in accordance to the divergence pattern. Here, we only need to consider the case responsible for the ongoing divergence. Then simplify the corresponding equation as far as possible, after which we can read off the divergence substitution.

Example 8 In Example 3 we start with equation $\mathsf{v}(y,z') \approx \mathsf{add}(x, \mathsf{v}(y,z))$ and expand on the left side, where we only consider the case corresponding to rule $(R8)$. This yields $\mathsf{v}(y-1, z'+y) \approx \mathsf{add}(x, \mathsf{v}(y,z))$ $[y > 0]$. Now, by simplifying as far as possible, we obtain $\mathsf{v}(y-1, z'+y) \approx \mathsf{add}(x, \mathsf{v}(y-1, z+y))$ $[y > 0]$ and we can read off the divergence substitution $\chi = [y := y-1,\ z' := z'+y,\ z := z+y]$.

As for the second question, we will next see how to use InGen, the original divergence and an SMT solver to generate candidate inequalities to be added to the lemma.

5.2 Generating additional invariants

In practice, it happens that the conjunction of all matrix invariants "almost" yields a lemma, in the sense that we only need some easy additional inequalities. For example, in the previous section we needed some inequalities (such as $z' > z \wedge z \geq 0$) to satisfy the termination requirement. In this section we will provide several ideas for the generation of such invariants.

Using inequalities from InGen

Example 9 Consider again $\mathsf{sum1}(x) \approx \mathsf{sum2}(x)$, but suppose we try to generate a lemma using matrix invariants instead of InGen. The divergence pattern is given by $\mathsf{u}(x,i',z') \to \mathsf{add}(x,\mathsf{u}(x',i,z))$ with $x' = x-1$, $\mathcal{V}_{init} = \emptyset$ and $\mathcal{V}_{div} = \{i',z',i,z\}$, where $\vec{e}_0 = (2,1,1,0)$ (the values for i'_0, z'_0, i_0, z_0 respectively) and divergence substitution $\chi = [i' := i'+1,\ z' := z'+i',\ i := i+1,\ z := z+y]$. This yields $\mathbf{L} = \begin{pmatrix} 1 & 2 & 1 & 1 & 0 \\ 1 & 3 & 3 & 2 & 1 \\ 1 & 4 & 6 & 3 & 3 \\ 1 & 5 & 10 & 4 & 6 \\ 1 & 6 & 15 & 5 & 10 \end{pmatrix}$. Compute $\ker(\mathbf{L}) = span\{(1,-1,0,1,0),(-1,1,-1,0,1)\}$, corresponding to invariants $1-i'+i=0$ and $-1+i'-z'+z=0$, respectively. This yields $\mathsf{u}(x,i',z') \approx \mathsf{add}\,(x,\mathsf{u}(x',i,z))$ $[i' = i+1 \wedge z'+1 = z+i' \wedge x' = x-1]$. This equation is too general, and cannot be proved with RI. The missing inequalities can be obtained from InGen: in Example 2 it was shown how InGen computes a lemma $\mathsf{u}(x,i'_0,z'_0) \approx \mathsf{add}\,(x,\mathsf{u}(x',i_0,z_0))$ $[i'_0 = i_0+1 \wedge z'_0 = z_0+i_0 \wedge x' = x-1 \wedge x \geq i_0 \wedge i_0 = i+1 \wedge z_0 = z+i \wedge x > 0]$. If we add $x \geq i_0$ (which becomes $x \geq i$ on $\mathcal{V}_{div}$) to our equation, we have a provable lemma.

However, how can we know that we can safely add an inequality to the constraint?

Verification by SMT In general, clauses produced by InGen are not always invariants and therefore, we should be careful when adding them to the constraint φ. We show how an SMT solver can help us to verify that, e.g., an inequality $s \geq t$ is an invariant with respect to substitution χ and constraint φ. For this, it needs to check that $\varphi \wedge s \geq t \wedge \psi_{EXP} \Longrightarrow \chi(s) \geq \chi(t)$ is (universally quantified) valid. Here, ψ_{EXP} is the additional constraint obtained in the divergence leg after performing `EXPANSION`.

Example 10 In Example 9, the SMT solver will verify that $(i' = i+1 \wedge z'+1 = z+i' \wedge x' = x-1) \wedge x \geq i \wedge \psi_{EXP} \Longrightarrow x \geq i+1$, so $x \geq i$ is added to the constraint. Here, $\psi_{EXP} := i' \leq x$. To clarify how to obtain this ψ_{EXP}, the part of the RI where ψ_{EXP} (colored blue) is introduced is shown below.

Lemma: $\mathsf{u}(x,i',z') \approx \mathsf{add}\,(x,\mathsf{u}(x',i,z))$ $[i' = i+1 \wedge z'+1 = z+i' \wedge x' = x-1 \wedge x \geq i]$

Rewriting induction: start with the lemma-equation and perform `EXPANSION` on $\mathsf{u}(x,i',z')$. We obtain

$$\mathcal{E} = \begin{cases} \mathsf{u}(x,i+1,z'+i') \approx \mathsf{add}\,(x,\mathsf{u}(x',i,z)) \ [i' \leq x \wedge i' = i+1 \wedge z'+1 = z+i' \wedge x' = x-1 \wedge x \geq i] \\ \mathsf{return}(z') \approx \mathsf{add}\,(x,\mathsf{return}(z)) \ [i' > x \wedge i' = i+1 \wedge z'+1 = z+i' \wedge x' = x-1 \wedge x \geq i] \end{cases}$$

$\mathcal{H} = \{\mathsf{u}(x,i',z') \to \mathsf{add}\,(x,\mathsf{u}(x',i,z)) \ [i' = i+1 \wedge z'+1 = z+i' \wedge x' = x-1 \wedge x \geq i]\}$

The RI-proof is now easily finished: the details are left to the reader. Notice that the lemma generalizes the divergence pattern of $\mathsf{sum1}(x) \approx \mathsf{sum2}(x)$, so can indeed be used to prove this equation.

We can use the same idea to build on Example 7 in Section 5.1.

Example 11 The equation in Example 7 could not be proven because of termination. InGen contains the clause $x > 0$. As this inequality does not contain any of the step-variables y_i, z_i or z'_i, we have $\chi(x) = x > 0 = \chi(0)$, and we immediately see that $z' = x+z \wedge x > 0 \wedge \psi_{EXP} \Longrightarrow x > 0$ is valid. Hence, we add $x > 0$ to the constraint and obtain the equation $\mathsf{v}(y,z') \approx \mathsf{add}(x,\mathsf{v}(y,z))$ $[z' = x+z \wedge x > 0]$.

Unfortunately, this is not sufficient for termination: in the corresponding induction rule (that will appear during the RI-process) the second argument to v decreases in each step, but is not bounded from below. To have termination, we should for instance have an additional clause $z \geq 0$ or $z' \geq 0$.

Using increasing/decreasing variables In Example 11 we saw that InGen cannot always provide us enough invariants to ensure termination. We show that we can use increasing/decreasing variables in order to generate more invariants. We propose an approach very similar to the one in Section 5.2: for a lemma $s \approx t\ [\varphi]$ with divergence substitution χ and divergence variables $\mathcal{V}_{div} = \{w_1, \ldots, w_n\}$, we check for every j if $\varphi \wedge \psi_{EXP} \Longrightarrow \chi(w_j) \geq w_j$. If this holds, we may add $w_j \geq e_{0,j}$ to φ. Similarly, if $\varphi \wedge \psi_{EXP} \Longrightarrow \chi(w_j) \leq w_j$, we may add $w_j \leq e_{0,j}$.

Example 12 To finish Example 11, note that $\chi(z) = z + y$ (as in Example 5) and $\psi_{EXP} = y > 0$ (as in Example 8). We have $(z' = x + z \wedge x > 0) \wedge y > 0 \Longrightarrow z + y > z$. Since $z_0 = 0$, we add $z \geq 0$ to the constraint of the lemma, yielding $\mathsf{v}(y, z') \approx \mathsf{add}(x, \mathsf{v}(y, z))\ [z' = x + z \wedge x > 0 \wedge z \geq 0]$ which can easily be proved, and can be used to finish the equivalence proof of Example 3. Alternatively (or in addition), we could add $z' \geq x$ by the same reasoning, which also yields a provable lemma.

Remark. Rather than essentially guessing good inequalities, we can consider what is necessary for specific termination techniques. In this idea, we simulate a proof for the lemma while ignoring the termination requirements. Then, we consider what additional requirements are needed for our termination method to conclude termination. This may provide a number of candidate constraints. We verify for each of the possibilities if it is satisfied in the original divergence pattern (possibly skipping the first few steps), and if so, test if the proof still goes through when these constraints are added to the lemma. For example, if we want to use a simple version of the *value criterion* [17] to prove termination of a rule $\mathsf{v}(y, z') \to \mathsf{add}(x, \mathsf{v}(y, z))\ [z' = x + z \wedge x > 0]$ we need $z' \geq 0$. This property is valid in all of the diverging equations, so we can simply add it to our lemma.

The downside of this approach is that it fundamentally breaks modularity, and ties the lemma generation technique to specific termination methods. This is why we prefer the more general technique of identifying an increase (or decrease) through χ and ψ_{EXP}.

Array programs Matrix invariants are not only applicable to integer programs. In Appendix A we show it can be used to prove the equivalence between two array programs. As usual, an additional inequality invariant is needed and we show how to gain this from a non-invariant produced by InGen.

6 Discussion and future work

We proposed a method for computing invariants on simple divergence patterns using matrix calculations. A script in GNU Octave [1] was implemented in order to calculate invariants of a manually inserted divergence pattern. A short demonstration is included in Appendix B. Considering existing work: using linear algebra for generating invariants is not entirely new (e.g. [8, 9]). We might in general benefit from the work in relational verification, which will be part of our future work.

Often, to obtain a valid lemma, additional inequality invariants are needed which cannot be generated by our matrix method. For this problem we proposed several ideas, all of them using an SMT solver. Here, we also might benefit from existing work, like widening operators [5, 6].

[1] Accessed from `https://github.com/kasperhagens/polynomial-invariants` on May 2023.

Fully automatic lemma generation (for large scale testing of our ideas) requires more: the only way to find out that some particular equation is an inductive theorem is by performing its RI-proof. This means that automatic lemma generation unavoidably needs an implementation of RI. This RI-tool should be able to automatically identify a divergence pattern and then calculate the required invariants.

References

[1] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. USA: Cambridge University Press, 2004.

[2] S. K. Lahiri, A. Murawski, O. Strichman, and M. Ulbrich, "Program equivalence (dagstuhl seminar 18151)," in *Dagstuhl Reports*, vol. 8, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

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

[4] N. Nishida, M. Kojima, and T. Kato, "On transforming imperative programs into logically constrained term rewrite systems via injective functions from configurations to terms," tech. rep., EasyChair, 2022.

[5] S. Sankaranarayanan, H. Sipma, and Z. Manna, "Constraint-based linear-relations analysis," pp. 53–68, 08 2004.

[6] R. Bagnara, P. Hill, E. Ricci, and E. Zaffanella, "Precise widening operators for convex polyhedra," *Sci. Comput. Program.*, vol. 58, pp. 28–56, 01 2005.

[7] S. Almagor, D. Chistikov, J. Ouaknine, and J. Worrell, "O-minimal invariants for discrete-time dynamical systems," *ACM Trans. Comput. Logic*, vol. 23, jan 2022.

[8] M. Muller-olm and H. Seidl, "Precise interprocedural analysis through linear algebra," vol. 39, pp. 330–341, 01 2004.

[9] S. de Oliveira, S. Bensalem, and V. Prevosto, "Polynomial invariants by linear algebra," in *Automated Technology for Verification and Analysis: 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings 14*, pp. 479–494, Springer, 2016.

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

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

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

[13] 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.

[14] 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.

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

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

[17] C. Kop, "Termination of LCTRSs," in *Proc. WST*, 2013.

A Array programs

A signature and interpretation for arrays was introduced in [3]: for each sort ι introduce a sort $\mathsf{array}(\iota)$ and theory symbols $\mathsf{size}_\iota : [\mathsf{array}(\iota)] \Rightarrow \mathsf{int}$, $\mathsf{select}_\iota : [\mathsf{array}_\iota \times \mathsf{int}] \Rightarrow \iota$, $\mathsf{store}_\iota : [\mathsf{array} \times \mathsf{int} \times \iota] \Rightarrow \mathsf{array}(\iota)$

where everything is interpreted as expected. For example $\mathcal{I}_{\mathsf{array}(\iota)} = \mathcal{I}_\iota^*$ and, for $a = \langle a_0, \ldots, a_{n-1}\rangle$ define $\mathcal{J}(\mathsf{store}_\iota)(a,k,v) = \langle a_0, \ldots, a_{k-1}, v, a_{k+1}, \ldots, a_{n-1}\rangle$ if $0 \leq k < n$ and $\mathcal{J}(\mathsf{store}_\iota)(a,k,v) = a$ otherwise.

When working with arrays it is convenient to have bounded quantifications because they allow us to express quantified statements with a parametrized range of quantification, such as $(\forall 0 \leq i < n)(\mathsf{select}(a,i) \neq 0)$, where we have a range parameter n. For our purposes, bounded quantifications are important because they are often necessary to express a generalization of a divergence. A method to automatically generate such generalizations from the divergence was introduced by [3]. We will not explain how this works, but we will use the bounded quantification it generates in the following example:

The LCTRS below represents two programs whose input is an array a of size l, returning the reverse of a. Both programs first allocate an array r of size l which is then filled with the elements from a in reversed direction. Here, revU fills in upwards direction whereas revD fills in downwards direction.

For readability we write $\mathsf{s}(a)$ instead of $\mathsf{size}(a)$, and $a[i]$ instead of $\mathsf{select}(a,i)$.

$$
\begin{array}{ll}
\mathsf{reverseU}(a,l) \to \mathsf{revU}(a,r,l,0) & [\mathsf{s}(r) = l] \\
\mathsf{revU}(a,r,l,i) \to \mathsf{revU}(a,\mathsf{store}(r,i,a[l-i-1]),l,i+1) & [i < l] \\
\mathsf{revU}(a,r,l,i) \to \mathsf{return}(r) & [i \geq l] \\
\mathsf{reverseD}(a,l) \to \mathsf{revD}(a,r,l,l) & [\mathsf{s}(r) = l] \\
\mathsf{revD}(a,r,l,i) \to \mathsf{revD}(a,\mathsf{store}(r,i-1,a[l-i]),l,i-1) & [i > 0] \\
\mathsf{revD}(a,r,l,i) \to \mathsf{return}(r) & [i \leq 0]
\end{array}
$$

We wish to prove that $\mathsf{revU}(a,l) \approx \mathsf{revD}(a,l)$ $[l > 0]$ but get a simple divergence shape $(\mathsf{revU}(a,r,l,i), \mathsf{revD}(a,q,l,I))$ with $\mathcal{V}_{div} = \{r,i,q,I\}$, $\mathcal{V}_{init} = \{l\}$ and divergence substitution $\chi = [r := \mathsf{store}(r,i,a[l-i-1]),\ i := i+1,\ q := \mathsf{store}(q,I-1,a[l-I]),\ I := I-1]$. Since q and r are arrays (and not integers) we cannot directly include them into the divergence matrix. We remove r and q from the divergence pattern and the remaining divergence variables are only i and I. Then the affine divergence matrix is given by $\mathbf{L} = \begin{pmatrix} 1 & 1 & l-1 \\ 1 & 2 & l-2 \\ 1 & 3 & l-3 \end{pmatrix}$ with $\ker(\mathbf{L}) = span\{(l,-1,-1)\}$, corresponding to $l-i-I=0$. This yields an equation $\mathsf{revU}(a,r,l,i) \approx \mathsf{revD}(a,q,l,I)$ $[l = i+I \wedge \mathsf{s}(r) = \mathsf{s}(q) = l > 0]$, which is not a lemma yet. We apply the method from [3] to generate the bounded quantifications. Together with InGen this gives us the following equation $\mathsf{revU}(a,r_0,l,i_0) \approx \mathsf{revD}(a,q_0,l,I_0)$ $[(\forall 0 \leq j < i)(a[l-j-1] = r[j]) \wedge (\forall I \leq j < l)(a[l-j-1] = q[j]) \wedge I_0 = l-1 \wedge l > 0 \wedge \ldots]$. Here, $I_0 = l-1$ is not an invariant, because $\varphi \wedge \psi_{EXP} \wedge I_0 = l-1 \Longrightarrow \chi(I_0) = \chi(l-1)$ is not universally valid. Still, the equality contains useful information: $I_0 = l-1$ together with $l > 0$ implies $I_0 \geq 0$, where $I_0 \geq 0$ *is* an invariant. Indeed, the SMT solver will confirm that $\varphi \wedge \psi_{EXP} \wedge I_0 \geq 0 \Longrightarrow \chi(I_0) \geq \chi(0)$. We obtained the following lemma, by which we can prove $\mathsf{revU}(a,l) \approx \mathsf{revD}(a,l)$ $[l > 0]$.

Lemma: $\mathsf{revU}(a,r,l,i) \approx \mathsf{revD}(a,q,l,I)$ $[\mathsf{s}(r) = \mathsf{s}(q) = I+i = l > 0 \wedge I \geq 0 \wedge (\forall 0 \leq j < i)(a[l-j-1] = r[j]) \wedge (\forall I \leq j < l)(a[l-j-1] = q[j])]$

B Demonstration

The affine invariants of $\mathsf{sum2}(x) \approx \mathsf{sum3}(x)$ can be calculated by executing the following commands

```
pkg install -forge symbolic-3.0.1.tar.gz
pkg load symbolic
syms x;
e0 = sym([x-1, x, 0]);
n=3;
d=1;
Invariants = invariants(n, d, @substitutionSumtwothree, e0);
```

Here, we call the function `substitutionSumtwothree.m` encoded by

```
function CHI = substitutionSumtwothree (Y)
  CHI(1) = Y(1)-1;
  CHI(2) = Y(2)+Y(1);
  CHI(3) = Y(3)+Y(1);
endfunction
```

representing the substitution $\chi = [y := y-1,\ z' := z'+y,\ z := z+y]$.

This produces the following output:

The monomial vector is given by

$$\begin{bmatrix} 1 \\ Y_{11} \\ Y_{12} \\ Y_{13} \end{bmatrix}$$

The divergence matrix is given by

$$\begin{bmatrix} 1 & x-1 & x & 0 \\ 1 & x-2 & 2\cdot x-1 & x-1 \\ 1 & x-3 & 3\cdot x-3 & 2\cdot x-3 \\ 1 & x-4 & 4\cdot x-6 & 3\cdot x-6 \end{bmatrix}$$

A basis for its kernel is given by

$$\begin{bmatrix} x \\ 0 \\ -1 \\ 1 \end{bmatrix}$$

Corresponding to the following invariants

Invariants = (Sym) $-Y_{12} + Y_{13} + x = 0$