Verifying Procedural Programs via Constrained Rewriting Induction

CARSTEN FUHS, Birkbeck, University of London
CYNTHIA KOP, University of Innsbruck and University of Copenhagen
NAOKI NISHIDA, Nagoya University

This article aims to develop a verification method for procedural programs via a transformation into logically constrained term rewriting systems (LCTRSs). To this end, we extend transformation methods based on integer term rewriting systems to handle arbitrary data types, global variables, function calls, and arrays, and to encode safety checks. Then we adapt existing rewriting induction methods to LCTRSs and propose a simple yet effective method to generalize equations. We show that we can automatically verify memory safety and prove correctness of realistic functions. Our approach proves equivalence between two implementations; thus, in contrast to other works, we do not require an explicit specification in a separate specification language.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving

General Terms: Formal Verification

Additional Key Words and Phrases: Constrained term rewriting, inductive theorem proving, rewriting induction, lemma generation, program analysis

ACM Reference Format:
Carsten Fuhs, Cynthia Kop, and Naoki Nishida. 2017. Verifying procedural programs via constrained rewriting induction. ACM Trans. Comput. Logic 18, 2, Article 14 (June 2017), 50 pages.
DOI: http://dx.doi.org/10.1145/3060143

1. INTRODUCTION

Ensuring with certainty that a program always behaves correctly is a hard problem. One approach to this is formal verification—proving with mathematical rigor that all executions of the program will have the expected outcome. Several methods for this have been investigated (e.g., see Huth and Ryan [2000]). However, classically many of them require expert knowledge to manually prove relevant properties about the code.

Instead, it is our hope to raise the degree of automation, ideally creating a fully automatic verification/refutation process and tools to raise developer productivity. Indeed, over the past years, automatic provers for program verification have flourished, as witnessed, for example, by tool competitions like SV-COMP [SV-COMP 2017] and the Termination Competition (http://termination-portal.org/wiki/Termination_Competition). Program verification is also recognized in industry, such as Facebook's safety prover

This work was supported by Austrian Science Fund (FWF) international project I963; Marie Skłodowska-Curie action "HORIP" (H2020-MSCA-IF-2014, 658162); the Japan Society for the Promotion of Science (JSPS); and Nagoya University's Graduate Program for Real-World Data Circulation Leaders from *MEXT*, Japan.
Authors' addresses: C. Fuhs, Birkbeck, University of London, Department of Computer Science and Information Systems, Malet Street, London WC1E 7HX, United Kingdom; email: carsten@dcs.bbk.ac.uk; C. Kop, University of Copenhagen, Department of Computer Science (Datalogisk Institut), Emil Holms Kanal 6, 2300 København S, Denmark; email: kop@di.ku.dk; N. Nishida, Nagoya University, Graduate School of Informatics, Furo-cho, Chikusa-ku, Nagoya 4648603, Japan; email: nishida@i.nagoya-u.ac.jp.

ACM 1529-3785/2017/06-ART14 $15.00
DOI: http://dx.doi.org/10.1145/3060143

Infer [Calcagno et al. 2015] or Microsoft's temporal prover T2 [Brockschmidt et al. 2016]. However, these tools generally use specific reasoning techniques for imperative programs and benefit from the progress in automated theorem proving over the past decades only to a limited extent. This suggests likely avenues for improvement.

One such avenue is *inductive theorem proving*. This method is well investigated in functional programming [Bundy 2001] and term rewriting, the underlying core calculus of functional programming. To check a functional program f against a specification by a reference implementation f_{spec}, it suffices that $f(\overrightarrow{x}) \approx f_{spec}(\overrightarrow{x})$ is an inductive theorem. Thus, no explicit specification language is needed: giving a (possibly not optimized) reference implementation f_{spec} in the same programming language suffices.

To analyze imperative programs (in C, Java, etc.), recent works have applied transformations into term rewriting systems (TRSs) (e.g., Otto et al. [2010]). In particular, *constrained rewriting systems* are popular as target language, since logical constraints to model the control flow can be separated from terms to model intermediate states [Furuichi et al. 2008; Falke and Kapur 2009; Sakata et al. 2009; Nakabayashi et al. 2010; Falke et al. 2011]. Unifying existing approaches, Kop and Nishida [2013] proposed the framework of logically constrained term rewriting systems (LCTRSs).

Aims. The aim of this article is twofold. First, we propose a new transformation method from procedural programs into constrained term rewriting. This transformation makes it possible to use the many methods available to term rewriting to also analyze imperative programs. Unlike previous methods, we do not limit interest to integer functions.

Second, we develop a verification method for LCTRSs, based on rewriting induction [Reddy 1990]—a well-investigated method of inductive theorem proving—to prove (total) equivalence of two functions. We also supply two generalization techniques, the main one of which is specialized for transformed iterative functions.

The applications are many. First, checking equivalence between different implementations comes to mind. This allows the user to determine automatically if a modification in the program has changed its semantics (e.g., see Godlin and Strichman [2013] and Lahiri et al. [2012]). Proposing equivalent replacements may even be done automatically, via algorithm recognition (e.g., see Alias and Barthou [2003]).

In compilation, automated equivalence checking can validate correctness of compiler optimizations on a per-instance basis [Necula 2000; Pnueli et al. 1998] or once-and-for-all for a given optimization template [Kundu et al. 2009; Lopes and Monteiro 2016]. Equivalence checking is also used in proofs of secure information flow [Terauchi and Aiken 2005] and can be used to prove safety properties (e.g., memory safety).

Why LCTRSs. Direct support of basic types, like the integers, and of constraints to restrict evaluation—features absent in basic TRSs—is essential to handle realistic programs. Unlike earlier constrained rewriting systems, LCTRSs do not limit the underlying theory to (linear) integer arithmetic: we might use (combinations of) arbitrary first-order theories, such as n-dimensional integer arrays, floating point numbers, and bitvectors. This makes it possible to natively handle sophisticated programs.

Despite the generality, we get strong results on LCTRSs by reducing analysis problems like termination and equivalence to a sequence of satisfiability problems over the underlying theories. Automatic tools—like our tool Ctrl [Kop and Nishida 2015] for rewriting, termination, and inductive theorem proving—can defer such queries to an external SAT modulo theories (SMT) solver [Nieuwenhuis et al. 2006], as a black box. Future advances in the SMT world then directly transfer to analysis of LCTRSs.

Structure. We first recall the LCTRS formalism from Kop and Nishida [2013] (Section 2) and show a way to translate procedural programs to LCTRSs (Section 3). Then

we lift rewriting induction methods for constrained rewriting to LCTRSs (Section 4) and strengthen them with two dedicated generalization techniques (Section 5). Finally, we discuss automation and experimental results (Section 6) as well as related and future work (Sections 7 and 8). We conclude in Section 9.

Contributions over the conference version. The present article provides several additional contributions over the conference version [Kop and Nishida 2014]. First, we significantly extend our method to translate procedural programs to LCTRSs. Second, we extend our theory of constrained inductive theorem proving to *disproving* equivalence (following Sakata et al. [2009] and Falke and Kapur [2012]) and add several inference rules. Third, we provide an additional generalization technique and a detailed proof strategy to automate rewriting induction for translated procedural programs. Fourth, we have improved the implementation and added an automatic translation from C programs to LCTRSs.

1.1. Motivating Example

Aside from business applications, automatic equivalence proving can be used as an aid in grading student programming assignments. Combining a test run of the assignments on a set of sample inputs (which identifies many incorrect programs but leaves false positives) with an automatic correctness check can save teachers a lot of time.

Example 1.1. Consider the following programming assignment.

Write a function `sum` *that, given an integer array and its length as input, returns the sum of its elements. Do not modify the input array.*

We consider four different C implementations of this exercise:

```
int sum1(int arr[],int n) {
  int ret=0;
  for(int i=0;i<n;i++)
    ret+=arr[i];
  return ret;
}
```

```
int sum2(int arr[], int n) {
  int ret, i;
  for (i = 0; i < n; i++) {
    ret += arr[i];
  }
  return ret;
}
```

```
int sum3(int arr[], int len) {
  int i;
  for (i = 0; i < len-1; i++)
    arr[i+1] += arr[i];
  return arr[len-1];
}
```

```
int sum4(int *arr, int k) {
  if (k <= 0) return 0;
  return arr[k-1] +
         sum4(arr, k-1);
}
```

The first solution (`sum1`) is correct. The second (`sum2`) is not, because `ret` is not initialized—which may be missed in standard tests depending on the compiler used. The third solution (`sum3`) is incorrect because the array is modified against the instructions and moreover gives a random result or segmentation fault if $\texttt{len} = 0$. The fourth solution (`sum4`) is correct.

These implementations can be transformed into the following LCTRSs:

$$\begin{array}{lll}
\text{(1a)} & \mathsf{sum1}(arr, n) \to \mathsf{u}(arr, n, 0, 0) & \\
\text{(1b)} & \mathsf{u}(arr, n, ret, i) \to \mathsf{error} & [i < n \wedge (i < 0 \vee i \geq \mathsf{size}(arr))] \\
\text{(1c)} & \mathsf{u}(arr, n, ret, i) \to \mathsf{u}(arr, n, ret + \mathsf{select}(arr, i), i + 1) & [i < n \wedge 0 \leq i < \mathsf{size}(arr)] \\
\text{(1d)} & \mathsf{u}(arr, n, ret, i) \to \mathsf{return}(arr, ret) & [i \geq n]
\end{array}$$

$$\begin{array}{lrcll}
\text{(2a)} & \mathsf{sum2}(arr, n) & \to & \mathsf{u}(arr, n, ret, 0) & \\
 & \multicolumn{4}{l}{\quad \mathsf{u} \text{ rules as copied from above}} \\
 & & & & \\
\text{(3a)} & \mathsf{sum3}(arr, len) & \to & \mathsf{v}(arr, len, 0) & \\
\text{(3b)} & \mathsf{v}(arr, len, i) & \to & \mathsf{error} & [i < len - 1 \wedge (i < 0 \vee i + 1 \geq \mathsf{size}(arr))] \\
\text{(3c)} & \mathsf{v}(arr, len, i) & \to & \mathsf{v}(\mathsf{store}(arr, i+1, \mathsf{select}(arr, i+1) + \mathsf{select}(arr, i)), len, i+1) & \\
 & & & & [i < len - 1 \wedge 0 \leq i \wedge i + 1 < \mathsf{size}(arr)] \\
\text{(3d)} & \mathsf{v}(arr, len, i) & \to & \mathsf{return}(arr, \mathsf{select}(arr, len - 1)) & \\
 & & & & [i \geq len - 1 \wedge 0 \leq len - 1 < \mathsf{size}(arr)] \\
\text{(3e)} & \mathsf{v}(arr, len, i) & \to & \mathsf{error} & [i \geq len - 1 \wedge (len - 1 < 0 \vee len - 1 \geq \mathsf{size}(arr))] \\
 & & & & \\
\text{(4a)} & \mathsf{sum4}(arr, k) & \to & \mathsf{return}(arr, 0) & [k \leq 0] \\
\text{(4b)} & \mathsf{sum4}(arr, k) & \to & \mathsf{error} & [k - 1 \geq \mathsf{size}(arr)] \\
\text{(4c)} & \mathsf{sum4}(arr, k) & \to & \mathsf{w}(\mathsf{select}(arr, k-1), \mathsf{sum4}(arr, k-1)) & [0 \leq k - 1 < \mathsf{size}(arr)] \\
\text{(4d)} & \mathsf{w}(n, \mathsf{error}) & \to & \mathsf{error} & \\
\text{(4e)} & \mathsf{w}(n, \mathsf{return}(a, r)) & \to & \mathsf{return}(a, n + r) &
\end{array}$$

Note that arrays carry an implicit size (their allocated memory) that is queried to model the runtime behavior of the C program and test for out-of-bound errors. The fresh variable in the right-hand side of (2a) models that the third parameter of u is assigned an *arbitrary* integer. The details of this transformation are discussed in Section 3.

Using inductive theorem proving, we can now prove that

—$\forall arr \in \mathsf{array(int)}.\ \forall len \in \mathsf{int}.\ \ \mathsf{sum1}(arr, len) \leftrightarrow^* \mathsf{sum4}(arr, len)$ if $0 \leq len \leq \mathsf{size}(arr)$
—$\exists arr \in \mathsf{array(int)}.\ \exists len \in \mathsf{int}.\ \ \mathsf{sum3}(arr, len) \not\leftrightarrow^* \mathsf{sum4}(arr, len)$ with $0 \leq len \leq \mathsf{size}(arr)$.

Thus, sum1 and sum4 return the same result on any input such that the given length does not cause out-of-bound errors, but sum3 and sum4 do not. (It seems likely that the disproof obtained from inductive theorem proving could be used to extract counterexample inputs, but at present we have not studied a systematic way of doing so.)

For sum2, we *do* have $\mathsf{sum2}(arr, len) \leftrightarrow^* \mathsf{sum4}(arr, len)$, as we can always choose to instantiate *ret* with 0. The system is not *confluent*; we can also prove that there exist a, n such that $\mathsf{sum2}(a, n) \to^* s \neq t \leftarrow^* \mathsf{sum4}(a, n)$ for terms s, t in normal form. As explained in Section 6, we use a proof strategy that typically proves only the "$\neq$" statement.

1.2. Practical Use

The primary application that we see for our technique is the following.

1.2.1. Comparing a Function to a Specification. As in Example 1.1, we can verify correctness of a C function f against a reference implementation g by translating both functions to LCTRS rules (Section 3) and proving that $\mathsf{f}(x_1, \ldots, x_n) \approx \mathsf{g}(x_1, \ldots, x_n)$ [true] is an inductive theorem. If we only need equivalence under given preconditions on the input variables—such as $0 \leq len \leq \mathsf{size}(arr)$ in Example 1.1—we formulate this as a constraint φ and analyze whether $\mathsf{f}(x_1, \ldots, x_n) \approx \mathsf{g}(x_1, \ldots, x_n)$ $[\varphi]$ is an inductive theorem.

Note that we do not require a separate specification language—although if desirable, it is of course possible to specify the reference implementation directly as an LCTRS.

Further possible applications of our technique include the following.

1.2.2. Code Optimization (or Other Improvement). Sometimes the "reference implementation" g suggested previously can simply be an existing—and inefficient, or inelegant—version of a function. Thus, inductive theorem proving can be used to

prove that it is safe to replace a function in a large real-life program by an optimized alternative.

1.2.3. Error Checking. As the transformation from C to LCTRSs includes error checking (as seen for memory safety violations in Example 1.1), we can use inductive theorem proving to verify the absence of such errors. This is done by adding error-checking rules, such as

$$\mathsf{errorfree}(\mathsf{return}(a, n)) \to \mathsf{true} \qquad \mathsf{errorfree}(\mathsf{error}) \to \mathsf{false},$$

and proving that $\mathsf{errorfree}(\mathsf{sum4}(a, n)) \approx \mathsf{true}\ [\varphi]$ is an inductive theorem, where φ is the precondition on the input. Aside from memory safety, this approach can be used to certify the absence of, for instance, divisions by zero or integer overflow. The key is in the transformation, where we can choose which constructions result in an error.

1.2.4. Classical Correctness Checks. Aside from comparisons to an example implementation, we can also specify a correctness property directly in SMT. For instance, given an implementation of the strlen function, its correctness could be verified by proving that

$$\mathsf{strlen}(x) \approx \mathsf{return}(n)\ [0 \le n < \mathsf{size}(x) \wedge \mathsf{select}(x, n) = 0 \wedge \forall i \in \{0, \ldots, n-1\}(\mathsf{select}(x, i) \neq 0)]$$

is an inductive theorem. Alternatively, we can use extra rules to test properties in SMT.

Example 1.2. To analyze correctness of an implementation of strcpy, we may use

$$\begin{aligned} \mathsf{test}(x, n, \mathsf{error}) &\to \mathsf{false} \\ \mathsf{test}(x, n, \mathsf{return}(y)) &\to b\ [b \Leftrightarrow \forall i \in \{0, \ldots, n\}(\mathsf{select}(x, i) = \mathsf{select}(y, i))] \end{aligned}$$

and prove that the following equation is an inductive theorem:

$$\begin{aligned} &\mathsf{test}(x, n, \mathsf{strcpy}(y, x)) \approx \mathsf{true} \\ &[0 \le n < \mathsf{size}(x) \wedge n < \mathsf{size}(y) \wedge \mathsf{select}(x, n) = 0 \wedge \forall i \in \{0, \ldots, n-1\}(\mathsf{select}(x, i) \neq 0)]. \end{aligned}$$

Note that this more sophisticated test is needed in this case, as correctness of strcpy does not require that $x = y$ if $\mathsf{strcpy}(x) \to^* \mathsf{return}(y)$ (the sizes of x and y may differ).

2. PRELIMINARIES

In this section, we briefly recall LCTRSs, following the definitions in Kop and Nishida [2013].

2.1. Logically Constrained Term Rewriting Systems

Many-sorted terms. We introduce terms, typing, substitutions, contexts, and subterms (with corresponding terminology) in the usual way for many-sorted term rewriting.

Definition 2.1. We assume given a set $\mathcal{S}$ of *sorts* and an infinite set $\mathcal{V}$ of *variables*, each variable equipped with a sort. A *signature* Σ is a set of *function symbols* f, disjoint from $\mathcal{V}$, each equipped with a *sort declaration* $[\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa$, with all ι_i and κ sorts. For readability, we often write κ instead of $[] \Rightarrow \kappa$. The set $\mathcal{T}erms(\Sigma, \mathcal{V})$ of *terms* over Σ and $\mathcal{V}$ contains any expression s such that $\vdash s : \iota$ can be derived for some sort ι, using

$$\frac{}{\vdash x : \iota}\ (x : \iota \in \mathcal{V}) \qquad \frac{\vdash s_1 : \iota_1 \quad \ldots \quad \vdash s_n : \iota_n}{\vdash f(s_1, \ldots, s_n) : \kappa}\ (f : [\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa \in \Sigma).$$

We fix Σ and $\mathcal{V}$. Note that for every term s, there is a unique sort ι with $\vdash s : \iota$.

Definition 2.2. Let $\vdash s : \iota$. We call ι the *sort of* s. Let $Var(s)$ be the set of variables occurring in s; we say that s is *ground* if $Var(s) = \emptyset$.

Definition 2.3. A *substitution* γ is a sort-preserving total mapping from $\mathcal{V}$ to $\mathcal{T}erms(\Sigma, \mathcal{V})$. The result $s\gamma$ of applying a substitution γ to a term s is s with all occurrences of a variable x replaced by $\gamma(x)$. The *domain* of γ, $Dom(\gamma)$, is the set of variables x with $\gamma(x) \neq x$. The notation $[x_1 := s_1, \ldots, x_n := s_n]$ denotes a substitution γ with $\gamma(x_i) = s_i$ for $1 \leq i \leq n$, and $\gamma(y) = y$ for $y \notin \{x_1, \ldots, x_n\}$. For two substitutions γ and δ, their composition $\gamma \circ \delta$ is given by $(\gamma \circ \delta)(x) = \gamma(\delta(x)) = (x\delta)\gamma$ for all variables x.

Two terms s and t are *unifiable* if there exists a substitution γ such that $s\gamma = t\gamma$. Then γ is called a *unifier* for s and t. If moreover for all unifiers γ' for s and t there is a substitution δ such that $\gamma' = \delta \circ \gamma$, we call γ a *most general unifier (mgu)* for s and t.

Definition 2.4. Given a term s, a *position* in s is a sequence p of positive integers such that $s_{|p}$ is defined, where $s_{|\epsilon} = s$ and $f(s_1, \ldots, s_n)_{|i \cdot p} = (s_i)_{|p}$. We call $s_{|p}$ a *subterm* of s. If $\vdash s_{|p} : \iota$ and $\vdash t : \iota$, then $s[t]_p$ denotes s with the subterm at position p replaced by t. A *context* C is a term containing one or more typed *holes* $\Box_i : \iota_i$. If $s_1 : \iota_i, \ldots, s_n : \iota_n$, we define $C[s_1, \ldots, s_n]$ as C with each $\Box_i$ replaced by s_i.

Logical terms. Specific to LCTRSs, we consider different kinds of symbols and terms.

Definition 2.5. We assume given:

—signatures Σ_{terms} and Σ_{theory} such that $\Sigma = \Sigma_{terms} \cup \Sigma_{theory}$;
—a mapping $\mathcal{I}$ that assigns to each sort ι occurring in Σ_{theory} a set $\mathcal{I}_\iota$;
—a mapping $\mathcal{J}$ that assigns to each $f : [\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa \in \Sigma_{theory}$ a function in $\mathcal{I}_{\iota_1} \times \cdots \times \mathcal{I}_{\iota_n} \Longrightarrow \mathcal{I}_\kappa$;
—for all sorts ι occurring in Σ_{theory} a set $\mathcal{V}al_\iota \subseteq \Sigma_{theory}$ of *values*: function symbols $a : [] \Rightarrow \iota$ such that $\mathcal{J}$ gives a bijective mapping from $\mathcal{V}al_\iota$ to $\mathcal{I}_\iota$.

We require that $\Sigma_{terms} \cap \Sigma_{theory} \subseteq \mathcal{V}al = \bigcup_\iota \mathcal{V}al_\iota$. The sorts occurring in Σ_{theory} are called *theory sorts*, and the symbols *theory symbols*. Symbols in $\Sigma_{theory} \setminus \mathcal{V}al$ are *calculation symbols*. A term in $\mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$ is called a *logical term*.

Definition 2.6. For ground logical terms, let $[\![f(s_1, \ldots, s_n)]\!] := \mathcal{J}_f([\![s_1]\!], \ldots, [\![s_n]\!])$. For every ground logical term s, there is a unique value c such that $[\![s]\!] = [\![c]\!]$; we say that c is the value of s. A *constraint* is a logical term φ of some sort bool with $\mathcal{I}_{\mathsf{bool}} = \mathbb{B} = \{\top, \bot\}$, the set of *Booleans*. A constraint φ is *valid* if $[\![\varphi\gamma]\!] = \top$ for *all* substitutions γ that map $Var(\varphi)$ to values, and *satisfiable* if $[\![\varphi\gamma]\!] = \top$ for *some* such substitutions. A substitution γ *respects* φ if $\gamma(x)$ is a value for all $x \in Var(\varphi)$ and $[\![\varphi\gamma]\!] = \top$.

Terms in $\mathcal{T}erms(\Sigma_{terms}, \emptyset)$ can be thought of as the primary objects of rewriting: a reduction typically begins and ends with such terms, with elements of $\Sigma_{theory} \setminus \mathcal{V}al$ (also called *calculation symbols*) to perform calculations in the underlying theory.

We typically choose a theory signature with $\Sigma_{theory} \supseteq \Sigma^{core}_{theory}$, where Σ^{core}_{theory} contains $\mathsf{true}, \mathsf{false} : \mathsf{bool}$, $\wedge, \vee, \Rightarrow$: $[\mathsf{bool} \times \mathsf{bool}] \Rightarrow \mathsf{bool}$, $\neg$: $[\mathsf{bool}] \Rightarrow \mathsf{bool}$, and, for all theory sorts ι, symbols $=_\iota, \neq_\iota$: $[\iota \times \iota] \Rightarrow \mathsf{bool}$, and an evaluation function $\mathcal{J}$ that interprets these symbols as expected. We omit the sort subscripts from $=$ and $\neq$ when clear from context.

Definition 2.7. The standard integer signature Σ^{int}_{theory} is $\Sigma^{core}_{theory} \cup \{+, -, *, \mathsf{exp}, \mathsf{div}, \mathsf{mod} : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int}; \leq, < : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{bool}\} \cup \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\}$ with values true, false, and n for all $n \in \mathbb{Z}$. Thus, we use n (in sans-serif font) as the function symbol for $n \in \mathbb{Z}$ (in *math* font). We define $\mathcal{J}$ in the natural way, except since all $\mathcal{J}_f$ must be total functions, we set $\mathcal{J}_{\mathsf{div}}(n, 0) = \mathcal{J}_{\mathsf{mod}}(n, 0) = \mathcal{J}_{\mathsf{exp}}(n, k) = 0$ for all n and all $k < 0$. Of course, when constructing LCTRSs, we normally add explicit error checks to prevent such calls.

Example 2.8. Let $\mathcal{S} = \{\mathsf{int}, \mathsf{bool}\}$, and $\Sigma = \Sigma_{terms} \cup \Sigma^{int}_{theory}$, where

$$\Sigma_{terms} = \{\mathsf{fact} : [\mathsf{int}] \Rightarrow \mathsf{int}\} \cup \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\}.$$

Then both int and bool are theory sorts. We also define set and function interpretations—for instance, $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$, $\mathcal{I}_{\mathsf{bool}} = \mathbb{B}$, and $\mathcal{J}$ is defined as earlier. With $=$ for $=_{\mathsf{int}}$ and infix notation, examples of logical terms are $0 = 0 + -1$ and $x + 3 \geq y + -42$. Both are constraints. Additionally, $5 + 9$ is also a (ground) logical term but not a constraint. Expected starting terms are, for example, $\mathsf{fact}(42)$ or $\mathsf{fact}(\mathsf{fact}(-4))$: ground terms fully built using symbols in Σ_{terms}.

Rules and rewriting. We adapt the standard notions of rewriting (e.g., see Baader and Nipkow [1998]) by including constraints and adding rules to perform calculations.

Definition 2.9. A *rule* is a triple $\ell \to r\ [\varphi]$ with ℓ and r terms of the same sort and φ a constraint. Here, ℓ has the form $f(\ell_1, \ldots, \ell_n)$ and contains at least one symbol in $\Sigma_{terms} \setminus \Sigma_{theory}$ (so ℓ is not a logical term). If $\varphi = \mathsf{true}$ with $\mathcal{J}(\mathsf{true}) = \top$, we may write $\ell \to r$. We define $\mathit{LVar}(\ell \to r\ [\varphi])$ as $\mathit{Var}(\varphi) \cup (\mathit{Var}(r) \setminus \mathit{Var}(\ell))$. A substitution γ *respects* $\ell \to r\ [\varphi]$ if $\gamma(x) \in \mathcal{V}al$ for all $x \in \mathit{LVar}(\ell \to r\ [\varphi])$, and $[\![\varphi\gamma]\!] = \top$. The rule is *left linear* if ℓ is linear (i.e., all variables occur at most once in ℓ) and *irregular* if $\mathit{Var}(\varphi) \setminus \mathit{Var}(\ell) \neq \emptyset$.

Note that it is allowed to have $\mathit{Var}(r) \not\subseteq \mathit{Var}(\ell)$, but fresh variables in the right-hand side may only be instantiated with *values*. This is done to model user input or random choice. Otherwise, variables outside the constraint may be instantiated by any term; we do not impose strategies like innermost or call-by-value reduction.

Definition 2.10. We assume given a set of rules $\mathcal{R}$ and let $\mathcal{R}_{\mathtt{calc}}$ be the set $\{f(x_1, \ldots, x_n) \to y\ [y = f(\overrightarrow{x})] \mid f : [\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa \in \Sigma_{theory} \setminus \mathcal{V}al\}$ (writing $\overrightarrow{x}$ for $x_1, \ldots, x_n$). The *rewrite relation* $\to_{\mathcal{R}}$ is a binary relation on terms, defined by

$$C[\ell\gamma]\ \to_{\mathcal{R}}\ C[r\gamma]\ \text{ if } \ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{\mathtt{calc}} \text{ and } \gamma \text{ respects } \ell \to r\ [\varphi].$$

Here, C is a context with exactly one hole. We say that the reduction occurs at position p if $C = C[\Box]_p$. Let $s \leftrightarrow_{\mathcal{R}} t$ if $s \to_{\mathcal{R}} t$ or $t \to_{\mathcal{R}} s$. A reduction step with $\mathcal{R}_{\mathtt{calc}}$ is called a *calculation*. A term is in *normal form* if it cannot be reduced with $\to_{\mathcal{R}}$. We say that t is a *normal form of* s if $s \to^*_{\mathcal{R}} t$ and t is a normal form. The relation $\to_{\mathcal{R}}$ is *confluent* if whenever $s \to^*_{\mathcal{R}} t$ and $s \to^*_{\mathcal{R}} t'$ there exists also some u with $t \to^*_{\mathcal{R}} u$ and $t' \to^*_{\mathcal{R}} u$.

We usually call the elements of $\mathcal{R}_{\mathtt{calc}}$ rules—or *calculation rules*–even though their left-hand side is a logical term. Note that if $\to_{\mathcal{R}}$ is confluent, every term has at most one normal form (intuitively, then $\mathcal{R}$ is deterministic with respect to big-step semantics).

Definition 2.11. For $f(\ell_1, \ldots, \ell_n) \to r\ [\varphi] \in \mathcal{R}$, we call f a *defined symbol*; nondefined elements of Σ_{terms} and all values are *constructors*. Let $\mathcal{D}$ be the set of all defined symbols and $\mathcal{C}ons$ the set of constructors. A term in $\mathcal{T}erms(\mathcal{C}ons, \mathcal{V})$ is a *constructor term*.

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

Example 2.12. To implement an LCTRS calculating the *factorial* function, we use the signature Σ from Example 2.8 and the following rules:

$$\mathcal{R}_{\mathsf{fact}} = \{\mathsf{fact}(x) \to 1\ [x \leq 0],\ \ \mathsf{fact}(x) \to x * \mathsf{fact}(x - 1)\ [\neg(x \leq 0)]\}.$$

Using calculation steps, a term $3 - 1$ reduces to 2 in 1 step (using the calculation rule $x - y \to z\ [z = x - y]$), and $3 * (2 * (1 * 1))$ reduces to 6 in 3 steps. Using also the rules in $\mathcal{R}_{\mathsf{fact}}$, $\mathsf{fact}(3)$ reduces in 10 steps to 6.

Example 2.13. To implement an LCTRS calculating the sum of elements in an array, let $\mathcal{I}_{\mathsf{bool}} = \mathbb{B}$, $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$, $\mathcal{I}_{\mathsf{array(int)}} = \mathbb{Z}^*$, so $\mathsf{array(int)}$ is mapped to finite-length integer sequences. Let $\Sigma_{theory} = \Sigma^{int}_{theory} \cup \{\mathsf{size} : [\mathsf{array(int)}] \Rightarrow \mathsf{int},\ \mathsf{select} : [\mathsf{array(int)} \times \mathsf{int}] \Rightarrow \mathsf{int}\} \cup \{\mathsf{a} \mid a \in \mathbb{Z}^*\}$. (We do *not* encode arrays as lists: every "array"—integer sequence—a corresponds to a unique symbol a.) The interpretation function $\mathcal{J}$ behaves on Σ^{int}_{theory} as usual, maps the values a to the corresponding integer sequence, and has

$$\mathcal{J}_{\mathsf{size}}(a) = k \text{ if } a = \langle n_0, \ldots, n_{k-1}\rangle \quad \mathcal{J}_{\mathsf{select}}(a, i) = \begin{array}{ll} n_i & \text{if } a = \langle n_0, \ldots, n_{k-1}\rangle \text{ and } 0 \leq i < k \\ 0 & \text{otherwise.}\end{array}$$

In addition, let $\Sigma_{terms} = \{\mathsf{sum}, \mathsf{sum0} : [\mathsf{array(int)}] \Rightarrow \mathsf{int}\} \cup \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\} \cup \{\mathsf{a} \mid a \in \mathbb{Z}^*\}$, and let $\mathcal{R}$ consist of

$$\begin{array}{lllll} \mathsf{sum}(x) & \to & \mathsf{sum0}(x, \mathsf{size}(x) - 1) & & \\ \mathsf{sum0}(x, k) & \to & \mathsf{select}(x, k) + \mathsf{sum0}(x, k - 1) & [k \geq 0] & \\ \mathsf{sum0}(x, k) & \to & 0 & [k < 0]. & \end{array}$$

Note that this implementation differs from the ones in Example 1.1, because there we analyzed encodings of imperative programs; on C level, there is no functionality for the programmer to explicitly query the size of an array. Here, we avoided boundary checks.

Values are new in LCTRSs compared to older styles of constrained rewriting. These representatives of the underlying theory are always *constants* (constructor symbols that do not take arguments), even if they represent complex structures, as seen in Example 2.13. Note that variables in a rule's constraint must be instantiated by values; for instance, in Example 2.12, a term $\mathsf{fact}(1 + 2)$ must be reduced by a calculation first. We also do not match modulo theories (e.g., we do not equate $0 + (x + y)$ with $y + x$ for matching).

Differences to Kop and Nishida [2013]. In the original definition of LCTRSs, variables in $\mathcal{V}$ are unsorted, and a separate *variable environment* is used for typing. In addition, $\to_{\mathcal{R}}$ is there defined as the union of two relations $\to_{\mathtt{rule}}$ and $\to_{\mathtt{calc}}$ rather than including $\mathcal{R}_{\mathtt{calc}}$. These changes give equivalent results, but the current definitions cause less bookkeeping. A larger difference is the restriction on rules: in Kop and Nishida [2013], left-hand sides must have a root symbol in $\Sigma_{terms} \setminus \Sigma_{theory}$. We follow Kop [2013] and Kop and Nishida [2014] in weakening this (only asking that they are not logical terms).

2.2. Quantification

The definition of LCTRSs does not permit constraints with quantifiers (constraints are terms, and first-order rewriting does not allow quantifiers in terms). In, for instance, an LCTRS over integers and arrays, which has $\mathsf{addtoend} : [\mathsf{int} \times \mathsf{array(int)}] \Rightarrow \mathsf{array(int)} \in \Sigma_{theory}$ and $\mathsf{extend} : [\mathsf{array(int)} \times \mathsf{int}] \Rightarrow \mathsf{array(int)} \in \Sigma_{terms}$, we cannot specify a rule like

$$\mathsf{extend}(arr, x) \to \mathsf{addtoend}(x, arr)\ [\forall y \in \{0, \ldots, \mathsf{size}(arr) - 1\}(x \neq \mathsf{select}(arr, y))].$$

However, one of the key features of LCTRSs is that theory symbols, including predicates, are not confined to a fixed list. Therefore, we can add a new symbol to Σ_{theory} (and $\mathcal{J}$). For the extend rule, we might introduce a symbol $\mathsf{notin} : [\mathsf{int} \times \mathsf{array(int)}] \Rightarrow \mathsf{bool}$ with $\mathcal{J}_{\mathsf{notin}}(u, \langle a_0, \ldots, a_{n-1}\rangle) = \top$ if and only if for all i, $u \neq a_i$, and replace the constraint by $\mathsf{notin}(x, arr)$. This generates exactly the same reduction relation as the original rule.

Thus, we can permit quantifiers in the constraints of rules and also on right-hand sides of rules, as an intuitive notation for fresh predicates. However, an *unbounded* quantification would likely not be useful, as it would give an undecidable relation $\to_{\mathcal{R}}$.

Comment: One might argue that adding symbols like this is problematic in practice: no SMT solver will support new symbols like notin. However, for the technique, this makes no difference. In an implementation, we might allow quantifiers as syntactic sugar (and pass the same sugar to the SMT solver) or add a layer on top of the SMT solver that translates the new symbol(s), replacing, for instance, `(notin u a)` with `(forall ((x Int)) (distinct u (select a x)))`.

2.3. Rewriting Constrained Terms

In LCTRSs, the objects of study are *terms*, with $\to_{\mathcal{R}}$ defining the relation between them. However, for analysis, it is often useful to consider constrained terms.

Definition 2.14. A *constrained term* is a pair $s\,[\varphi]$ of a term s and a constraint φ. We say that $s\,[\varphi]$ and $t\,[\psi]$ are *equivalent*, notation $s\,[\varphi] \sim t\,[\psi]$, if for all substitutions γ that respect φ there is a substitution δ that respects ψ such that $s\gamma = t\delta$, and vice versa.

Intuitively, a constrained term $s\,[\varphi]$ represents all terms $s\gamma$ where γ respects φ and can be used to reason about such terms. Equivalent constrained terms represent the same set of terms. For example, $\mathsf{f}(0)\,[\mathsf{true}] \sim \mathsf{f}(x)\,[x = 0]$, and $\mathsf{g}(x, y)\,[x > y] \sim \mathsf{g}(z, u)\,[u \le z - 1]$. Note that $s\,[\varphi] \sim s\,[\psi]$ if and only if $\forall\overrightarrow{x}\,(\exists\overrightarrow{y}\,(\varphi) \leftrightarrow \exists\overrightarrow{z}\,(\psi))$ holds, where $\mathit{Var}(s) = \{\overrightarrow{x}\}$, $\mathit{Var}(\varphi) \setminus \mathit{Var}(s) = \{\overrightarrow{y}\}$ and $\mathit{Var}(\psi) \setminus \mathit{Var}(s) = \{\overrightarrow{z}\}$.

Definition 2.15. For a rule $\rho := \ell \to r\ [\psi] \in \mathcal{R} \cup \mathcal{R}_{\mathtt{calc}}$ and position q, we let $s\,[\varphi] \to_{\rho,q} t\,[\varphi]$ if there exists a substitution γ such that $s_{|q} = \ell\gamma$, $t = s[r\gamma]_q$, $\gamma(x)$ is a value or variable in $\mathit{Var}(\varphi)$ for all $x \in \mathit{LVar}(\ell \to r\ [\psi])$, and $\varphi \Rightarrow (\psi\gamma)$ is valid. Let $s\,[\varphi] \to_{\mathtt{base}} t\,[\varphi]$ if $s\,[\varphi] \to_{\rho,q} t\,[\varphi]$ for some ρ, q. The relation $\to_{\mathcal{R}}$ on constrained terms is defined as $\sim \cdot \to_{\mathtt{base}} \cdot \sim$. We say that $s\,[\varphi] \to_{\mathcal{R}} t\,[\psi]$ *at position* q *by rule* ρ if $s\,[\varphi] \sim \cdot \to_{\rho,q} \cdot \sim t\,[\psi]$.

Example 2.16. In the LCTRS from Example 2.12, we have $\mathsf{fact}(x)\,[x > 3] \to_{\mathcal{R}} x * \mathsf{fact}(x - 1)\,[x > 3]$. Now we can use a calculation rule $x - y \to z\ [z = x - y]$, with a nonempty $\sim$-step, as follows: $x*\mathsf{fact}(x-1)\,[x > 3] \sim x*\mathsf{fact}(x-1)\,[x > 3 \wedge z = x-1] \to_{\mathtt{base}} x * \mathsf{fact}(z)\,[x > 3 \wedge z = x - 1]$. The $\sim$-relation holds because indeed $\forall x(x > 3 \leftrightarrow \exists z(x > 3 \wedge z = x - 1))$.

Example 2.17. The $\sim$-relation also allows us to reformulate the constraint after a reduction. For example, with the rule $\mathsf{f}(x) \to \mathsf{g}(y)\ [y > x]$, we have $\mathsf{f}(x)\,[x > 3] \sim \mathsf{f}(x)\,[x > 3 \wedge y > x] \to_{\mathtt{base}} \mathsf{g}(y)\,[x > 3 \wedge y > x] \sim \mathsf{g}(y)\,[y > 4]$. We do not have that $\mathsf{f}(x)\ [\mathsf{true}] \to_{\mathcal{R}} \mathsf{g}(x + 1)\ [\mathsf{true}]$, as $x + 1$ cannot be instantiated to a value.

Example 2.18. A constrained term does not always need to be reduced in the most general way. With the rule $\mathsf{f}(x) \to \mathsf{g}(y)\ [y > x]$, we have $\mathsf{f}(0)\ [\mathsf{true}] \sim \mathsf{f}(0)\ [y > 0] \to_{\mathtt{base}} \mathsf{g}(y)\ [y > 0]$, but we also have $\mathsf{f}(0)\ [\mathsf{true}] \sim \mathsf{f}(0)\ [1 > 0] \to_{\mathtt{base}} \mathsf{g}(1)\ [1 > 0] \sim \mathsf{g}(1)\ [\mathsf{true}]$.

As intended, constrained reductions give information about usual reductions.

THEOREM 2.19. *If* $s\,[\varphi] \to_{\mathcal{R}} t\,[\psi]$*, then for all substitutions* γ *that respect* φ *there exists* δ *that respects* ψ *such that* $s\gamma \to_{\mathcal{R}} t\delta$*. Both steps use the same rule and position.*

PROOF. We first observe (**): *if* $u\,[\xi] \to_{\mathtt{base}} q\,[\xi]$*, then for any substitution* γ *that respects* ξ *also* $u\gamma \to_{\mathcal{R}} q\gamma$. Proof: if $u\,[\xi] \to_{\mathtt{base}} q\,[\xi]$, then there are $p, \ell \to r\ [c]$ and δ such that $u_{|p} = \ell\delta$, $q = u[r\delta]_p$, $\delta(x) \in \mathit{Var}(\xi) \cup \mathcal{V}al$ for all $x \in \mathit{LVar}(\ell \to r\ [c])$ and $\xi \Rightarrow (c\delta)$ is valid. With $\eta = \gamma \circ \delta$, we have $(u\gamma)_{|p} = u_{|p}\gamma = \ell\delta\gamma = \ell\eta$ and $q\gamma = u[r\delta]_p\gamma = (u\gamma)[r\delta\gamma]_p = (u\gamma)[r\eta]_p$. We also have $\eta(x) = \delta(x)\gamma \in \mathcal{V}al$ for $x \in \mathit{LVar}(\ell \to r\ [c])$ because γ respects ξ and, since $[\![\xi\gamma]\!] = \top$ and $\xi \Rightarrow (c\delta)$ is valid, also $[\![(c\delta)\gamma]\!] = [\![c\eta]\!] = \top$. Thus, indeed $u\gamma \to_{\mathcal{R}} q\gamma$.

Now suppose that $s\,[\varphi] \to_{\mathcal{R}} t\,[\varphi]$, so $s\,[\varphi] \sim s'\,[\xi] \to_{\mathtt{base}} t'\,[\xi] \sim t\,[\psi]$, and let γ respect φ. By definition of $\sim$, there is some substitution η that respects ξ such that $s\gamma = s'\eta$. By (**), $s'\eta \to_{\mathcal{R}} t'\eta$. Again by definition of $\sim$, we find δ that respects ψ such that $t'\eta = t\delta$. □

THEOREM 2.20. *If $s\,[\varphi] \to_{\mathcal{R}} t\,[\psi]$, then for all substitutions δ that respect ψ there exists γ that respects φ such that $s\gamma \to_{\mathcal{R}} t\delta$. Both steps use the same rule and position.*

PROOF. Parallel to the proof of Theorem 2.19, if $s\,[\varphi] \sim s'\,[\xi] \to_{\mathtt{base}} t'\,[\xi] \sim t\,[\psi]$, then by definition of $\sim$ there are suitable η, γ such that $t\delta = t'\eta \leftarrow_{\mathcal{R}} s'\eta = s\gamma$. □

Comment: The relation $\to_{\mathcal{R}}$ on constrained terms is not stable. For instance, in the system from Example 2.18, we can derive $\mathsf{f}(x)\,[\mathsf{true}] \to_{\mathcal{R}} \mathsf{g}(x)\,[\mathsf{true}]$ even though $\mathsf{f}(0)\,[\mathsf{true}] \not\to_{\mathcal{R}} \mathsf{g}(0)\,[\mathsf{true}]$. This is because the variables in a constrained term $s\,[\varphi]$ are fully changeable; one can see variables in $Var(s)$ as universal and the others as existential. This is not problematic, as we do not instantiate constrained terms; to reason with constrained reduction, we only use Theorems 2.19 and 2.20.

3. TRANSFORMING IMPERATIVE PROGRAMS INTO THE LCTRS

Equivalence-preserving transformations of imperative programs into constrained rewriting systems operating on integers have been investigated in works such as Falke and Kapur [2009], Falke et al. [2011], and Furuichi et al. [2008]; more generally, such translations from imperative to functional programs have been investigated at least since McCarthy [1960]. Although these works use different definitions of constrained rewriting, the proposed transformations can be adapted to produce LCTRSs that operate on integers (i.e., use Σ_{theory} as in Example 2.12). In addition, we can extend the ideas to also handle more advanced programming structures, such as arrays and exceptions.

In this section, we discuss several ideas toward a translation from C to LCTRS. A more detailed and formal treatment of the limitation to integers and one-dimensional integer arrays is available online, along with an implementation, at http://www.trs.css.i.nagoya-u.ac.jp/c2lctrs/.

Given the extensiveness of the C specification, we will not attempt to prove that the result of our transformation corresponds to the origin. Instead, we shall rely on an appeal to intuition. An advantage is that the same ideas apply to other programming languages—for example, we should be able to use similar translations for Python or Java.

3.1. Transforming Simple Integer Functions

The base form of the transformation—limited to integer functions with no global variables or function calls—is very similar to the transformations for integer TRSs in Falke and Kapur [2009], Falke et al. [2011], and Furuichi et al. [2008]. Each function is transformed separately. We introduce a function symbol for every statement (including declarations), which operates on the variables in scope. The transition from one statement to another is encoded as a rule, with assignments reflected by argument updates in the right-hand side, and conditions by the constraint. Return statements are encoded by reducing to an expression $\mathsf{return}_f(e)$, where $\mathsf{return}_f : [\mathsf{int}] \Rightarrow \mathsf{result}_f$ is a constructor.

Example 3.1. Consider the following C function and its translation.

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

$$
\begin{array}{rcll}
\mathsf{fact}(x) & \to & \mathsf{u}_1(x,1) & \\
\mathsf{u}_1(x,z) & \to & \mathsf{u}_2(x,z,1) & \\
\mathsf{u}_2(x,z,i) & \to & \mathsf{u}_3(x,z,i) & [i \le x] \\
\mathsf{u}_2(x,z,i) & \to & \mathsf{u}_5(x,z) & [\neg(i \le x)] \\
\mathsf{u}_3(x,z,i) & \to & \mathsf{u}_4(x,z*i,i) & \\
\mathsf{u}_4(x,z,i) & \to & \mathsf{u}_2(x,z,i+1) & \\
\mathsf{u}_5(x,z) & \to & \mathsf{return}_{\mathsf{fact}}(z) &
\end{array}
$$

For Σ_{theory}, we assume the standard integer signature; Σ_{terms} contains fact, all u_i, and the constructor return_f, all of which have output sort result_f and argument sorts int.

A realistic translation of C code must also handle the absence of a Boolean data type, operator precedence, and expressions with side effects (e.g., a loop condition $--x$). All of this is easily doable[1] (and included in our implementation); however, for the sake of brevity, we will not go into detail here.

Finally, the generated system is optimized to make it more amenable to analysis:[2]

—rules are combined where possible—for example, replacing a pair of rules $\ell \to \mathsf{u}(r_1,\dots,r_n)\ [\varphi]$ and $\mathsf{u}(x_1,\dots,x_n) \to s$ [true] by $\ell \to s[x_1 := r_1,\dots,x_n := r_n]$ if u is not used elsewhere;
—unused arguments of function symbols are removed, such as the second (but not the first!) argument of u in an LCTRS with rules $\mathsf{u}(x,y,z) \to \mathsf{u}(x-1,y+1,z*2)\ [x > 0]$ and $\mathsf{u}(x,y,z) \to \mathsf{return}(z)\ [\neg(x > 0)]$;
—constraints are simplified—for instance, replacing $\neg(x > 0)$ by $x \le 0$ in the preceding rules.

We will use these optimizations also for the extended transformations of Sections 3.2 through 3.6.

Comment: When *time complexity* (defined as, e.g., the number of certain calculation steps) is considered, the argument removal step is dangerous, as it may remove calculations. In such cases, we would use a different simplification method.

Example 3.2. Optimizing the LCTRS from Example 3.1, we obtain

$$
\begin{array}{rcll}
\mathsf{fact}(x) & \to & \mathsf{u}_2(x,1,1) & \\
\mathsf{u}_2(x,z,i) & \to & \mathsf{u}_2(x,z*i,i+1) & [i \le x] \\
\mathsf{u}_2(x,z,i) & \to & \mathsf{return}_{\mathsf{fact}}(z) & [i > x].
\end{array}
$$

Differences from older work. In contrast to existing transformations to integer TRSs (e.g., Falke and Kapur [2009], Falke et al. [2011], and Furuichi et al. [2008]), we do not consider *basic blocks* but simply create rules for every statement; this gives no substantial difference after optimization. Additionally, return_f is new here: in the work by Falke et al., the return statement is omitted, as they focus on *termination*, whereas in Furuichi et al., the final term reduces directly to the return value (e.g., $\mathsf{u}_4(x,z) \to z + x\ [x \le 0]$).

3.2. Noninteger Data Types

Integers are not special: as the definition of LCTRSs permits arbitrary theories, we can handle any data type in C. For instance, we might interpret `double` as either real

[1]This is discussed in the formal treatment at http://www.trs.css.i.nagoya-u.ac.jp/c2lctrs/formal.pdf.
[2]Variations of such preprocessing steps preserving the properties of interest to simplify the output of an automatic translation are fairly standard in program analysis (e.g., see Albert et al. [2008], Alpuente et al. [2007], Beyer et al. [2009], Falke et al. [2011], Giesl et al. [2017], and Spoto et al. [2009]).

numbers or double-precision floating point numbers; this choice is left to the user and may vary by application. The only requirement is that a suitable theory signature—with the corresponding SMT solver if the system is to be analyzed automatically—is available. The translation is straightforward, with the only difficulty that type casts must be made explicit, and we need to use separate symbols such as $+.$ for `double` addition.

Example 3.3. Consider the following C function and its translation.

```
double halfsum(double thold) {
  double ret = 0.0;
  for (int d = 2; d < 100;
                  d *= 2) {
    ret += 1.0 / d;
    if (ret > thold) return ret;
} }
```

$$
\begin{array}{lcll}
\mathsf{halfsum}(t) & \to & \mathsf{u}_2(t, 0.0, 2) & \\
\mathsf{u}_2(t,r,d) & \to & \mathsf{u}_4(t, r +. \ 1.0/\mathsf{todouble}(d), d) & [d < 100] \\
\mathsf{u}_2(t,r,d) & \to & \mathsf{return}_{\mathsf{halfsum}}(rnd) & [d \geq 100] \\
\mathsf{u}_4(t,r,d) & \to & \mathsf{return}_{\mathsf{halfsum}}(r) & [r >. \ t] \\
\mathsf{u}_4(t,r,d) & \to & \mathsf{u}_2(t,r,d*2) & [r \leq. \ t]
\end{array}
$$

This demonstrates both an explicit cast and one possible way to handle an undefined return value (by a fresh variable, which may be instantiated with a random value).

3.3. Error Handling

The transformation of Section 3.1 does not fully reflect the original C program: as computers have limited memory, integers are internally represented as *bitvectors*. To address this, we could change the theory. Rather than using $\mathbb{Z}$, we let $\mathcal{V}al_{\mathsf{int}} = \{\mathtt{MININT}, \ldots, \mathtt{MAXINT}\}$ and make $\mathcal{J}_+$, $\mathcal{J}_-$, and $\mathcal{J}_*$ wrap around (e.g., $\mathcal{J}_-(\mathtt{MININT}, 1) = \mathtt{MAXINT}$). The resulting LCTRS has the same rules but acts more closely to the real program behavior.

However, integer overflow is often indicative of an *error*. Indeed, in C an overflow for the type `int` leads to undefined behavior (which also surfaces in optimizing compilers such as `gcc` or `clang`). To model this (or other instances of undefined behavior in C, such as a missing `return` statement), we will reduce to a special error state.

Thus, for every rule $\mathsf{u}_i(x_1, \ldots, x_n) \to r \ [\varphi]$, if this rule represents a transition where an error may occur under condition τ, then we split it in two:

$$\mathsf{u}_i(x_1, \ldots, x_n) \to r \ [\varphi \wedge \neg\tau] \qquad \mathsf{u}_i(x_1, \ldots, x_n) \to \mathsf{error}_f \ [\varphi \wedge \tau].$$

As usual, we simplify the resulting constraint (writing, e.g., $x < 0$ instead of $\neg(x \geq 0)$).

Example 3.4. Continuing Example 3.2, we generate the following rewrite rules.

$$
\begin{array}{lcll}
\mathsf{fact}(x) & \to & \mathsf{u}_2(x, 1, 1) & \\
\mathsf{u}_2(x,z,i) & \to & \mathsf{u}_2(x, z*i, i+1) & [i \leq x \wedge z*i \leq \mathtt{MAXINT} \wedge z*i \geq \mathtt{MININT} \wedge i+1 \leq \mathtt{MAXINT}] \\
\mathsf{u}_2(x,z,i) & \to & \mathsf{error}_{\mathsf{fact}} & [i \leq x \wedge (z*i > \mathtt{MAXINT} \vee z*i < \mathtt{MININT} \vee i+1 > \mathtt{MAXINT})] \\
\mathsf{u}_2(x,z,i) & \to & \mathsf{return}_{\mathsf{fact}}(z) & [i > x]
\end{array}
$$

Note that we could easily model assertions and `throw` statements for exceptions in the same way. Division by zero is handled in a similar way.

We can choose whether to add error transitions before or after the simplification step. The distinction is important: when simplifying, calculations that do not contribute to the final result are thrown away. In the case of overflow errors, it may seem reasonable to consider the postsimplification rules, as we did in Example 3.4. In the case of for instance division by zero, we should add the errors to the presimplification rules.

Comment: When transforming a function into an LCTRS, we can choose what errors to model. For instance, we could ignore overflows (effectively assuming unbounded integers) but still test for division by zero. We could also let error_f be a constructor that takes an argument—for instance, $\mathsf{error}_f : [\mathsf{Errors}] \Rightarrow \mathsf{result}_f \in \Sigma_{terms}$, where Errors is a sort with constructors IntegerOverflow, DivisionByZero, and so on.

3.4. Global Variables

Thus far, we have considered very *local* code: a function never calls other functions or modifies global variables. By altering the return constructors, we easily change the latter: we assume that a function symbol is given all global variables that it uses as input, and that it returns those global variables it alters as output, along with its return value. This change also allows for nonredundant `void` functions.

Example 3.5. Consider the following short program and its (simplified) translation.

```
int best;
int up(int x) {
  if (x > best)  {best = x; return 1;}
  return 0;
}
```

$$\mathsf{up}(b, x) \to \mathsf{return}_{\mathsf{up}}(x, 1) \ [x > b]$$
$$\mathsf{up}(b, x) \to \mathsf{return}_{\mathsf{up}}(b, 0) \ [x \leq b]$$

3.5. Function Calls

Next, let us consider *function calls*. A difficulty is that they may occur in an expression (e.g., $\mathsf{fact}(3) + 5$) that is not well sorted in the corresponding LCTRS: $\mathsf{fact}(3)$ has sort $\mathsf{result}_{\mathsf{fact}}$, not int. To avoid this issue, and to propagate errors, we split off function calls occurring inside expressions other than $var = func(arg_1, \ldots, arg_n)$ and store their return value into a temporary variable. Take the following as an example.

```
int ncr(int x, int y) {
  int a = fact(x);
  int b = fact(y) * fact(x - y);
  return a / b;
}
```

$\Longrightarrow$

```
int ncr(int x, int y) {
  int a = fact(x);
  int tmp1 = fact(y);
  int tmp2 = fact(x - y);
  int b = tmp1 * tmp2;
  return a / b;
}
```

This change may cause declarations at places in the function where a C compiler would not accept them, but for the translation, this is no issue. We translate the resulting function by executing function calls in a separate parameter and using a separate step to examine the outcome of a function call and assign it to the relevant variable(s).

Example 3.6. The preceding `ncr` program is transformed to the following optimized LCTRS (where we test for division by zero but not integer overflow for simplicity).

$$\mathsf{ncr}(x, y) \to \mathsf{u}_2(x, y, \mathsf{fact}(x))$$
$$\mathsf{u}_2(x, y, \mathsf{return}_{\mathsf{fact}}(k)) \to \mathsf{u}_3(x, y, k, \mathsf{fact}(y))$$
$$\mathsf{u}_3(x, y, a, \mathsf{return}_{\mathsf{fact}}(k)) \to \mathsf{u}_4(x, y, a, k, \mathsf{fact}(x - y))$$
$$\mathsf{u}_4(x, y, a, t_1, \mathsf{return}_{\mathsf{fact}}(k)) \to \mathsf{error}_{\mathsf{ncr}}$$
$$\mathsf{u}_4(x, y, a, t_1, \mathsf{return}_{\mathsf{fact}}(k)) \to \mathsf{return}_{\mathsf{ncr}}(a \ \mathsf{div} \ (t_1 * k)) \ [t_1 * k \neq 0]$$
$$\mathsf{u}_2(x, y, \mathsf{error}_{\mathsf{fact}}) \to \mathsf{error}_{\mathsf{ncr}}$$
$$\mathsf{u}_3(x, y, a, \mathsf{error}_{\mathsf{fact}}) \to \mathsf{error}_{\mathsf{ncr}}$$
$$\mathsf{u}_4(x, y, a, t_1, \mathsf{error}_{\mathsf{fact}}) \to \mathsf{error}_{\mathsf{ncr}} \ [t_1 * k = 0]$$

3.6. Statically Allocated Arrays

Finally, let us consider arrays. After seeing Example 1.1 and the way side effects were handled in Section 3.4, this is largely as expected. For now, we will not consider aliasing.

To start, we must fix a theory signature and corresponding interpretations. For a given theory sort ι that admits at least one value, say 0_ι, let $\mathsf{array}(\iota)$ be a new sort and $\mathcal{I}_{\mathsf{array}(\iota)} = \mathcal{I}_\iota^*$—so each value corresponds to a finite sequence. We introduce the following theory symbols (in addition to Σ_{theory}^{int} and other desired theories):

—$\mathsf{size}_\iota : [\mathsf{array}(\iota)] \Rightarrow \mathsf{int}$: we define $\mathcal{J}_{\mathsf{size}_\iota}(a)$ as the length of the sequence a.
—$\mathsf{select}_\iota : [\mathsf{array}(\iota) \times \mathsf{int}] \Rightarrow \iota$: if $a = \langle a_0, \ldots, a_{n-1}\rangle$, we define $\mathcal{J}_{\mathsf{select}_\iota}(a, k) = a_k$ if $0 \leq k < n$ and $\mathcal{J}_{\mathsf{select}_\iota}(a, k) = 0_\iota$ otherwise.
—$\mathsf{store}_\iota : [\mathsf{array}(\iota) \times \mathsf{int} \times \iota] \Rightarrow \mathsf{array}(\iota)$: if $a = \langle a_0, \ldots, a_{n-1}\rangle$, we 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.

We will usually omit the subscript ι when the sort is clear from context.

Our arrays are different from SMT-LIB (see http://www.smt-lib.org/), where arrays are functions from one (possibly infinite) domain to another. For program analysis, finite-length sequences seem practical instead. SMT problems on our arrays can be translated to SMT-LIB format using an additional integer variable a_{size} for the size of an array a and universal quantification to set entries outside the array to a fixed value.

We encode *lookups* $a[i]$ as $\mathsf{select}(a, i)$; for *assignments* $a[i] = e$, we replace a by $\mathsf{store}(a, i, e)$. To ensure correctness here, we add boundary checks to the constraint and reduce to error_f if such a check is not satisfied. After an assignment, the updated variable is included in the return value since the underlying memory of the array was altered.

Example 3.7. Consider the following C implementation of the strcpy function, which copies the contents of `original` into the array `goal`, until a 0 is reached.

```
void strcpy(char goal[], char original[]) {
  int i = 0;
  for (; original[i] != 0; i++) goal[i] = original[i];
  goal[i] = 0;
}
```

For simplicity, we think of strings as integer arrays (although alternative choices for $\mathcal{I}_{\mathsf{char}}$ make little difference). The function never updates `original` but may update `goal`, so the return value must include the latter. We obtain the following LCTRS.

$$
\begin{array}{rcll}
\mathsf{strcpy}(gl, org) & \to & \mathsf{v}(gl, org, 0) & \\
\mathsf{v}(gl, org, i) & \to & \mathsf{error}_{\mathsf{strcpy}} & [i < 0 \vee i \geq \mathsf{size}(org)] \\
\mathsf{v}(gl, org, i) & \to & \mathsf{w}(gl, org, i) & [0 \leq i < \mathsf{size}(org) \wedge \mathsf{select}(org, i) = 0] \\
\mathsf{v}(gl, org, i) & \to & \mathsf{error}_{\mathsf{strcpy}} & [0 \leq i < \mathsf{size}(org) \wedge \mathsf{select}(org, i) \neq 0 \wedge i \geq \mathsf{size}(gl)] \\
\mathsf{v}(gl, org, i) & \to & \mathsf{v}(\mathsf{store}(gl, i, \mathsf{select}(org, i)), org, i + 1) & \\
 & & & [0 \leq i < \mathsf{size}(org) \wedge \mathsf{select}(org, i) \neq 0 \wedge i < \mathsf{size}(gl)] \\
\mathsf{w}(gl, org, i) & \to & \mathsf{error}_{\mathsf{strcpy}} & [i < 0 \vee i \geq \mathsf{size}(gl)] \\
\mathsf{w}(gl, org, i) & \to & \mathsf{return}_{\mathsf{strcpy}}(\mathsf{store}(gl, i, 0)) & [0 \leq i < \mathsf{size}(gl)]
\end{array}
$$

Here, the notation $0 \leq i < \mathsf{size}(org)$ is shorthand for $0 \leq i \wedge i < \mathsf{size}(org)$. Note that this LCTRS could be further simplified by combining the third rule with the last two rules.

Comment: It should now be clear how the systems from Section 1.1 have been translated from C code to LCTRSs. The only deviation is that there we have included the array *arr* in the return value of sum1, sum2, and sum4, which is not necessary as it is not modified in these cases. This was done to allow for a direct comparison with sum3, where the array is modified. In addition, the return and error symbols in these examples are not indexed for the same reason.

3.7. Dynamically Allocated Arrays and Aliasing

The transformation in Section 3.6 allows us to abstract from the underlying memory model when encoding arrays. This makes analysis easier but does not allow for aliasing or pointer arithmetic beyond accessing an array element. As a result, properties we prove about strcpy from Example 3.7 might fail to hold for a call like $\mathsf{strcpy}(a, a)$.

As we seek to handle only part of the language, this does not need to be an issue; in practice, a fair number of programs are written without explicit pointer use and with easily removable aliasing only. For example, we might replace $\mathsf{strcpy}(a, a)$ by $\mathsf{strcpy}'(a)$, and create new rules for strcpy' by collapsing the variables in the rules for strcpy. To handle programs with more sophisticated pointer use, including dynamically allocated arrays, we can encode the memory as a list of arrays and pass this along as a variable. This is somewhat beyond the scope of this article but is explored later in Appendix A.2.

3.8. Remarks

The treatment in this section is both informal and incomplete: we have discussed only a fraction of the C language—albeit an important fraction for verification. We believe that these ideas easily extend further, such as with the `switch` statement, user-defined data structures, or standard library functions, as well as compiler-specific choices. It is important to note that the translation gives several choices. Most pertinently, we saw the choices of what sort interpretations to use (e.g., whether `int` should be mapped to the set of integers or bitvectors) and what errors to consider.

In this article, and in line with our automatic translation at http://www.trs.css.i.nagoya-u.ac.jp/c2lctrs/, we have chosen to work with real integers and not test for overflows. We also do not permit aliasing. By avoiding the more sophisticated translation steps, we obtain LCTRSs that are correspondingly easier to analyze.

The LCTRSs from this transformation are well behaved: all rules are left linear and nonoverlapping,[3] and they have the property that all ground terms can be reduced or are constructor terms. Rules $\ell \to r\ [\varphi]$ can have variables in r or φ that do not occur in ℓ: this is mostly due to unspecified values in the C code. Where such variables do not occur—or are removed in the optimization step—the resulting LCTRSs are confluent.

4. REWRITING INDUCTION FOR THE LCTRS

In this section, we adapt the inference rules from Reddy [1990], Falke and Kapur [2012], and Sakata et al. [2009] to inductive theorem proving with LCTRSs. This provides the core theory for rewriting induction, strengthened with two generalization techniques in Section 5.

We start by listing some restrictions that we need to impose on LCTRSs for the method to work (Section 4.1). Then we provide the theory for the technique (Section 4.2) and some illustrative examples (Section 4.3). Compared to older definitions of rewriting induction, we make several changes to best handle the new formalism. We complete by proving correctness (Section 4.4).

4.1. Restrictions

For rewriting induction to be successful, we need to impose certain restrictions.

Definition 4.1. In the following, we limit interest to LCTRSs that satisfy restrictions (1) through (4):

[3]Nonoverlappingness means that for every term s and rule $\rho : \ell \to r\ [\varphi]$ such that s reduces with ρ at the root position: (a) there are no other rules ρ' such that s reduces with ρ' at the root position, and (b) if s reduces with any rule at a nonroot position q, then q is not a position of ℓ. For our translations, this holds because (a) rules with the same defined symbol have either incompatible constraints or nonunifiable arguments, and (b) in a rule $f(\ell_1, \ldots, \ell_n) \to r\ [\varphi]$, the terms ℓ_i do not contain defined or calculation symbols.

(1) all core theory symbols are present in Σ_{theory}: $\Sigma_{theory} \supseteq \Sigma^{core}_{theory}$;
(2) the LCTRS is *terminating*: there is no infinite reduction $s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} \cdots$;
(3) the system is *quasi-reductive*: i.e., for every ground term s, either $s \in \mathcal{T}erms(Cons, \emptyset)$ (we say that s is a *ground constructor term*) or there is some t such that $s \to_{\mathcal{R}} t$;
(4) there are ground terms of every sort occurring in Σ.

Property 1 is the standard assumption from Section 2. We will need symbols such as $=$, $\wedge$, and $\Rightarrow$ to add new information to a constraint. Termination (property 2) essentially indicates that a program cannot run indefinitely; this is crucial for our inductive reasoning, as the method uses induction on an extension of $\to_{\mathcal{R}}$ on terms.

Property 3 indicates that an evaluation cannot get "stuck"—roughly, that pattern matching and case analysis are exhaustive. Termination and quasi-reductivity together ensure that every ground term reduces to a constructor term. This makes it possible to do an exhaustive case analysis on the rules applicable to an equation and lets us assume that variables are always instantiated by ground constructor terms.

The last property is natural, as inductive theorem proving makes a statement on *ground* terms; there is no point in regarding empty sorts. Together with quasi-reductivity and termination, this implies that all sorts admit ground *constructor* terms.

Methods to prove both quasi-reductivity and termination have previously been published for different styles of constrained rewriting (e.g., see Falke and Kapur [2012] for quasi-reductivity and Falke [2009] and Sakata et al. [2011] for termination. These methods are easily adapted to LCTRSs. Quasi-reductivity is handled in Kop [2017] and is moreover always satisfied by systems obtained from the transformations in Section 3. Some basics of termination analysis for LCTRSs are discussed in Kop [2013].

Example 4.2. As a running example in this section, we will consider $\mathcal{R}_{\mathsf{fact}}$, which combines the factorial function from Example 3.2 with a recursive variant obtained from `int fact(int x) { if (x <= 1) return 1; else return x * fact(x - 1); }`.

$$
\begin{array}{lll}
(1)\ \mathsf{factiter}(x) \to \mathsf{iter}(x, 1, 1) & & (4)\ \mathsf{factrec}(x) \to \mathsf{return}(1) \quad [x \leq 1] \\
(2)\ \mathsf{iter}(x, z, i) \to \mathsf{iter}(x, z * i, i + 1) \quad [i \leq x] & & (5)\ \mathsf{factrec}(x) \to \\
(3)\ \mathsf{iter}(x, z, i) \to \mathsf{return}(z) \quad [i > x] & & \qquad \mathsf{mul}(x, \mathsf{factrec}(x - 1)) \quad [x > 1] \\
 & & (6)\ \mathsf{mul}(x, \mathsf{return}(y)) \to \mathsf{return}(x * y)
\end{array}
$$

(Function symbols were renamed for readability.) We can choose a signature that includes Σ^{core}_{theory}, and each of the sorts—int, bool, result—clearly admits ground terms (e.g., $\mathsf{0}$, false, $\mathsf{return}(\mathsf{0})$). The system was obtained using Section 3 and thus is quasi-reductive. Termination follows because in the recursive rule (2), the value $x - i$ is decreased, while bounded from below by 0, and in the recursion in rule (5), x decreases against the bound 1. This could be proved using, for example, interpretations with support for built-in integers and nontheory symbols [Fuhs et al. 2009], and is automatically handled by our tool Ctrl.

4.2. Rewriting Induction

We now introduce the notions of constrained equations and inductive theorems.

Definition 4.3. A *(constrained) equation* is a triple $s \approx t\ [\varphi]$ with s and t terms and φ a constraint. We write $s \simeq t\ [\varphi]$ to denote either $s \approx t\ [\varphi]$ or $t \approx s\ [\varphi]$. A substitution γ *respects* $s \approx t\ [\varphi]$ if γ respects φ and $Var(s) \cup Var(t) \subseteq Dom(\gamma)$; it is called a *ground constructor substitution* if all $\gamma(x)$ with $x \in Dom(\gamma)$ are ground constructor terms.

An equation $s \approx t\ [\varphi]$ is an *inductive theorem* of an LCTRS $\mathcal{R}$ if $s\gamma \leftrightarrow^*_{\mathcal{R}} t\gamma$ for any ground constructor substitution γ that respects this equation.

Intuitively, if an equation $f(\overrightarrow{x}) \approx g(\overrightarrow{x})\ [\varphi]$ is an inductive theorem, then f and g define the same function (conditional on φ and assuming confluence). As we require termination, we thus consider *total equivalence* in the categorization of Godlin and Strichman [2008]: on all inputs, both programs terminate and return the same values.

To prove that an equation is an inductive theorem, we consider nine inference rules in Sections 4.2.1 through 4.2.9. Four originate in Reddy [1990]; three are based on extensions [Bouhoula 1997; Falke and Kapur 2012; Sakata et al. 2009]; and two are new. All of these rules modify a triple $(\mathcal{E}, \mathcal{H}, b)$, called a *proof state*. Here, $\mathcal{E}$ is a set of equations, $\mathcal{H}$ is a set of rules with $\to_{\mathcal{R}\cup\mathcal{H}}$ terminating, and $b \in \{\text{COMPLETE}, \text{INCOMPLETE}\}$. A rule in $\mathcal{H}$ plays the role of an *induction hypothesis* for "proving" the equations in $\mathcal{E}$ and is called an *induction rule*. The flag b indicates whether we can use the current proof state to *refute* that the initial equation is an inductive theorem; we can do so if $b = \text{COMPLETE}$.

The definition of these rules is used in the following result, proved in Section 4.4.

THEOREM 4.4. *Let an LCTRS with rules $\mathcal{R}$ and signature Σ, satisfying the restrictions from Definition 4.1, be given. Let $\mathcal{E}$ be a finite set of equations and let flag $=$ COMPLETE if we can confirm that $\mathcal{R}$ is confluent and flag $=$ INCOMPLETE otherwise. If $(\mathcal{E}, \emptyset, flag) \vdash_{\mathtt{ri}}^{*} (\emptyset, \mathcal{H}, flag')$ for some $\mathcal{H}, flag'$, then every equation in $\mathcal{E}$ is an inductive theorem of $\mathcal{R}$. If $(\mathcal{E}, \emptyset, flag) \vdash_{\mathtt{ri}}^{*} \bot$, then there is some equation in $\mathcal{E}$ that is not an inductive theorem of $\mathcal{R}$.*

Example 4.5. We will illustrate the various rules by proving that $\mathsf{factrec}$ and $\mathsf{factiter}$ are equivalent on positive input[4] by showing that (FCT.A) is an inductive theorem:

$$\text{(FCT.A)}\quad \mathsf{factrec}(n) \approx \mathsf{factiter}(n)\ [n \geq 1].$$

$\mathcal{R}_{\mathsf{fact}}$ is confluent: as seen in Section 3.8, it is left linear and nonoverlapping, and the right-hand sides do not introduce fresh variables, so confluence is given by Theorem 4 of Kop and Nishida [2013]. Thus, we start with the proof state $(\ \{\ \text{(FCT.A)}\ \},\ \emptyset,\ \text{COMPLETE}\ )$.

Let us now define the nine inference rules to reduce proof states.

4.2.1. SIMPLIFICATION. Our first inference rule originates in Reddy [1990] and can be considered one of the core rules of rewriting induction.

Definition 4.6. If $s \approx t\ [\varphi] \to_{\mathcal{R}\cup\mathcal{H}} u \approx t\ [\psi]$, where $\approx$ is seen as a fresh constructor for the purpose of constrained term reduction,[5] then we may derive

$$(\mathcal{E} \uplus \{(s \simeq t\ [\varphi])\}, \mathcal{H}, b)\ \vdash_{\mathtt{ri}}\ (\mathcal{E} \cup \{(u \approx t\ [\psi])\}, \mathcal{H}, b).$$

This inference rule allows us to reduce one side of an equation. This is altered from Reddy's definition by using constrained rather than normal reduction.

Example 4.7. Following Example 4.5, we observe that $\mathsf{factiter}(n)$ can be reduced by the unconstrained rule (1). Thus, using SIMPLIFICATION, we obtain the proof state:

$$(\{\ \text{(FCT.B)} :\ \mathsf{iter}(n, 1, 1) \approx \mathsf{factrec}(n)\ [n \geq 1]\ \},\ \emptyset,\ \text{COMPLETE}).$$

Here we reduce the right-hand side of the equation (recall that $s \simeq t$ in the rule means that $s \approx t$ or $t \approx s$); the reduced term moves to the left-hand side of the new equation.

[4]We limit interest to positive input for demonstration purposes only: these functions give the same result on all input, but considering only $n \geq 1$ allows us to apply the inference rules in a convenient order.

[5]It does not suffice if $s\ [\varphi] \to_{\mathcal{R}} u\ [\psi]$: when reducing constrained terms, unused variables may be manipulated at will, which causes problems if they are used in t. For example,

$$f(x+0)\ [x > y] \sim f(x+0)\ [z = x+0] \to_{\mathtt{base}} f(z)\ [z = x+0] \sim f(x)\ [x < y],$$

but we should certainly not replace an equation $f(x+0) \approx g(y)\ [x > y]$ by $f(x) \approx g(y)\ [x < y]$.

Next, observe that $\mathsf{iter}(n, 1, 1)$ can be reduced by rule (2) if $n \geq 1$; SIMPLIFICATION then gives

$$(\{\ (\text{FCT.C}) :\ \mathsf{iter}(n, 1 * 1, 1 + 1) \approx \mathsf{factrec}(n)\ [n \geq 1]\ \},\ \emptyset,\ \text{COMPLETE}).$$

Recall that constrained reduction also allows for steps with calculation rules (e.g., see Example 2.16). The added complexity is that we must decide how to handle the fresh variable that these rules introduce. In this article, we use the following strategy:

—if $s \to_{\mathtt{calc}} u$, then $s \approx t\ [\varphi]$ is simplified to $u \approx t\ [\varphi]$—for example, $\mathsf{f}(0 + 1) \approx r\ [\varphi]$ reduces to $\mathsf{f}(1) \approx r\ [\varphi]$;
—a calculation containing variables can be replaced by a fresh variable, which is defined in the (updated) constraint—for example, $\mathsf{f}(x + 1) \approx r\ [\varphi]$ reduces to $\mathsf{f}(y) \approx r\ [\varphi \wedge y = x + 1]$. If such a definition already occurs in the constraint, the relevant variable is used instead—for example, $\mathsf{f}(x+1) \approx r\ [\varphi \wedge y = x+1]$ reduces to $\mathsf{f}(y) \approx r\ [\varphi \wedge y = x+1]$.

Example 4.8. The proof state from Example 4.7 is further simplified to

$$(\{\ (\text{FCT.D}) :\ \mathsf{iter}(n, 1, 2) \approx \mathsf{factrec}(n)\ [n \geq 1]\ \},\ \emptyset,\ \text{COMPLETE}).$$

4.2.2. EXPANSION. Our second core rule also originates from Reddy [1990] but has been more heavily adapted to support irregular rules.

Definition 4.9. Let s, t be terms and φ a constraint, all with variables distinct from those in $\mathcal{R}$ (we can always rename the variables in the rules to support this), and p a position of s. Let $\mathit{Expd}(s \approx t\ [\varphi], p)$ be a set of equations containing, for all rules $\ell \to r\ [\psi] \in \mathcal{R}$ such that ℓ is unifiable with $s_{|p}$ with most general unifier γ, an equation $s' \approx t'\ [\varphi']$, where $s\gamma \approx t\gamma\ [(\varphi\gamma) \wedge (\psi\gamma)] \to_{\mathcal{R}} s' \approx t'\ [\varphi']$ with rule $\ell \to r\ [\psi]$ at position $1 \cdot p$. Here, as in SIMPLIFICATION, $\approx$ is seen as a fresh constructor for the reduction. If $s_{|p}$ is *basic* (i.e., $s_{|p} = f(s_1, \ldots, s_n)$ with $f \in \mathcal{D}$ and all s_i constructor terms), we may derive

$$(\mathcal{E} \uplus \{s \simeq t\ [\varphi]\}, \mathcal{H}, b)\ \vdash_{\mathtt{ri}}\ (\mathcal{E} \cup \mathit{Expd}(s \approx t\ [\varphi], p), \mathcal{H}, b).$$

If, moreover, $\mathcal{R} \cup \mathcal{H} \cup \{s \to t\ [\varphi]\}$ is terminating, we may even derive

$$(\mathcal{E} \uplus \{s \simeq t\ [\varphi]\}, \mathcal{H}, b)\ \vdash_{\mathtt{ri}}\ (\mathcal{E} \cup \mathit{Expd}(s \approx t\ [\varphi], p), \mathcal{H} \cup \{s \to t\ [\varphi]\}, b).$$

Intuitively, this inference rule uses narrowing for a *case analysis*: *Expd* generates all resulting equations if a ground constructor instance of $s \approx t\ [\varphi]$ is reduced at position p of s. In addition, we save the current equation as a rule to take an induction step.

Example 4.10. Following Example 4.8, we consider which rules may apply to an instance of $\mathsf{factrec}(n)$ with $n \geq 1$. For $\mathit{Expd}(\mathsf{factrec}(n) \approx \mathsf{iter}(n, 1, 2)\ [n \geq 1], \epsilon)$, we choose

$$\left\{ \begin{array}{ll} (\text{FCT.E}): & \mathsf{return}(1) \approx \mathsf{iter}(n, 1, 2)\ [n \geq 1 \wedge n \leq 1], \\ (\text{FCT.F}): & \mathsf{mul}(n, \mathsf{factrec}(n-1)) \approx \mathsf{iter}(n, 1, 2)\ [n \geq 1 \wedge n > 1] \end{array} \right\}.$$

In both cases, we used the unifier $\gamma = [x := n]$. If we write (FCT.D^{-1}) for the rule generated from the inverse of (FCT.D) (so the rule $\mathsf{factrec}(n) \to \mathsf{iter}(n, 1, 2)\ [n \geq 1]$), then $\mathcal{R} \cup \{(\text{FCT.D}^{-1})\}$ is terminating as the new rule does not cause mutual recursion between iter and factrec. We continue with $(\ \{(\text{FCT.E}), (\text{FCT.F})\},\ \{(\text{FCT.D}^{-1})\},\ \text{COMPLETE}\ )$. Now we can show the second kind of calculation step, using SIMPLIFICATION on (FCT.F), which gives

$$\left(\left\{ \begin{array}{ll} (\text{FCT.E}): & \mathsf{return}(1) \approx \mathsf{iter}(n, 1, 2)\ [n \geq 1 \wedge n \leq 1], \\ (\text{FCT.G}): & \mathsf{mul}(n, \mathsf{factrec}(m)) \approx \mathsf{iter}(n, 1, 2)\ [n > 1 \wedge m = n - 1] \end{array} \right\},\ \begin{array}{c} \{(\text{FCT.D}^{-1})\}, \\ \text{COMPLETE} \end{array} \right).$$

Here, we also removed the redundant clause $n \geq 1$, which is allowed by definition of $\to_{\mathcal{R}}$ on constrained terms. As $n \geq 1 \wedge n \leq 1$ implies that $n = 1$, we may use SIMPLIFICATION

with rule (3) on (FCT.E), and with rule (2) followed by calculations on (FCT.G), to get

$$\left(\left\{\begin{array}{lll}\text{(FCT.H):} & \mathsf{return}(1) \approx \mathsf{return}(1) & [n=1],\\ \text{(FCT.I):} & \mathsf{iter}(n,2,3) \approx \mathsf{mul}(n,\mathsf{factrec}(m)) & [n>1\wedge m=n-1]\end{array}\right\},\ \begin{array}{c}\{(\text{FCT.D}^{-1})\},\\ \textsc{complete}\end{array}\right).$$

Now we can use "induction": we eliminate the occurrence of factrec with a SIMPLIFICATION step using the induction rule (FCT.D^{-1}) and substitution $[n := m]$. This gives

$$\left(\left\{\begin{array}{lll}\text{(FCT.H):} & \mathsf{return}(1) \approx \mathsf{return}(1) & [n=1],\\ \text{(FCT.J):} & \mathsf{mul}(n,\mathsf{iter}(m,1,2)) \approx \mathsf{iter}(n,2,3) & [n>1\wedge m=n-1]\end{array}\right\},\ \begin{array}{c}\{(\text{FCT.D}^{-1})\},\\ \textsc{complete}\end{array}\right).$$

Note that the choice of *Expd* is nondeterministic, as it uses reduction of constrained terms. The most natural choice for $\mathit{Expd}(s \approx t\ [\varphi],\ p)$—which we use in examples—is

$$\{\ s[r]_p\gamma \approx t\gamma\ [(\varphi\gamma)\wedge(\psi\gamma)]\ \mid\ \ell \to r\ [\psi] \in \mathcal{R},\ s|_p \text{ unifies with } \ell \text{ with mgu } \gamma\ \}.$$

However, for *irregular* rules in particular, it may be strategic to choose a different set. Consider, for example, a (nonconfluent) LCTRS with rules $f(x) \to g(y)\ [x > 0 \wedge x > y]$ and $f(x) \to g(y)\ [x \le 0 \wedge x \le y]$. With the choice for $\mathit{Expd}(s \approx t\ [\varphi],\ p)$ above, an equation $f(x) \approx g(0)$ [true] results in $\{\ g(y) \approx g(0)\ [x > 0 \wedge x > y],\ g(y) \approx g(0)\ [x \le 0 \wedge x \le y]\ \}$. If g is a constructor, neither of these equations can be handled. Using the full definition of EXPANSION, we can choose $g(0) \approx g(0)$ [true] for both equations.

Also note that there is no choice in the orientation of the rule added to $\mathcal{H}$: this is determined by the side of the equation on which the expansion was applied. Thus, in Example 4.10, we were not allowed to add (FCT.D) instead of (FCT.D^{-1}).

Our definition of EXPANSION differs from both its original and existing work on constrained rewriting induction. To start, those works define $\mathit{Expd}(s \approx t\ [\varphi],\ p)$ simply as the "natural choice" that we suggested. Second, we included a case where no rule is added to allow for progress when adding the rule might cause nontermination. Forms of this case appear as a separate rule in other work, such as CASE ANALYSIS in Bouhoula [1997] and REWRITE/PARTIAL SPLITTING in Bouhoula and Jacquemard [2008a, 2008b]. A weaker form with constraints is given in Falke and Kapur [2012] (CASE-SIMPLIFY).

4.2.3. DELETION. The last of the core rules serves to remove solved equations from $\mathcal{E}$.

Definition 4.11. If $s = t$ or φ is not satisfiable, we can delete $s \approx t\ [\varphi]$ from $\mathcal{E}$:

$$(\mathcal{E} \uplus \{s \approx t\ [\varphi]\}, \mathcal{H}, b)\ \vdash_{\mathtt{ri}}\ (\mathcal{E}, \mathcal{H}, b).$$

Compared to the corresponding rule in Reddy [1990], the unsatisfiability case is new; it is similar to the corresponding rules in Sakata et al. [2009] and Falke and Kapur [2012].

Example 4.12. Following Example 4.10, the left- and right-hand sides of (FCT.H) are the same, so we may remove the equation with DELETION, obtaining ({ (FCT.J) }, { (FCT.D^{-1}) }, COMPLETE). We will see the other form of DELETION later in Example 4.18.

4.2.4. POSTULATE. Sometimes it is useful to make the problem seemingly harder. To this end, we consider the last inference rule from Reddy [1990].

Definition 4.13. For any set of equations $\mathcal{E}'$, we can derive

$$(\mathcal{E}, \mathcal{H}, b)\ \vdash_{\mathtt{ri}}\ (\mathcal{E} \cup \mathcal{E}', \mathcal{H}, \textsc{incomplete}).$$

The POSTULATE rule allows us to add additional equations to $\mathcal{E}$ (although at a price: we cannot conclude nonequivalence after adding a potentially unsound equation). The reason to do so is that in proving the equations in $\mathcal{E}'$ to be inductive theorems, we may derive new induction rules. These can then be used to simplify the elements of $\mathcal{E}$.

Example 4.14. Following Example 4.12, EXPANSION followed by SIMPLIFICATION gives

$$\text{(FCT.K)}:\ \mathsf{mul}(n, \mathsf{iter}(m, 2, 3)) \approx \mathsf{iter}(n, 6, 4)\ \ [n \geq 3 \wedge m = n - 1].$$

But now a pattern starts to arise. Expanding and fully simplifying again, we obtain

$$\text{(FCT.L)}:\ \mathsf{mul}(n, \mathsf{iter}(m, 6, 4)) \approx \mathsf{iter}(n, 24, 5)\ \ [n \geq 4 \wedge m = n - 1].$$

And so on. Here, (FCT.K) cannot be handled by the induction rule (FCT.J^{-1}), nor can (FCT.L) be handled by (FCT.K^{-1}). We have a *divergence*: a sequence of increasingly complex equations, each generated from the same leg in an EXPANSION (see also the *divergence critic* in Walsh [1996]). Yet the previous induction rules never apply to the new equation. This suggests that we need a lemma equation. We use POSTULATE to get

$$\left(\left\{ \begin{array}{ll} \text{(FCT.J)}: & \mathsf{mul}(n, \mathsf{iter}(m, 1, 2)) \approx \mathsf{iter}(n, 2, 3) \\ & \quad [n > 1 \wedge m = n - 1] \\ \text{(FCT.M)}: & \mathsf{mul}(n, \mathsf{iter}(m, x, y)) \approx \mathsf{iter}(n, x', y') \\ & \quad [n \geq y \wedge m = n - 1 \wedge y' = y + 1 \wedge x' = x * y] \end{array} \right\},\ \begin{array}{c} \{(\text{FCT.D}^{-1})\}, \\ \text{INCOMPLETE} \end{array} \right).$$

Using EXPANSION on the right-hand of (FCT.M), we have

$$\left(\left\{ \begin{array}{ll} \text{(FCT.J)}: & \mathsf{mul}(n, \mathsf{iter}(m, 1, 2)) \approx \mathsf{iter}(n, 2, 3) \\ & \quad [n > 1 \wedge m = n - 1] \\ \text{(FCT.N)}: & \mathsf{iter}(n, x' * y', y' + 1) \approx \mathsf{mul}(n, \mathsf{iter}(m, x, y)) \\ & \quad [n \geq y \wedge m = n - 1 \wedge y' = y + 1 \wedge x' = x * y \wedge y' \leq n] \\ \text{(FCT.O)}: & \mathsf{return}(x') \approx \mathsf{mul}(n, \mathsf{iter}(m, x, y)) \\ & \quad [n \geq y \wedge m = n - 1 \wedge y' = y + 1 \wedge x' = x * y \wedge y' > n] \end{array} \right\},\ \begin{array}{c} \left\{ \begin{array}{l} (\text{FCT.D}^{-1}) \\ (\text{FCT.M}^{-1}) \end{array} \right\}, \\ \text{INCOMPLETE} \end{array} \right).$$

But now we have added (FCT.M^{-1}) as an induction rule. As a result—since $n > 1$ clearly implies that $n \geq 2$—we can use SIMPLIFICATION with a substitution $[n := n, x := 1, y := 2, x' := 2, y' := 3]$ to reduce (FCT.J) to the equation $\mathsf{mul}(n, \mathsf{iter}(m, 1, 2)) \approx \mathsf{mul}(n, \mathsf{iter}(m, 1, 2))$ [...], which we may immediately remove by DELETION. We continue with the proof state ({ (FCT.N), (FCT.O) }, { (FCT.D^{-1}), (FCT.M^{-1}) }, INCOMPLETE).

Although the need to choose arbitrary new equations for use in POSTULATE may seem somewhat problematic, this is actually a key step. Complex theorems typically require more than straight induction, both in our setting and in mathematical proofs in general. Thus, generation of suitable *lemma equations* $\mathcal{E}'$ is not only part, but even at the heart, of inductive theorem proving. Hence, this subject has been extensively investigated [Bundy et al. 2005; Kapur and Sakhanenko 2003; Kapur and Subramaniam 1996; Nakabayashi et al. 2010; Urso and Kounalis 2004; Walsh 1996], and a large variety of lemma generation techniques exist, at least in the setting without constraints.

4.2.5. GENERALIZATION. A very typical use of POSTULATE is to *generalize* a problematic equation. For simplicity, we add a shortcut to do this in one step.

Definition 4.15. If for all substitutions γ that respect φ there is a substitution δ that respects ψ with $s\gamma = s'\delta$ and $t\gamma = t'\delta$, then we can derive

$$(\mathcal{E} \uplus \{s \approx t\ [\varphi]\}, \mathcal{H}, b) \vdash_{\mathtt{ri}} (\mathcal{E} \cup \{s' \approx t'\ [\psi]\}, \mathcal{H}, \text{INCOMPLETE}).$$

This inference rule is rarely necessary: we could usually add $s' \approx t'\ [\psi]$ using POSTULATE and use the resulting induction rules to eliminate $s \approx t\ [\varphi]$, as we did in Example 4.14. By generalizing instead, we avoid extra steps, and intuitively we strengthen an induction statement rather than add a separate lemma. Without constraints, GENERALIZATION can be seen as a combination of POSTULATE and the SUBSUMPTION rule in Bouhoula [1997]. As there are several results for generalizing equations in the literature [Bundy

et al. 1993, 2005; Basin and Walsh 1992; Walsh 1996; Urso and Kounalis 2004], the combination is useful beyond just this article.

Example 4.16. In Example 4.14, we could have used GENERALIZATION immediately to move from the proof state ({ (FCT.J) }, { (FCT.D^{-1}) }, INCOMPLETE) to ({ (FCT.M) }, { (FCT.D^{-1}) }, INCOMPLETE).

4.2.6. EQ-DELETION. The following rule, which was adapted from Sakata et al. [2009], provides a link between the equation part $s \approx t$ and the constraint.

Definition 4.17. Let C be an arbitrary context with n holes (C may contain symbols in Σ_{theory}). If all $s_i, t_i \in \mathcal{T}erms(\Sigma_{theory}, Var(\varphi))$, then we can derive

$$(\mathcal{E} \uplus \{C[s_1, \ldots, s_n] \simeq C[t_1, \ldots, t_n]\ [\varphi]\}, \mathcal{H}, b) \vdash_{\mathtt{ri}} (\mathcal{E} \cup \{C[s_1, \ldots, s_n] \approx C[t_1, \ldots, t_n]\ [\varphi \wedge \neg(\textstyle\bigwedge_{i=1}^{n} s_i = t_i)]\}, \mathcal{H}, b).$$

Intuitively, if $\bigwedge_{i=1}^{n} s_i = t_i$ holds, then $C[s_1, \ldots, s_n]\gamma \leftrightarrow^*_{\mathcal{R}_{\mathtt{calc}}} C[t_1, \ldots, t_n]\gamma$, so we are done. EQ-DELETION excludes this case from the equation. In combination with DELETION, this rule gives a more general variation of THEORY$_\top$ in Falke and Kapur [2012].

Example 4.18. Continuing from Example 4.14 (or Example 4.16), we observe that $n \geq y$, $y' = y + 1$ and $y' > n$ together imply $n = y$, and with $m = n - 1$ we thus have $y > m$ as well. Therefore, SIMPLIFICATION on (FCT.O) by rule (3) followed by (6) gives

$$\text{(FCT.P)}: \quad \mathsf{return}(n * x) \approx \mathsf{return}(x')\ [n = y \wedge m = n - 1 \wedge y' = y + 1 \wedge x' = x * y].$$

We can use EQ-DELETION with the context $C[\Box] = \mathsf{return}(\Box)$ to replace (FCT.P) by

$$\text{(FCT.Q)}: \ \mathsf{return}(n*x) \approx \mathsf{return}(x')\ [n = y \wedge m = n-1 \wedge y' = y+1 \wedge x' = x*y \wedge \neg(n*x = x')]$$

As $n = y$ and $x' = x * y$ together imply that $n * x = x'$, the constraint of this equation is not satisfiable. We may remove it using DELETION, giving the proof state ({ (FCT.N) }, { (FCT.D^{-1}), (FCT.M^{-1}) }, INCOMPLETE).

EQ-DELETION is among the core rules for constrained rewriting induction: almost all inductive proofs use it, in contrast to the remaining three inference rules.

Example 4.19. To complete our example, consider (FCT.N). As $y + 1 = y' \leq n \wedge m = n - 1$ implies that $y \leq m$, we may apply SIMPLIFICATION with rule (2) to replace it by

$$\text{(FCT.R)}: \ \mathsf{mul}(n, \mathsf{iter}(m, x * y, y + 1)) \approx \mathsf{iter}(n, x' * y', y' + 1)\ [n \geq y \wedge m = n - 1 \wedge y' = y + 1 \wedge x' = x * y \wedge y' \leq n]\ .$$

Then, using SIMPLIFICATION with calculations (and observing that both $x * y$ and $y + 1$ are "defined" in the constraint, as discussed in Section 4.2.1), we get

$$\text{(FCT.S)}: \quad \mathsf{mul}(n, \mathsf{iter}(m, x', y')) \approx \mathsf{iter}(n, x'', y'')\ [n \geq y' \wedge m = n - 1 \wedge x' = x * y \wedge x'' = x' * y' \wedge y'' = y' + 1]\ .$$

(We removed the clauses with y from the constraint, as y does not occur in the equation part.) But now the induction rule (FCT.M^{-1}) applies! As this rule is irregular, we must be careful. We use the substitution $\gamma = [n := n, m := m, x'' := x', y' := y'', x := x', y := y']$, which also affects variables not occurring in the left-hand side. The substituted constraint for the rule is $n \geq y' \wedge m = n - 1 \wedge y'' = y' + 1 \wedge x'' = x' * y'$, which is indeed implied by the constraint of (FCT.S). Using SIMPLIFICATION, we thus obtain

$$\left(\left\{ \begin{array}{l} \text{(FCT.T)}: \quad \mathsf{mul}(n, \mathsf{iter}(m, x', y')) \approx \mathsf{mul}(n, \mathsf{iter}(m, x', y')) \\ \quad [n \geq y' \wedge m = n - 1 \wedge x' = x * y \wedge x'' = x' * y' \wedge y'' = y' + 1] \end{array} \right\}, \ \begin{array}{c} \{\cdots\}, \\ \text{INCOMPLETE} \end{array} \right).$$

As the left- and right-hand sides of the remaining equation are the same, we may remove it using DELETION. This leaves a proof state of the form $(\emptyset, \mathcal{H}, \text{INCOMPLETE})$, so by Theorem 4.4, the equation $\mathsf{factrec}(n) \approx \mathsf{factiter}(n)$ $[n \geq 1]$ is an inductive theorem.

4.2.7. CONSTRUCTOR. Where Falke and Kapur [2012] and Sakata et al. [2009] focus on systems with only theory symbols and defined symbols, here we are also interested in nontheory constructors, such as error_f and return_f. To support this, we add the following inference rule.

Definition 4.20. If f is a constructor, we can derive

$$(\mathcal{E} \uplus \{f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)[\varphi]\}, \mathcal{H}, b) \vdash_{\mathtt{ri}} (\mathcal{E} \cup \{s_i \approx t_i\ [\varphi] \mid 1 \leq i \leq n\}, \mathcal{H}, b).$$

The CONSTRUCTOR rule originates in Bouhoula [1997], where it is called POSITIVE DECOMPOSITION, although variations occur in earlier work on implicit induction, such as Huet and Hullot [1982]. It is used to split up a large equation into smaller problems. This inference rule is particularly useful in applications where a recursive structure, such as a list, is inductively built up but will also be invaluable as part of a disproof.

Example 4.21. Suppose that in Example 4.5 we had started with (BAD.A): $\mathsf{factiter}(x) \approx \mathsf{factrec}(x - 1)$ [true]. Following some expansions and simplifications, we arrive at

$$\left(\left\{ \begin{array}{ll} \text{(BAD.B):} & \mathsf{return}(2) \approx \mathsf{return}(1) \quad [x = 2] \\ \text{(BAD.C):} & \mathsf{iter}(x, 1, 1) \approx \mathsf{factrec}(y) \quad [y = x - 1 \wedge y > 1] \end{array} \right\}, \mathcal{H}, \text{COMPLETE} \right)$$

(for some $\mathcal{H}$). We can use CONSTRUCTOR to replace (BAD.B) by (BAD.D): $2 \approx 1$ $[x = 2]$.

4.2.8. DISPROVE. Recall that to show that an equation is not an inductive theorem, we must derive $\bot$ from a COMPLETE proof state. For this, we use DISPROVE.

Definition 4.22. Suppose that $\vdash s : \iota$ and one of the following holds:

—$s, t \in \mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$, ι is a theory sort, and $\varphi \wedge s \neq t$ is satisfiable;
—$s = f(\overrightarrow{s})$ and $t = g(\overrightarrow{t})$ with f, g distinct constructors and φ satisfiable;
—$s \in \mathcal{V} \setminus \mathcal{V}ar(\varphi)$, φ is satisfiable, at least two different constructors have output sort ι, and either t is a variable distinct from s or t has the form $g(\overrightarrow{t})$ with $g \in \mathcal{C}ons$.

Then we may derive the following.

$$(\mathcal{E} \uplus \{s \simeq t\ [\varphi]\}, \mathcal{H}, \text{COMPLETE}) \vdash_{\mathtt{ri}} \bot$$

The first case of this rule corresponds to $\text{THEORY}_\top$ in Falke and Kapur [2012] and Theorem 7.2 in Sakata et al. [2009]; note that the restriction to theory sorts only excludes the case where s and t are nonlogical variables. The second case corresponds to POSITIVE CLASH in Bouhoula [1997]. The third case is new in rewriting induction but appears in Huet and Hullot [1982], an implicit induction method based on completion.

Example 4.23. Following Example 4.21, we observe that $x = 2 \wedge 2 \neq 1$ is satisfiable. Thus, by DISPROVE, we reduce $(\{\text{(BAD.D), (BAD.C)}\}, \mathcal{H}, \text{COMPLETE})$ to $\bot$. By confluence of $\mathcal{R}_{\mathsf{fact}}$, we see that $\mathsf{factiter}(x)$ and $\mathsf{factrec}(x - 1)$ have different normal forms for some x.

4.2.9. COMPLETENESS. A downside of POSTULATE and GENERALIZATION is the potential loss of the completeness flag. To weaken this problem—and empower automatic tools to combine the search for a proof and a disproof—we add our final inference rule.

Definition 4.24. For any set of equations $\mathcal{E}$ and $\mathcal{E}' \subseteq \mathcal{E}$, we can derive the following:

$$\begin{array}{c} \text{If } (\mathcal{E}, \mathcal{H}, \text{COMPLETE}) \vdash^*_{\mathtt{ri}} (\mathcal{E}', \mathcal{H}', \text{INCOMPLETE}), \\ \text{then } (\mathcal{E}, \mathcal{H}, \text{COMPLETE}) \vdash_{\mathtt{ri}} (\mathcal{E}', \mathcal{H}', \text{COMPLETE}). \end{array}$$

Essentially, COMPLETENESS allows us to return the completeness flag that was lost due to a POSTULATE or GENERALIZATION step, once we have managed to remove all the added/generalized lemma equations. In practice, a tool or human prover might have a derivation that could be denoted $(\mathcal{E}, \mathcal{H}, \text{COMPLETE}) \vdash_{\mathtt{ri}\ (\text{POSTULATE})} (\mathcal{E} \cup \mathcal{E}', \mathcal{H}, \text{INCOMPLETE}) \vdash_{\mathtt{ri}} \cdots \vdash_{\mathtt{ri}} (\mathcal{E}, \mathcal{H} \cup \mathcal{H}', \text{INCOMPLETE}) \vdash_{\mathtt{ri}\ (\text{COMPLETENESS})} (\mathcal{E}, \mathcal{H} \cup \mathcal{H}', \text{COMPLETE})$ by remembering the set $\mathcal{E}$ where the completeness flag was lost.

Example 4.25. Recall Example 4.14. Starting in $(\{\,(\text{FCT.J})\,\},\ \{\,(\text{FCT.D}^{-1})\,\},\ \text{COMPLETE})$, we lost completeness by adding a lemma equation. Then, after using EXPANSION, we arrived at $(\{\,(\text{FCT.J}), (\text{FCT.N}), (\text{FCT.O})\,\},\ \{\,(\text{FCT.D}^{-1}), (\text{FCT.M}^{-1})\,\},\ \text{INCOMPLETE})$. Applying the proof steps of Examples 4.18 and 4.19 without touching (FCT.J), we could reduce this state to $(\{\,(\text{FCT.J})\,\},\ \{\,(\text{FCT.D}^{-1}), (\text{FCT.M}^{-1})\,\},\ \text{INCOMPLETE})$. But the only equation (FCT.J) in this set is the one we started with. Thus, we may restore the completeness flag, resulting in $(\{\,(\text{FCT.J})\,\},\ \{\,(\text{FCT.D}^{-1}), (\text{FCT.M}^{-1})\,\},\ \text{COMPLETE})$.

There are many other potential inference rules that we could consider, as various extensions of the base method have been studied in the literature (e.g., see Bouhoula [1997]). For now, we stick to these nine rules and leave the remainder to future work.

4.3. Examples

The running example in Section 4.2 gives a good general idea of the power of the method and the way that it is applied. In this section, we present some further examples. For brevity, we only list the equations $\mathcal{E}$ in each step, not the completeness flag or induction rules $\mathcal{H}$. Unless stated otherwise, these induction rules are not applicable to new equations.

Example 4.26. Let us look at an assignment to implement `strlen`, a string function that operates on 0-terminated `char` arrays. As `char` is a numeric data type, we use integer arrays in the LCTRS translation (although another underlying sort $\mathcal{I}_{\mathsf{char}}$ would make little difference). The example function and its LCTRS translation are as follows.

```
int strlen(char *s){
  for(int i = 0;;i++){
    if(s[i] == 0)
      return i;
  }
}
```

$$
\begin{array}{lll}
(1) & \mathsf{strlen}(x) \to \mathsf{u}(x, 0) & \\
(2) & \mathsf{u}(x, i) \to \mathsf{error} & [i < 0 \vee i \geq \mathsf{size}(x)] \\
(3) & \mathsf{u}(x, i) \to \mathsf{return}(i) & [0 \leq i < \mathsf{size}(x) \wedge \mathsf{select}(x, i) = 0] \\
(4) & \mathsf{u}(x, i) \to \mathsf{u}(x, i+1) & [0 \leq i < \mathsf{size}(x) \wedge \mathsf{select}(x, i) \neq 0]
\end{array}
$$

Note that the bounds checks guarantee termination. To see that strlen does what we would expect it to do, we want to know that for *valid C strings*, $\mathsf{strlen}(a)$ returns the first integer i such that $a[i] = 0$. Following Section 1.2.4, this corresponds to the following equation:

$$
\begin{array}{ll}
(\text{LEN.A}) & \mathsf{strlen}(x) \approx \mathsf{return}(n) \\
& [0 \leq n < \mathsf{size}(x) \wedge \forall i \in \{0, \ldots, n-1\}(\mathsf{select}(x, i) \neq 0) \wedge \mathsf{select}(x, n) = 0].
\end{array}
$$

Here we use bounded quantification, which, as described in Section 2.2, can be seen as syntactic sugar for an additional predicate; the underlying LCTRS could, for example, use a symbol nonzero and replace $\forall i \in \{0, \ldots, n-1\}(\mathsf{select}(x, i) \neq 0)$ by $\mathsf{nonzero}(x, n)$ in the constraint.

We first use SIMPLIFICATION with rule (1), which gives (LEN.B):

$$\mathsf{u}(x, 0) \approx \mathsf{return}(n)\ [0 \leq n < \mathsf{size}(x) \wedge \forall i \in \{0, \ldots, n-1\}(\mathsf{select}(x, i) \neq 0) \wedge \mathsf{select}(x, n) = 0].$$

We continue with EXPANSION, again on the left-hand side. Since the constraint implies that $0 < \mathsf{size}(x)$, the error case (2) is unsatisfiable, so we delete it, which leaves

$$\begin{array}{ll}(\text{LEN.C}) & \mathsf{return}(0) \approx \mathsf{return}(n) \quad [0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0,\dots,n-1\}(\mathsf{select}(x,i) \neq 0) \wedge \\ & \qquad \mathsf{select}(x,n) = 0 \wedge 0 \le 0 < \mathsf{size}(x) \wedge \mathsf{select}(x,0) = 0] \\ (\text{LEN.D}) & \mathsf{u}(x, 0+1) \approx \mathsf{return}(n) \quad [0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0,\dots,n-1\}(\mathsf{select}(x,i) \neq 0) \wedge \\ & \qquad \mathsf{select}(x,n) = 0 \wedge 0 \le 0 < \mathsf{size}(x) \wedge \mathsf{select}(x,0) \neq 0].\end{array}$$

As the constraint of (LEN.C) implies that $n = 0$, we can remove (LEN.C) using EQ-DELETION and DELETION. (LEN.D) is simplified with a calculation:

$$\begin{array}{ll}(\text{LEN.E}) & \mathsf{u}(x,1) \approx \mathsf{return}(n) \quad [0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0,\dots,n-1\}(\mathsf{select}(x,i) \neq 0) \wedge \\ & \qquad \mathsf{select}(x,n) = 0 \wedge 0 < \mathsf{size}(x) \wedge \mathsf{select}(x,0) \neq 0],\end{array}$$

which we expand again (once more skipping the error case due to unsatisfiability):

$$\begin{array}{ll}(\text{LEN.F}) & \mathsf{return}(1) \approx \mathsf{return}(n) \quad [0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0,\dots,n-1\}(\mathsf{select}(x,i) \neq 0) \wedge \\ & \qquad \mathsf{select}(x,n) = 0 \wedge 0 < \mathsf{size}(x) \wedge \mathsf{select}(x,0) \neq 0 \wedge 0 \le 1 < \mathsf{size}(x) \wedge \mathsf{select}(x,1) = 0] \\ (\text{LEN.G}) & \mathsf{u}(x, 1+1) \approx \mathsf{return}(n) \quad [0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0,\dots,n-1\}(\mathsf{select}(x,i) \neq 0) \wedge \\ & \qquad \mathsf{select}(x,n) = 0 \wedge 0 < \mathsf{size}(x) \wedge \mathsf{select}(x,0) \neq 0 \wedge 0 \le 1 < \mathsf{size}(x) \wedge \mathsf{select}(x,1) \neq 0].\end{array}$$

The constraint of (LEN.F) implies that $n = 1$, so we easily remove this equation. (LEN.G) is simplified using a calculation and then expanded again:

$$\begin{array}{ll}(\text{LEN.H}) & \mathsf{return}(2) \approx \mathsf{return}(n) \quad [\cdots \wedge 2 < \mathsf{size}(x) \wedge \mathsf{select}(x,2) = 0] \\ (\text{LEN.I}) & \mathsf{u}(x, 2+1) \approx \mathsf{return}(n) \quad [0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0,\dots,n-1\}(\mathsf{select}(x,i) \neq 0) \\ & \qquad \wedge\, \mathsf{select}(x,n) = 0 \wedge 0 < \mathsf{size}(x) \wedge \mathsf{select}(x,0) \neq 0 \wedge 1 < \\ & \qquad \mathsf{size}(x) \wedge \mathsf{select}(x,1) \neq 0 \wedge 2 < \mathsf{size}(x) \wedge \mathsf{select}(x,2) \neq 0].\end{array}$$

We drop (LEN.H) easily. Simplifying (LEN.I) and reformulating its constraint gives

$$\begin{array}{ll}(\text{LEN.J}) & \mathsf{u}(x,3) \approx \mathsf{return}(n) \quad [0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0,\dots,n-1\}(\mathsf{select}(x,i) \neq 0) \wedge \\ & \qquad \mathsf{select}(x,n) = 0 \wedge 0 \le 2 < \mathsf{size}(x) \wedge \forall j \in \{0,\dots,2\}(\mathsf{select}(x,j) \neq 0)].\end{array}$$

Note that we grouped together the $\neq 0$ statements into a quantification, which looks a lot like the other quantification in the constraint. Now let us generalize! We will use the generalized equation (LEN.K): $\mathsf{u}(x,k) \approx \mathsf{return}(n)$ $[\varphi]$, where

$$\begin{array}{ll}\varphi: & k = m+1 \wedge 0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0,\dots,n-1\}(\mathsf{select}(x,i) \neq 0) \wedge \\ & \mathsf{select}(x,n) = 0 \wedge 0 \le m < \mathsf{size}(x) \wedge \forall j \in \{0,\dots,m\}(\mathsf{select}(x,j) \neq 0).\end{array}$$

Obviously, (LEN.J) is an instance of (LEN.K); we use EXPANSION to obtain

$$\begin{array}{lll}(\text{LEN.L}) & \mathsf{error} \approx \mathsf{return}(n) & [\varphi \wedge (k < 0 \vee k \ge \mathsf{size}(x))] \\ (\text{LEN.M}) & \mathsf{return}(k) \approx \mathsf{return}(n) & [\varphi \wedge 0 \le k < \mathsf{size}(x) \wedge \mathsf{select}(x,k) = 0] \\ (\text{LEN.N}) & \mathsf{u}(x,k+1) \approx \mathsf{return}(n) & [\varphi \wedge 0 \le k < \mathsf{size}(x) \wedge \mathsf{select}(x,k) \neq 0].\end{array}$$

The two $\forall$ statements in φ, together with $\mathsf{select}(x,n) = 0$, imply that $m < n$, so $k \le n$. Consequently, (LEN.L) has an unsatisfiable constraint and may be deleted: $k < 0$ cannot hold because $k = m+1$ and $0 \le m$, nor $k \ge \mathsf{size}(x)$ because $k \le n$ and $n < \mathsf{size}(x)$.

For (LEN.M), the two $\forall$ statements together with $\mathsf{select}(x,k) = 0$ imply that $n - 1 < k$, so $n \le k$. Thus, $n = k$. EQ-DELETION gives an equation with an unsatisfiable constraint, which we remove using DELETION. As for (LEN.N), we use SIMPLIFICATION with a calculation and reformulate the constraint to obtain

$$\begin{array}{ll}(\text{LEN.O}) & \mathsf{u}(x,p) \approx \mathsf{return}(n) \quad [p = k+1 \wedge \mathsf{select}(x,n) = 0 \wedge 0 \le n < \mathsf{size}(x) \wedge \\ & \qquad \forall i \in \{0,\dots,n-1\}(\mathsf{select}(x,i) \neq 0) \wedge 0 \le k < \mathsf{size}(x) \wedge \\ & \qquad \forall j \in \{0,\dots,k\}(\mathsf{select}(x,j) \neq 0) \wedge \text{some constraints on } m].\end{array}$$

This equation is simplified to an equation of the form $\mathsf{return}(n) \approx \mathsf{return}(n)$ $[\dots]$ using the induction rule obtained from (LEN.K); we complete with DELETION.

Example 4.27. We consider $\mathcal{R}_{\mathsf{sum}}$, the LCTRS with the two correct implementations of the motivating Example 1.1—that is, rules (1a) through (1d) and (4a) through (4e).

The rules are terminating because in the recursive rule (1c), $n - i$ decreases in every step and is bounded from below by 0, and in rule (4c), the value k decreases against the bound 0.

To prove equivalence of these implementations when the given length is within the array bounds, we must show that (ARR.A) is an inductive theorem:

$$\text{(ARR.A)}\ \ \mathsf{sum1}(a,k) \approx \mathsf{sum4}(a,k)\ [0 \le k \le \mathsf{size}(a)].$$

The derivation follows a similar pattern as with factorial: we first simplify the left-hand side using rule (1a), then expand on the right and use the induction rule, $\mathsf{sum4}(a,k) \to \mathsf{u}(a,k,0,0)\ [0 \le k \le \mathsf{size}(a)]$, to eliminate the remaining occurrence of sum4. This gives

$$\begin{aligned}\mathsf{w}(n,\mathsf{u}(a,k',0,0)) &\approx \mathsf{u}(a,k,r,1)\\ [k' = k-1 \wedge 0 \le k' < \mathsf{size}(a) &\wedge n = \mathsf{select}(a,k') \wedge r = 0 + \mathsf{select}(a,0)].\end{aligned}$$

Continuing to expand and simplify, we easily remove the equations resulting from rules (1b) and (1d) in every step, but the recursive rule (1c) causes a divergence.

$$\begin{aligned}&\mathsf{u}(a,k,r_2,3) \approx \mathsf{w}(n,\mathsf{u}(a,k',r_1,2))\ \ [k'=k-1 \wedge 2 < k \le \mathsf{size}(a) \wedge r_2 = r_1 + \mathsf{select}(a,1) \wedge \ldots]\\ &\mathsf{u}(a,k,r_3,4) \approx \mathsf{w}(n,\mathsf{u}(a,k',r_2,3))\ \ [k'=k-1 \wedge 3 < k \le \mathsf{size}(a) \wedge r_3 = r_2 + \mathsf{select}(a,2) \wedge \ldots]\\ &\mathsf{u}(a,k,r_4,5) \approx \mathsf{w}(n,\mathsf{u}(a,k',r_3,4))\ \ [k'=k-1 \wedge 4 < k \le \mathsf{size}(a) \wedge r_4 = r_3 + \mathsf{select}(a,3) \wedge \ldots]\end{aligned}$$

We can easily complete after generalizing any of these equations to

$$\begin{aligned}\text{(ARR.GEN)}:\ \mathsf{u}(a,k,r,i) \approx \mathsf{w}(n,\mathsf{u}(a,k',r',i'))\ \ [k' = k-1 \wedge 0 \le i' < k \le \mathsf{size}(a) \wedge\\ i' = i-1 \wedge r = r' + \mathsf{select}(a,i') \wedge n = \mathsf{select}(a,k')].\end{aligned}$$

Example 4.28. Recall strcpy from Example 3.7 and the analysis rules and equation from Example 1.2. The inductive proof follows roughly the same lines as the one for strlen and is found automatically by our tool (see Section 6). We reach a divergence in equations such as follows:

- $\mathsf{test}(x,n,\mathsf{v}(a,x,1)) \approx \mathsf{true}\ [0 \le n < \mathsf{size}(x) \wedge n < \mathsf{size}(a) \wedge \mathsf{select}(x,n) = 0 \wedge \forall i \in \{0,\ldots,n-1\}(\mathsf{select}(x,i) = 0) \wedge \mathsf{select}(x,0) \ne 0 \wedge \mathsf{select}(x,0) = \mathsf{select}(a,0)]$
- $\mathsf{test}(x,n,\mathsf{v}(b,x,2)) \approx \mathsf{true}\ [0 \le n < \mathsf{size}(x) \wedge n < \mathsf{size}(b) \wedge \mathsf{select}(x,n) = 0 \wedge \forall i \in \{0,\ldots,n-1\}(\mathsf{select}(x,i) = 0) \wedge \mathsf{select}(x,0) \ne 0 \wedge \mathsf{select}(x,0) = \mathsf{select}(b,0) \wedge \mathsf{select}(x,1) \ne 0 \wedge \mathsf{select}(b,1) = \mathsf{select}(x,1)]$
- $\mathsf{test}(x,n,\mathsf{v}(c,x,3)) \approx \mathsf{true}\ [\cdots \wedge \mathsf{select}(c,2) \ne 0 \wedge \mathsf{select}(c,2) = \mathsf{select}(x,2)].$

To generalize, we abstract $1,2,3$ by $k \ge 0$, collect similar statements into quantifications, and remove the endpoint. We quickly complete after this Generalization to

$$\begin{aligned}\mathsf{test}(x,n,\mathsf{v}(c,x,k)) \approx \mathsf{true}\ [0 \le n < \mathsf{size}(x) \wedge n < \mathsf{size}(c) \wedge \mathsf{select}(x,n) = 0 \wedge 0 \le k \wedge\\ \forall i \in \{0,\ldots,n-1\}(\mathsf{select}(x,i) \ne 0) \wedge \forall i \in \{0,\ldots,k-1\}(\mathsf{select}(x,i) \ne 0) \wedge\\ \forall i \in \{0,\ldots,k-1\}(\mathsf{select}(c,i) = \mathsf{select}(x,i))].\end{aligned}$$

Example 4.29. Let us compare two implementations of the Fibonacci function.

$$\begin{array}{lrll}
(1) & \mathsf{fibrec}(x) & \to & 0 & [x \le 0]\\
(2) & \mathsf{fibrec}(1) & \to & 1 & \\
(3) & \mathsf{fibrec}(x) & \to & \mathsf{plus}(\mathsf{fibrec}(x-1),\mathsf{fibrec}(x-2)) & [x \ge 2]\\
(4) & \mathsf{plus}(\mathsf{return}(x),\mathsf{return}(y)) & \to & \mathsf{return}(x+y) & \\
(5) & \mathsf{fibiter}(x) & \to & \mathsf{iter}(x,1,0,1) & \\
(6) & \mathsf{iter}(x,i,y,z) & \to & \mathsf{iter}(x,i+1,z,y+z) & [x \ge i]\\
(7) & \mathsf{iter}(x,i,y,z) & \to & \mathsf{return}(y) & [x < i]
\end{array}$$

Starting with the equation $\mathsf{fibrec}(x) \approx \mathsf{fibiter}(x)$ [true] eventually results in a divergence.

$$\begin{array}{rcl}
\mathsf{iter}(n,3,1,2) & \approx & \mathsf{plus}(\mathsf{iter}(m,\mathsf{iter}(m,2,1,1)),\mathsf{iter}(k,\mathsf{iter}(k,1,0,1)))\ [m=n-1\wedge k=n-2]\\
\mathsf{iter}(n,4,2,3) & \approx & \mathsf{plus}(\mathsf{iter}(m,\mathsf{iter}(m,3,1,2)),\mathsf{iter}(k,\mathsf{iter}(k,2,1,1)))\ [m=n-1\wedge k=n-2]\\
\mathsf{iter}(n,5,3,5) & \approx & \mathsf{plus}(\mathsf{iter}(m,\mathsf{iter}(m,4,2,3)),\mathsf{iter}(k,\mathsf{iter}(k,3,1,2)))\ [m=n-1\wedge k=n-2]
\end{array}$$

The proof is easily finished by using the following generalization.

$$\begin{array}{c}
\mathsf{iter}(n_3,i_3,z_3,z_4) \approx \mathsf{plus}(\mathsf{iter}(n_2,i_2,z_2,z_3),\mathsf{iter}(n_1,i_1,z_1,z_2))\\
[n_2=n_3-1\wedge n_1=n_2-2\wedge i_3=i_2+1\wedge i_2=i_1+1\wedge z_3=z_1+z_2\wedge z_4=z_2+z_3]
\end{array}$$

Thus, we can show equivalence of functions with wildly different time complexities (fibrec's running time is exponential in the input value, whereas that of fibiter is linear).

Example 4.30. Finally, we consider an example that Section 6, item 2 of Godlin and Strichman [2008] describes as beyond their method. Here, two recursive imperative programs calculating $\sum_{i=1}^{n} i$ are compared. The methods from Section 3 yield the following LCTRS:

$$\begin{array}{llllll}
(1) & \mathsf{f}(n) \to \mathsf{return}(n) & [n\le 0] & (4) & \mathsf{g}(n) \to \mathsf{return}(n) & [n\le 1]\\
(2) & \mathsf{f}(n) \to \mathsf{u}(n,\mathsf{f}(n-1)) & [n>0] & (5) & \mathsf{g}(n) \to \mathsf{v}(n,\mathsf{g}(n-1)) & [n>1]\\
(3) & \mathsf{u}(n,\mathsf{return}(m)) \to \mathsf{return}(n+m) & & (6) & \mathsf{v}(n,\mathsf{return}(m)) \to \mathsf{return}(n+m). &
\end{array}$$

Starting with the equation $\mathsf{f}(x) \approx \mathsf{g}(x)$ [true] eventually results in a divergence:

$$\begin{array}{ll}
\text{(CR.A):} & \mathsf{u}(x,\mathsf{u}(y_1,\mathsf{g}(y_2))) \approx \mathsf{v}(x,\mathsf{u}(z_1,\mathsf{g}(z_2)))\\
& [x>1\wedge y_1=x-1\wedge z_1=x-1\wedge y_2=y_1-1\wedge z_2=z_1-1]\\
\text{(CR.B):} & \mathsf{u}(x,\mathsf{u}(y_1,\mathsf{u}(y_2,\mathsf{g}(y_3)))) \approx \mathsf{v}(x,\mathsf{u}(z_1,\mathsf{u}(z_2,\mathsf{g}(z_3))))\\
& [x>1\wedge y_1=x-1\wedge z_1=x-1\wedge y_2=y_1-1\wedge z_2=z_1-1\wedge y_3=y_2-1\wedge z_3=z_2-1]\\
\text{(CR.C):} & \mathsf{u}(x,\mathsf{u}(y_1,\mathsf{u}(y_2,\mathsf{u}(y_3,\mathsf{g}(y_4))))) \approx \mathsf{v}(x,\mathsf{u}(z_1,\mathsf{u}(z_2,\mathsf{u}(z_3,\mathsf{g}(z_4))))) [\dots].
\end{array}$$

As the constraints imply that each $y_i = z_i$, these equations can all be generalized to $\mathsf{u}(x,\mathsf{u}(y,z)) \approx \mathsf{v}(x,\mathsf{u}(y,z))$ $[x>1]$. Again, the proof is quickly completed.

4.4. Soundness and Completeness of Rewriting Induction

We now give an intuition on how to prove Theorem 4.4. The complete proof can be found in Appendix B. We follow the proof method of Sakata et al. [2009], which builds on the original proof idea in Reddy [1990]. This uses the relation $\leftrightarrow_{\mathcal{E}}$, defined by

$$C[s\gamma]_p \leftrightarrow_{\mathcal{E}} C[t\gamma]_p \text{ if } s \approx t\ [\varphi] \in \mathcal{E} \text{ or } t \approx s\ [\varphi] \in \mathcal{E}, \text{ and } \gamma \text{ respects } \varphi$$

for $\mathcal{E}$ a set of equations. The proof is split up into several auxiliary lemmas. To start, we have the following lemma.

Lemma 4.31. *All equations in $\mathcal{E}$ are inductive theorems if and only if $\leftrightarrow_{\mathcal{E}} \subseteq \leftrightarrow_{\mathcal{R}}^*$ on ground terms (so if s,t are ground and $s \leftrightarrow_{\mathcal{E}} t$, then also $s \leftrightarrow_{\mathcal{R}}^* t$).*

This is obvious from the definitions. The next lemma originates in Sakata et al. [2009], which is adapted from Koike and Toyama [2000] and is key to our method.

Lemma 4.32 [Sakata et al. 2009]. *Let $\to_1$ and $\to_2$ be binary relations over some set A. Then $\leftrightarrow_1^* = \leftrightarrow_2^*$ if all of the following hold:*

$-\!\!\to_1 \subseteq \to_2$,
$-\!\!\to_2$ is well founded, and
$-\!\!\to_2 \subseteq (\to_1 \cdot \to_2^* \cdot \leftrightarrow_1^* \cdot \leftarrow_2^*)$.

Proof. It follows from $\to_1 \subseteq \to_2$ that $\leftrightarrow_1^* \subseteq \leftrightarrow_2^*$. To show that $\leftrightarrow_2^* \subseteq \leftrightarrow_1^*$, we prove $\to_2^* \subseteq \leftrightarrow_1^*$ by well-founded induction on $\to_2$. Since the base case $s \to_2^* s$ is clear, we

suppose that $s \to_2 t \to_2^* u$. As $\to_2 \subseteq (\to_1 \cdot \to_2^* \cdot \leftrightarrow_1^* \cdot \leftarrow_2^*)$, there must be some a, b, c such that $s \to_1 a \to_2^* b \leftrightarrow_1^* c \leftarrow_2^* t$. Since $\to_1 \subseteq \to_2$ (i.e., $s \to_2 a$), we can apply the induction hypothesis both on a and on t, so $a \leftrightarrow_1^* b \leftrightarrow_1^* c \leftrightarrow_1^* t$ and $t \leftrightarrow_1^* u$. Therefore, $s \leftrightarrow_1^* u$. □

We will use Lemma 4.32 with $\to_{\mathcal{R}}$ for $\to_1$, and $\to_{\mathcal{R}\cup\mathcal{H}}$ for $\to_2$. Soundness of the algorithm then follows if $\leftrightarrow_{\mathcal{E}}$ is included in $\leftrightarrow_{\mathcal{H}}^*$ whenever $(\mathcal{E}, \emptyset, \mathit{flag}) \vdash_{\mathtt{ri}}^* (\emptyset, \mathcal{H}, \mathit{flag}')$.

Theorem 4.4 is the combination of Lemma 4.31 with following Lemmas 4.33 and 4.34.

LEMMA 4.33. *If* $(\mathcal{E}, \emptyset, \mathit{flag}) \vdash_{\mathtt{ri}}^* (\emptyset, \mathcal{H}, \mathit{flag}')$*, then* $\leftrightarrow_{\mathcal{E}} \subseteq \leftrightarrow_{\mathcal{R}}$ *holds on ground terms.*

PROOF IDEA. Let $\leftarrow\!\!\Vdash_{\mathcal{E}}$ denote a *parallel* application of zero or more $\leftrightarrow_{\mathcal{E}}$ steps. We first show that $(\mathcal{E}, \mathcal{H}, \mathit{flag}) \vdash_{\mathtt{ri}} (\mathcal{E}', \mathcal{H}', \mathit{flag}')$ by any rule other than COMPLETENESS implies both (a) $\leftarrow\!\!\Vdash_{\mathcal{E}} \subseteq \left(\to_{\mathcal{R}\cup\mathcal{H}'}^* \cdot \leftarrow\!\!\Vdash_{\mathcal{E}'} \cdot \leftarrow_{\mathcal{R}\cup\mathcal{H}'}^*\right)$ on ground terms, and (b) if $\to_{\mathcal{R}\cup\mathcal{H}} \subseteq (\to_{\mathcal{R}} \cdot \to_{\mathcal{R}\cup\mathcal{H}}^* \cdot \leftarrow\!\!\Vdash_{\mathcal{E}} \cdot \leftarrow_{\mathcal{R}\cup\mathcal{H}}^*)$ on ground terms, then $\to_{\mathcal{R}\cup\mathcal{H}'} \subseteq (\to_{\mathcal{R}} \cdot \to_{\mathcal{R}\cup\mathcal{H}'}^* \cdot \leftarrow\!\!\Vdash_{\mathcal{E}'} \cdot \leftarrow_{\mathcal{R}\cup\mathcal{H}'}^*)$ on ground terms. We show this by considering how each step alters $\mathcal{E}$ and $\mathcal{H}$, which we use to see that $(\mathcal{E}, \mathcal{H}, \mathit{flag}) \vdash_{\mathtt{ri}}^* (\mathcal{E}', \mathcal{H}', \mathit{flag}')$ implies (a) and (b), by induction on the total number of $\vdash_{\mathtt{ri}}$-steps in the derivation (counting also the hidden steps inside COMPLETENESS). Thus, if $(\mathcal{E}, \emptyset, \mathit{flag}) \vdash_{\mathtt{ri}}^* (\emptyset, \mathcal{H}, \mathit{flag}')$, then $\to_{\mathcal{R}\cup\mathcal{H}} \subseteq \to_{\mathcal{R}} \cdot \to_{\mathcal{R}\cup\mathcal{H}}^* \cdot \leftarrow_{\mathcal{R}\cup\mathcal{H}}^*$, so we can apply Lemma 4.32 to conclude that $\to_{\mathcal{R}}$ and $\to_{\mathcal{R}\cup\mathcal{H}}$ are the same (on ground terms). Therefore, and by property (a), $\leftrightarrow_{\mathcal{E}} \subseteq \leftarrow\!\!\Vdash_{\mathcal{E}} \subseteq \to_{\mathcal{R}\cup\mathcal{H}}^* \cdot \leftarrow_{\mathcal{R}\cup\mathcal{H}}^* \subseteq \leftrightarrow_{\mathcal{R}}^*$. □

LEMMA 4.34. *If* $\mathcal{R}$ *is confluent and* $(\mathcal{E}, \emptyset, \text{COMPLETE}) \vdash_{\mathtt{ri}}^* \bot$*, then* $\leftrightarrow_{\mathcal{E}} \not\subseteq \leftrightarrow_{\mathcal{R}}$ *holds on ground terms.*

PROOF IDEA. By confluence and termination together, we can speak of the normal form $u\!\downarrow_{\mathcal{R}}$ of any term u; if u is ground, then by quasi-reductivity its normal form is a ground constructor term. A property of confluence is that if $w \leftrightarrow_{\mathcal{R}}^* q$, then $w\!\downarrow_{\mathcal{R}} = q\!\downarrow_{\mathcal{R}}$. Thus, it suffices to prove that for some $s \approx t\ [\varphi] \in \mathcal{E}$ there is a ground constructor substitution γ that respects this equation such that $s\gamma \neq t\gamma$. We first note that if $(\mathcal{E}, \mathcal{H}, \text{COMPLETE}) \vdash_{\mathtt{ri}} \bot$, then this can only be a DISPROVE step; in all cases, the equation that causes the disproof has this property. We also see, by examining the various inference rules, that if $(\mathcal{E}_1, \mathcal{H}_1, \text{COMPLETE}) \vdash_{\mathtt{ri}} (\mathcal{E}_2, \mathcal{H}_2, \text{COMPLETE})$ and both (a) $\to_{\mathcal{R}\cup\mathcal{H}_1} \subseteq \to_{\mathcal{R}} \cdot \to_{\mathcal{R}\cup\mathcal{H}_1}^* \cdot \leftarrow\!\!\Vdash_{\mathcal{E}} \cdot \leftarrow_{\mathcal{R}\cup\mathcal{H}_1}^*$ and (b) $\leftrightarrow_{\mathcal{E}_1} \cup \leftrightarrow_{\mathcal{H}_1} \subseteq \leftrightarrow_{\mathcal{R}}^*$ on ground terms, then also $\leftrightarrow_{\mathcal{E}_2} \cup \leftrightarrow_{\mathcal{H}_2} \subseteq \leftrightarrow_{\mathcal{R}}^*$ on ground terms. In a reduction $(\mathcal{E}, \emptyset, \text{COMPLETE}) = (\mathcal{E}_1, \mathcal{H}_1, \mathit{flag}_1) \vdash_{\mathtt{ri}} \cdots \vdash_{\mathtt{ri}} (\mathcal{E}_n, \mathcal{H}_n, \mathit{flag}_n) \vdash_{\mathtt{ri}} \bot$, we may assume (a) by the observations in the proof of Lemma 4.33, and (b) is inductively preserved. As $\leftrightarrow_{\mathcal{E}_n\cup\mathcal{H}_n}$ cannot be included in $\leftrightarrow_{\mathcal{R}}^*$, therefore neither can $\leftrightarrow_{\mathcal{E}} = \leftrightarrow_{\mathcal{E}_1\cup\mathcal{H}_1}$. We complete by Lemma 4.31. □

5. GENERALIZING EQUATIONS

Divergence, as encountered in all examples in Section 4, is very common in inductive theorem proving: we often need a more general claim to obtain a stronger induction hypothesis. As it is not always easy to find a suitable generalization, the (automatic) generation of suitable generalizations, and lemma equations for POSTULATE, has been extensively investigated [Bundy et al. 2005; Kapur and Sakhanenko 2003; Kapur and Subramaniam 1996; Nakabayashi et al. 2010; Urso and Kounalis 2004; Walsh 1996].

Also for transformed procedural programs, we will certainly need a large variety of lemma generation techniques to handle most practical cases. We start the work by proposing two methods to generalize equations, specialized to deal with constraints.

5.1. Generalizing Initializations

Our first and most important technique fundamentally relies on the constrained setting. Although it may appear deceptively simple (at its core, the generalization just drops a part of the constraint), it is particularly effective for dealing with loops.

Example 5.1. Let us state the rules of $\mathcal{R}_{\mathsf{fact}}$ from Example 4.2 in an alternate way. We replace rule (1) $\mathsf{factiter}(x) \to \mathsf{iterm}(x, 1, 1)$ by (1′): $\mathsf{factiter}(x) \to \mathsf{iter}(x, v_1, v_2)$ $[v_1 = 1 \wedge v_2 = 1]$. In other words, the values corresponding to *initializations* `int z = 1; int i = 1;` are moved into the constraint. Evidently, this change does not alter the relation $\to_{\mathcal{R}}$.

Now consider what happens if we use the same steps as in Examples 4.2 through 4.14. The resulting proof has the same shape but with more complex equations. Some instances include the following.

$$\begin{aligned}
&(\text{FCT.B}'): \ \mathsf{iter}(n, v_1, v_2) \approx \mathsf{factrec}(n) \ \ [n \geq 1 \wedge v_1 = 1 \wedge v_2 = 1]\\
&(\text{FCT.D}'): \ \mathsf{iter}(n, z_1, i_1) \approx \mathsf{factrec}(n) \ \ [n \geq 1 \wedge v_1 = 1 \wedge v_2 = 1 \wedge z_1 = v_1 * v_2 \wedge i_1 = v_2 + 1]\\
&(\text{FCT.J}'): \ \mathsf{mul}(n, \mathsf{iter}(m, z_1, i_1)) \approx \mathsf{iter}(n, z_2, i_2) \ [n > 1 \wedge v_1 = 1 \wedge v_2 = 1 \wedge m = n - 1 \wedge\\
&\qquad\qquad z_1 = v_1 * v_2 \wedge i_1 = v_2 + 1 \wedge z_2 = z_1 * i_1 \wedge i_2 = i_1 + 1]\\
&(\text{FCT.K}'): \ \mathsf{mul}(n, \mathsf{iter}(m, z_2, i_2)) \approx \mathsf{iter}(n, z_3, i_3) \ [n > 1 \wedge v_1 = 1 \wedge v_2 = 1 \wedge m = n - 1 \wedge\\
&\qquad z_1 = v_1 * v_2 \wedge i_1 = v_2 + 1 \wedge z_2 = z_1 * i_1 \wedge i_2 = i_1 + 1 \wedge z_3 = z_2 * i_2 \wedge i_3 = i_2 + 1]
\end{aligned}$$

Here the left- and right-hand side of the divergent equations (FCT.J′) and (FCT.K′) are the same modulo variable renaming, while the constraint grows. Essentially, we keep track of parts of the history of an equation in its constraint. We generalize (FCT.J′) by dropping all clauses $v_i = q_i$ where v_i is an initialization variable and q_i a value. We rename the variables v_i (as they no longer play a special role) and obtain the following.

$$\begin{aligned}
&(\text{FCT.M}'): \qquad \mathsf{mul}(n, \mathsf{iter}(m, z_1, i_1)) \approx \mathsf{iter}(n, z_2, i_2)\\
&\qquad [n > 1 \wedge m = n - 1 \wedge z_1 = x_1 * x_2 \wedge i_1 = x_2 + 1 \wedge z_2 = z_1 * i_1 \wedge i_2 = i_1 + 1]
\end{aligned}$$

We can complete the derivation with (FCT.M′) as we did with (FCT.M) before.

Formally, what we do here is threefold. First, we alter the set of rules we work from.

Definition 5.2 (*Initialization-Free Rules*). Given $\mathcal{R}$, fix a set $\mathcal{V}_{\mathtt{init}} \subsetneq \mathcal{V}$ of variables not occurring in $\mathcal{R}$. The *initialization-free* counterpart $\mathcal{R}'$ of $\mathcal{R}$ is obtained by stepwise replacing any rule $\ell \to C[f(r_1, \ldots, r_i, \ldots, r_n)]$ $[\varphi]$ with $f \in \mathcal{D}$ and r_i a value by $\ell \to C[f(r_1, \ldots, v, \ldots, r_n)]$ $[\varphi \wedge v = r_i]$ for some fresh $v \in \mathcal{V}_{\mathtt{init}}$, until no such rules remain.

Then, to apply Generalization to an equation $s \approx t$ $[\varphi_1 \wedge \cdots \wedge \varphi_n]$, we choose

$$s \approx t \left[\bigwedge \{\varphi_i \mid 1 \leq i \leq n \wedge \varphi_i \text{ does not have the form } v = u \text{ with } v \in \mathcal{V}_{\mathtt{init}} \text{ and } u \in \mathcal{V}al\} \right]$$

as the generalized equation and rename its variables in $\mathcal{V}_{\mathtt{init}}$ to variables in $\mathcal{V}$.

Finally, we restrict the Simplification and Expansion steps to preserve initialization constraints throughout the proof. The strategy we use in Ctrl—which includes an approach to handle the $v \in \mathcal{V}_{\mathtt{init}}$—is described in Section 6.1, but in particular:

—When we rename rules for use in Simplification or Expansion, the renaming must respect membership in $\mathcal{V}_{\mathtt{init}}$ (i.e., if x is renamed to y, then $y \in \mathcal{V}_{\mathtt{init}}$ if and only if $x \in \mathcal{V}_{\mathtt{init}}$).
—In $\sim$-steps, any conjuncts $v = n$ are ignored: to simplify $s \approx t$ $[\varphi \wedge v_1 = n_1 \wedge \cdots \wedge v_k = n_k]$, we modify $s \approx t$ $[\varphi]$, obtaining $s' \approx t'$ $[\varphi']$, and continue with $s' \approx t'$ $[\varphi \wedge v_1 = n_1 \wedge \cdots \wedge v_k = n_k]$. Thus, we avoid, for example, translating $f(v_i)$ $[v_i = 0]$ back to $f(0)$ [true].

5.2. Abstracting Equivalent Recursive Calls

Our second generalization technique aims to remove recursive symbols where possible.

Definition 5.3. For symbols f, g, let $f \rightsquigarrow g$ if there is a rule $f(\overrightarrow{\ell}) \to r$ $[\varphi]$ with g a symbol in r. A symbol f is *recursive* if it is a defined symbol with $f \rightsquigarrow^+ f$.

The key idea is to identify equivalent occurrences of a recursive call on both sides of an equation and to replace them by a variable. For example, $\mathsf{g}(x) + \mathsf{f}(y) \approx \mathsf{f}(z) + \mathsf{g}(x)$ $[y \geq z \wedge y \leq z]$ is replaced by $a + b \approx b + a$ [true], because for values k, n, m: if $n \geq m \wedge m \leq n$ holds, then both $\mathsf{g}(k)$ and $\mathsf{g}(k)$, as well as $\mathsf{f}(n)$ and $\mathsf{f}(m)$, are syntactically equal.

Definition 5.4. A *recursion-abstraction* of $s \approx t$ $[\varphi]$ is any equation of the form $C[x_1, \ldots, x_n] \approx D[x_{i_1}, \ldots, x_{i_n}]$ such that (a) $s = C[s_1, \ldots, s_n]$ and $t = D[t_{i_1}, \ldots, t_{i_n}]$ for some $\vec{s}, \vec{t}$; (b) $\{i_j \mid 1 \leq j \leq n\} = \{1, \ldots, n\}$; (c) neither C nor D contain recursive symbols; (d) each s_j and t_j has a recursive symbol as root symbol; (e) for $1 \leq i \leq n$ and all ground substitutions γ that respect $s \approx t$ $[\varphi]$: $s_i\gamma = t_i\gamma$; and (f) $x_1, \ldots, x_n$ are fresh with respect to s, t.

For a given equation, at most one choice of C, D is possible, and there are only finitely many permutations $i_1, \ldots, i_n$. Requirement (e) can be checked by confirming that an equation $s_j \approx t_j$ $[\varphi]$ is removed by the combination of EQ-DELETION and DELETION.

Example 5.5. In Example 4.30, we find an abstraction for (CR.A) by choosing $C = \mathsf{u}(x, \mathsf{u}(y_1, \Box))$, $D = \mathsf{v}(x, \mathsf{u}(z_1, \Box))$, $s_1 = \mathsf{g}(y_2)$ and $t_1 = \mathsf{g}(z_2)$. Requirement (e) holds: if we write φ for the constraint of (CR.A), EQ-DELETION on $\mathsf{g}(y_2) \approx \mathsf{g}(z_2)$ $[\varphi]$ produces the unsatisfiable constraint $\varphi \wedge y_2 \neq z_2$. Thus, we generalize the equation to $\mathsf{u}(x, \mathsf{u}(y_1, a)) \approx \mathsf{v}(x, \mathsf{u}(z_1, a))$ $[\varphi]$, which is $\sim$-equivalent to the equation used in Example 4.30.

Example 5.6. Given $\mathsf{g}(x) + \mathsf{f}(y) \approx \mathsf{f}(z) + \mathsf{g}(x)$ $[y \geq z \wedge y \leq z]$, let C and D be $\Box + \Box$, $s_1 = \mathsf{g}(x), s_2 = \mathsf{f}(y), t_1 = \mathsf{g}(x), t_2 = \mathsf{f}(z), i_1 = 2$, and $i_2 = 1$. We must see that for all γ that respect $y \geq z \wedge y \leq z$: $\mathsf{g}(x)\gamma = \mathsf{g}(x)\gamma$ and $\mathsf{f}(y)\gamma = \mathsf{f}(z)\gamma$. Both are easily confirmed, so we generalize to $x_1 + x_2 \approx x_2 + x_1$ $[y \geq z \wedge y \leq z] \sim a + b \approx b + a$ [true] as suggested.

One can see this generalization heuristic as an instance of the inference rule SPECIALIZATION by Aubin [1979] for unconstrained explicit induction, restricted to recursive function calls and combined with SUBSTITUTIVITY OF EQUALITY from the same work. Here we lift equality from syntactic level to semantic level in SMT.

5.3. Discussion

The first method to generalize equations is strong (Section 5.1), but only for equations of a specific form: we can only use the method if the equation part of the divergence has the same shape every time. This is the case for fact, because the rule that causes the divergence has the form $\mathsf{iter}(x_1, \ldots, x_n) \to \mathsf{iter}(r_1, \ldots, r_n)$ $[\varphi]$, preserving its outer shape.

In general, the method is most likely to be successful for the analysis of tail-recursive functions (with accumulators), such as those obtained from procedural programs. We can also handle mutually recursive functions, like $\mathsf{u}(x_1, \ldots, x_n) \to \mathsf{w}(r_1, \ldots, r_m)$ $[\varphi]$ and $\mathsf{w}(y_1, \ldots, y_m) \to \mathsf{u}(q_1, \ldots, q_n)$ $[\psi]$. It is not suitable for analyzing systems with (only) non–tail recursion, however. Here, the second technique comes in (Section 5.2). Although we do not claim that this technique is very powerful, it is often useful to eliminate apparently simple equations. It is also straightforward to use in practice.

Note that strlen and strcpy also have the required tail-recursive form to successfully use the first generalization method. However, here we additionally have to collect multiple clauses into a quantification before generalizing, as with equation (LEN.I).

One may wonder if generalizing initializations loses too much. For example, when removing $v_i = 1$, we also forget that $v_i \geq 0$. However, this is usually not an issue: if a rule is constrained with $v_i \geq 0$, this clause is added to the constraint of the equation via EXPANSION before we generalize, as in the expansion from (LEN.B). There is, however, a possible issue with losing information on the relations *between* variables; we will say more on this in Section 6.2.

6. IMPLEMENTATION

The method for program verification in this article can be broken down into two parts:

(1) transforming a procedural program into an LCTRS;
(2) proving correctness properties on this LCTRS using rewriting induction.

An initial implementation of part 1, limited to functions on integers and one-dimensional statically allocated integer arrays, is available at http://www.trs.css.i.nagoya-u.ac.jp/c2lctrs/.

In future work, it is our hope to extend this implementation to include the remaining features discussed in Section 3 and Appendix A.2, such as floating points and explicit pointers.

Part 2, the core method on LCTRSs, has been implemented in our tool Ctrl [Kop and Nishida 2015], along with basic techniques to verify termination, confluence, and quasi-reductivity. To handle constraints, the tool is coupled both with a small internal reasoner and the external SMT solver Z3 [de Moura and Bjørner 2008]. Z3 is equipped to prove *unsatisfiability* as well as satisfiability, which is essential for testing *validity*.

The internal reasoner serves to detect satisfiability or validity of simple statements quickly, without a call to an SMT solver, and to preprocess certain kinds of queries that arise often (e.g., for termination proving by polynomial interpretations, we preprocess queries with $\exists\forall$-quantifier prefix to $\exists$-queries). The reasoner is also used to simplify the constraints of equations, such as by combining statements into quantifications (which is an essential part of the derivations for functions like strlen or strcpy).

We also translate our array formulas into the SMT-LIB array format as discussed in Section 3.6, encoding an array as a function from $\mathbb{Z}$ to $\mathbb{Z}$ with a second variable for its size.

The latest version of Ctrl (tool paper: Kop and Nishida [2015]) can be downloaded at http://cl-informatik.uibk.ac.at/software/ctrl/.

6.1. Strategy

Let us discuss the various choices made during a derivation with rewriting induction.

6.1.1. What Inference Rule to Apply. Ctrl always selects the first rule (combination) from the following list:

(1) EQ-DELETION (if applicable) immediately followed by DELETION;
(2) DISPROVE, but without the limitation to COMPLETE proof states;
(3) CONSTRUCTOR;
(4) SIMPLIFICATION;
(5) a limited form of EXPANSION;
(6) GENERALIZATION using a recursion-abstraction;
(7) GENERALIZATION of all initialization variables $v_i \in \mathcal{V}_{\mathtt{init}}$ at once;
(8) the full form of EXPANSION.

6.1.2. Generalization and Backtracking. Core to the rewriting induction process is a backtracking mechanism. Every proof state $(\mathcal{E}, \mathcal{H})$ keeps track of all ancestor states on which GENERALIZATION was applied; a state is COMPLETE if it has no such ancestors. The completeness restriction on DISPROVE is dropped; however, when DISPROVE succeeds on an incomplete state, the prover does not conclude failure but instead backtracks to the most recent ancestor and continues without (immediately) generalizing. Typically, if a GENERALIZATION is attempted too soon in the proof and results in an unsound equation, this can be derived very quickly, which allows Ctrl to conclude failure of the GENERALIZATION step and to move on to the remaining expansions.

Example 6.1. Following Example 4.26 (but altered with initialization-free rules), our strategy moves from $(\{(\text{LEN.A}')\}, \emptyset)$ to $(\{(\text{LEN.B}')\}, \emptyset)$ as before. But here, "restricted expansion" does not apply (as we will see in Example 6.3), nor is there a recursion-abstraction. Thus, we generalize the initializations, obtaining

$$\left(\left\{\begin{array}{ll}(\text{BGEN}) & \mathsf{u}(x, r_0) \approx \mathsf{return}(n)\\ & [0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0, \ldots, n-1\}(\mathsf{select}(x, i) \ne 0) \wedge \mathsf{select}(x, n) = 0]\end{array}\right\},\ \emptyset\right).$$

We store $(\{(\text{LEN.B}')\}, \emptyset)$ as an ancestor state of $(\{(\text{BGEN})\}, \emptyset)$. The only option now is Expansion. Expanding in the left-hand side gives three equations, including

$$\mathsf{return}(r_0) \approx \mathsf{return}(n)$$
$$[0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0, \ldots, n-1\}(\mathsf{select}(x, i) \ne 0) \wedge \mathsf{select}(x, n) = 0].$$

Constructor gives $r_0 \approx n\ [\varphi]$, where φ is satisfied by, for example, $[r_0 := 0, n := 1, x := [1, 0]]$; by Disprove, we obtain $\bot$. However, the state is incomplete, as it has an ancestor stored. Thus, we backtrack to $(\{(\text{LEN.B}')\}, \emptyset)$ and continue with full expansion.

The Completeness rule is implemented via the same mechanism: if $(\mathcal{E}, \mathcal{H})$ has a most recent ancestor $(\mathcal{E}', \mathcal{H}')$ with $\mathcal{E} \subseteq \mathcal{E}'$, then $(\mathcal{E}', \mathcal{H}')$ is dropped from the ancestor list. If a Disprove succeeds when the list is empty, we conclude failure, resulting in NO if the system is confluent and MAYBE otherwise.

Example 6.2. In Example 4.16, we would add $(\{(\text{FCT.J})\}, \{(\text{FCT.D}^{-1})\})$ to the list of ancestors when generalizing (FCT.J) to (FCT.M). Once (FCT.T) is removed in Example 4.19, we are allowed to remove this state from the list (although since the proof is finished at that point, it is not really necessary in this example).

Aside from backtracking due to Disprove, there is a second backtracking mechanism: although Simplification and Expansion prioritize choices (for positions and rules) most likely to result in success, sometimes the first choice does not work out, but the second one does. Thus, Ctrl uses an evaluation limit: when a path has more than N expansions, it is aborted, and the prover backtracks to a direct parent. Ctrl starts with $N = 2$ and increases this limit if it does not result in a successful proof or disproof.

6.1.3. Simplification. For Simplification, there are three choices to be made: the position, the rule, and how to instantiate fresh variables in that rule.

For the position, Ctrl selects the leftmost, innermost position where a rule matches. This prevents a need to reevaluate a term after its subterms change.

For the rule, rules in $\mathcal{H}$ are attempted before rules in $\mathcal{R}$; if a rule leads to a (presumed) divergence, the backtracking mechanism ensures that the next one is tried.

In some cases—in particular for induction rules—the right-hand side and perhaps the constraint of a rule contain variables not occurring in the left-hand side, such as (FCT.M^{-1}) in Example 4.14 and (LEN.K) in Example 4.26. Here, Ctrl tries to instantiate as many variables in the rule by variables in the equation as possible. To rewrite an equation $s \approx t\ [\varphi_1 \wedge \cdots \wedge \varphi_n]$ at the root of s with a rule $\ell \to r\ [\psi_1 \wedge \cdots \wedge \psi_m]$, we first determine a γ such that $s = \ell\gamma$ and $\gamma(v_i) = v_i$ for all $v_i \in \mathcal{V}_{\mathtt{init}}$. If any ψ_i has the form $C[x, y_1, \ldots, y_k]$ with $x \in \mathit{Dom}(\gamma)$ and all $y_i \notin \mathit{Dom}(\gamma)$, and there is some $\varphi_j = C\gamma[\gamma(x), s_1, \ldots, s_k]$, then we extend γ with $[y_i := s_i]$ for all i. This process is finite and corresponds to the choices for the equations (FCT.S) and (LEN.O). Other variables are chosen fresh.

Note: If some rule can be applied but the backtracking mechanism aborts all attempts, Ctrl backtracks to the parent state rather than continuing with Expansion. This is because testing suggests that allowing Expansion to be applied on terms not in $\mathcal{R}$-normal form is generally not effective and causes an explosive number of states.

6.1.4. Expansion. To categorize EXPANSIONs for steps (5) and (8) of Section 6.1.1, we analyze *recursion*. Let $f \gtrsim g$ if $f \rightsquigarrow^* g$ (following Definition 5.3), and let $f \succ g$ if $f \gtrsim g$ and $g \not\gtrsim f$. Symbols are split into five categories: *constructors*, *calculation symbols*, *nonrecursive defined symbols*, *tail-recursive symbols*, and *non–tail-recursive symbols*. A recursive symbol is *tail recursive* if its only defining rules (in $\mathcal{R}$) have either the form $f(\ell_1, \ldots, \ell_k) \to x$ $[\varphi]$ with x a variable or the form $f(\ell_1, \ldots, \ell_k) \to g(r_1, \ldots, r_m)$ $[\varphi]$ with $f \succ h$ for all h in any r_i. Recursive functions not of this form are *non–tail recursive*.

An expansion of $s \simeq t$ $[\varphi]$ at position p of s, with $s_{|p} = f(\vec{u})$, is *restricted*—so eligible for step (5)—if (a) f is nonrecursive, or (b) the induction rule $s \to t$ $[\varphi]$ is admissible and either f is tail recursive and $(Var(s) \cup Var(t)) \cap \mathcal{V}_{\mathtt{init}} = \emptyset$, or f is non–tail recursive. The induction rule is added only in case (b). Here, a rule $\rho : g(\ell_1, \ldots, \ell_k) \to r$ $[\varphi]$ is *admissible* if $\mathcal{R} \cup \mathcal{H} \cup \{\rho\}$ is terminating *and* $g \in \mathcal{D}$: we do not add rules with a constructor or calculation symbol as root symbol g, as this makes it harder to prove termination, which may prevent the addition of more promising rules later on.

For *unrestricted* expansion, an induction rule is added when admissible, unless f is tail recursive. The unrestricted tail-recursive case concerns rules such as those got from (FCT.J), (LEN.B), and (LEN.E), which—testing suggests—are typically not useful. Omitting them lets Ctrl skip many termination checks, a bottleneck in the process. Similarly, we do not add induction rules when expanding at a nonrecursive position.

Example 6.3. In Examples 4.2 through 4.18, the first expansion occurs in $(\{(\text{FCT.D}')\}, \emptyset, \text{COMPLETE})$, in the right-hand side. This is not an arbitrary choice: restricted expansion cannot be used with the tail-recursive symbol iter, only the non–tail-recursive symbol factrec. Then our strategy closely follows the given derivation. When we reach $(\{(\text{FCT.J})\}, \{(\text{FCT.D}^{-1})\}, \text{COMPLETE})$, restricted expansion is impossible, so we generalize instead. After this, an expansion on the iter symbol on either side is restricted. We can complete the example without backtracking or using unrestricted EXPANSION.

For the position at which to expand, we follow the same approach as for SIMPLIFICATION, trying all suitable positions via the backtracking mechanism. However, rather than a pure leftmost innermost choice, in the restricted case (step (5) of Section 6.1.1), we prioritize the more promising equations by first attempting expansions on a non–tail-recursive symbol, then those with a nonrecursive defined symbol, and finally those with a tail-recursive one. In the unrestricted setting, we follow the leftmost innermost strategy.

Testing shows that this method is very effective for proving equivalence between a non–tail-recursive and a tail-recursive function (as needed for equivalence of a recursive and an iterative C function). The examples of Section 4 show its effect: by eliminating the non–tail-recursive functions early on, we are more likely to arrive at a diverging sequence where all equations have the same outer shape (e.g., $(\mathsf{u}(\overrightarrow{q}_i) \approx C[\mathsf{u}(\overrightarrow{v}_i), \mathsf{u}(\overrightarrow{w}_i)] \mid i \in \mathbb{N})$). As observed in Section 5.3, this is ideal for our generalization method.

Following an EXPANSION, we first process those new equations in $Expd(s \approx t\ [\varphi], p)$ whose multiset of new symbols is smallest in the recursion order $\succ$. Thus, for example, in Example 4.10, after expanding (FCT.D), we consider (FCT.E), which has new symbols {return, 1}, before (FCT.F), with new symbols {mul, factrec, −, 1}, since factrec $\succ$ return, 1. Intuitively, "smaller" terms are "closer" to the end of a function, which allows DISPROVE to succeed faster and thus aids the backtracking mechanism.

6.1.5. Constraint Modification. Following SIMPLIFICATION and EXPANSION, Ctrl modifies the constraint as follows. First, when a clause φ_i in the constraint $\varphi_1 \wedge \cdots \wedge \varphi_n$ is implied by the others, it is removed unless it is a definition clause $v_i = n$. We also remove

clauses for variables that do not play a role. Most importantly, Ctrl introduces *ranged quantifications* $\forall x \in \{k_1, \ldots, k_n\}(\varphi(x))$ whenever possible, provided $n \geq 3$ (to lessen the effect of coincidence). Formally, we could describe our approach as follows:

> If φ has clauses $C[a], C[b], C[c]$ for some context C and variables a, b, c, as well as $b = f(a)$ and $c = f(b)$, then we may replace the C-clauses by $\forall i \in \{0, \ldots, 2\}(C[f^i(a)])$.

This is more general than what we use; it lets us, for instance, replace $a[i] = 0 \wedge a[j] = 0 \wedge a[k] = 0 \wedge j = i+2 \wedge k = j+2$ by $\forall l \in \{0, \ldots, 2\}(a[i+2 \cdot l] = 0)$, for $f = \lambda x.x+2$. But to represent f^i, Ctrl must know the relevant theory. Therefore, we currently only consider clauses where $b = a + 1$ and $c = b + 1$, and replace them by $\forall i \in \{a, \ldots, c\}(C[i])$. Since we implement loop counters as integers, this still captures a large group of constraints.

After $\forall$-introduction, if a boundary of the range (0 and 2 in the example) is some $v_i \in \mathcal{V}_{\mathtt{init}}$, we replace it by the value it is defined as to avoid generalizing the starting point of a quantification. Thus, for example, $\forall j \in \{v_0, \ldots, k\}(\mathsf{select}(x, j) \neq 0) \wedge v_0 = 0 \wedge i = v_0 + 1 \wedge k = i + 1$ is replaced by $\forall j \in \{0, \ldots, k\}(\mathsf{select}(x, j) \neq 0) \wedge v_0 = 0 \wedge i = v_0 + 1 \wedge k = i + 1$.

6.1.6. Nonconfluence. Our strategy is admittedly unfair to nonconfluent systems. A successful application of DISPROVE is treated as evidence of an unsound equation, which is not the case without confluence: the nonconfluent (LC)TRSs $\mathcal{R} = \{\mathsf{f} \to \mathsf{a},\ \mathsf{f} \to \mathsf{b},\ \mathsf{g} \to \mathsf{a},\ \mathsf{g} \to \mathsf{b}\}$ along with the inductive theorem $\mathsf{f} \approx \mathsf{g}$ highlights that we only have to prove that two functions can produce the same result, not that they always do.

This is deliberate: when proving that two functions produce the same result, we can see nonconfluent LCTRSs as inherently incorrect. Thus, we focus on confluent systems. For LCTRSs whose confluence is unknown, it is preferable to show nonequivalence (which translates to a MAYBE in the output) over equivalence.

6.2. Experiments

To assess performance and precision of Ctrl empirically, we tested five assignments from a group of students in the first-year programming course in Nagoya, all automatically translated to LCTRSs by c2lctrs: `sum`: given n, implement $\sum_{i=1}^{n} i$; `fib`: compute the nth Fibonacci number; `sumfrom`: given n, m, implement $\sum_{i=n}^{m} i$; `strlen` and `strcpy`. We compared the first three to LCTRS versions of recursive reference implementations;[6] for `strlen` and `strcpy`, we used a specification as in Examples 4.26 and 4.28.[7] We also tested our own implementations of `fact` from Example 4.2 and `arrsum` from Example 4.27, along with 25 function comparisons from the literature and 12 memory-safety benchmarks from the *Competition on Software Verification* [SV-COMP 2017]. The benchmarks (also from the literature) are typically fairly small: the largest, `lit03_GS13_fig6`, has 70 lines of C code and 55 rewrite rules. We used an Intel i7-5600U CPU at 2.6GHz under Linux.

We quickly found that many of the student programs had failed to account for boundary conditions, such as empty strings or negative input. On such programs, ctrl answers NO, or MAYBE if the system cannot be proved confluent—that is, if not all variables are initialized. To limit the impact of these errors, we did a second test, where we altered the specification to account for these mistakes. The results of both tests are summarized in Figure 1.

[6]However, honesty compels us to mention that for `fib`, we used a manual translation because the one obtained from c2lctrs was impractical: where our manual translation has a rule $\mathsf{fibrec}(x) \to \mathsf{plus}(\mathsf{fibrec}(x-1), \mathsf{fibrec}(x-2))\ [x \geq 2]$, the automatic one splits the two recursive calls (recall Section 3.5). Therefore, a more sophisticated termination argument is needed, and it is harder to eliminate the recursion in the inductive process. Handling such cases in the future will likely necessitate an additional lemma generation technique.
[7]Interestingly, in `strcpy02`, the student's `strlen` solution is called as a helper function for `strcpy`.

function	YES	NO	MAYBE	time
sum	9 / 9	0 / 0	6 / 6	2.1 / 2.1
fib	4 / 10	6 / 1	3 / 2	7.6 / 5.6
sumfrom	3 / 3	1 / 0	2 / 3	1.8 / 2.1
strlen	1 / 2	0 / 0	5 / 4	4.1 / 4.0
strcpy	3 / 5	0 / 0	3 / 1	21.8 / 17.1
arrsum	1 / 1	0 / 0	0 / 0	3.9 / 3.9
fact	1 / 1	0 / 0	0 / 0	2.2 / 2.2
literature	4 / 5	3 / 2	18 / 18	4.0 / 3.9
safety	3 / 3	2 / 2	7 / 7	22.3 / 22.3
total	29 / 39	12 / 5	44 / 41	

Legend: *YES indicates that a proof was found, NO a disproof (so a conclusion $\bot$); MAYBE denotes that Ctrl found no proof or disproof, took more than 60 seconds, or failed to prove termination of the LCTRS. The* time *column lists the average runtime on YES and NO results.*

Fig. 1. Results of Ctrl in the initial test (before the slash) and with obvious mistakes fixed (after the slash).

We found five classes of recurring failures. The first class consists of cases where the function was wrong, but Ctrl could not answer NO as it could not prove confluence. This accounts for six MAYBEs in the initial test and two in the second, and could be considered an incorrect implementation. Second (six failures in either table) is the termination requirement: we need termination independent from the starting symbol, which is often not satisfied or cannot be proved by our admittedly limited termination module.

The remaining groups of failures each demonstrate a weakness of our method. The third failure occurs when generalization drops a relation between two variables, such as when x and y are both initialized to 0 and then increased by 1 in every loop iteration (with loops corresponding to tail-recursive functions); after generalizing, the information that they are equal is lost. Typically, this manifests as an EXPANSION where the nondiverging case can easily be removed before generalization but afterward gives an equation that can be disproved. This suggests a natural direction for improvement.

The fourth group includes those benchmarks where our primary generalization technique (Section 5.1) does not apply because there are no variables to generalize. This happens when both sides have non–tail-recursive functions or loops counting down rather than up. Recursion-abstraction (Section 5.2) lets us solve several benchmarks, but further lemma generation will be needed for the majority. Nonetheless, this generalization technique does allow us to handle Example 4.30, which can be challenging for existing approaches.

The final group concerns nested loops. Ctrl's strategy fails because the counters for the inner and outer loop are generalized at the same time. However, inductive proofs with Ctrl's interactive mode show that such benchmarks can be handled by our method. Thus, in future work, a more sophisticated generalization strategy would be desirable.

Demonstrative examples of these last three issues are given in Appendix D. A full evaluation page, including exact problem statements, is given at http://cl-informatik.uibk.ac.at/software/ctrl/tocl/.

7. RELATED WORK

The related work can be split into two categories: (1) the literature on rewriting induction and (2) the work on program verification and equivalence analysis.

7.1. Rewriting Induction

Our inductive theorem–proving method builds on a long literature about rewriting induction (e.g., see Bouhoula [1997], Falke and Kapur [2012], Reddy [1990], and Sakata et al. [2009]). Its core method extends existing techniques to the LCTRS formalism introduced in Kop and Nishida [2013], thus generalizing the possibilities of earlier work.

The most relevant related works are those of Falke and Kapur [2012] and Sakata et al. [2009], defining rewriting induction for different styles of constrained rewriting. Both use only *integer* functions and predicates; it is not clear how to generalize these approaches to more advanced theories. The more general setting of LCTRSs enables rewriting induction also for systems with for instance arrays, bitvectors, or real numbers. Moreover, not restricting the predicates in Σ_{theory} enables (a limited form of) quantifiers in constraints.

These advantages are enabled by subtle changes to the inference rules, particularly SIMPLIFICATION and EXPANSION. Our changes let us modify constraints of an equation and handle irregular rules with fresh variables in the constraint. This additionally enables EXPANSION steps to create such (otherwise infeasible) rules. The method requires a very different implementation from previous definitions: we need separate strategies to simplify constraints (e.g., deriving quantified statements) and, for the desired generality, must rely primarily on external solvers to manipulate constraints.

Moreover, we have introduced a completely new generalization technique, as a powerful tool for analyzing loops in particular. Nakabayashi et al. [2010] use a similar idea (abstracting the initialization values), but the execution is very different: for an equation $s \approx t\ [\varphi]$, first $s \approx t$ is adapted via templates obtained from the rules, then φ is generalized via a set of relations between positions tracked by the proof process. In our method, the constraint carries all of the information. We succeed on all examples in Nakabayashi et al. [2010], and on some where their method fails (see Appendix C; e.g., for nonnegative `n`, a `for`-loop summing up from `1` to `n` is compared to `n*(n+1)/2`).

For unconstrained systems, the literature contains several generalization methods (e.g., Kapur and Sakhanenko [2003], Kapur and Subramaniam [1996], and Urso and Kounalis [2004]). Mostly, our method in Section 5.1 is very different from these approaches. Most similar, perhaps, is the work of Kapur and Sakhanenko [2003], which also proposes a method to generalize initial values. As observed by Nakabayashi et al. [2010], this method is not sufficient for even our simplest benchmarks sum and fact, as the argument for the loop variable cannot be generalized; in contrast, our method has no problem with such variables. As discussed in Section 5.2, the recursion-abstraction technique presented there essentially lifts a technique from explicit induction [Aubin 1979] to constrained rewriting induction.

As far as we are aware, there is no other work for lemma generation of rewrite systems (or functional programs) obtained from procedural programs.

Like Giesl et al. [2007], we verify procedural programs via a transformation to a functional program, followed by an invocation of an inductive theorem prover. In an unconstrained setting, they propose an equivalence-preserving program transformation to a non–tail-recursive program to eliminate accumulator arguments. A combination of their approach with ours could be beneficial; for example, for programs with nested loops.

7.2. Automatic Program Verification and Equivalence Proving

Our goal is to (automatically) verify correctness properties of procedural programs. Fully automated verifiers for properties like (memory) safety and termination are regularly assessed at the *Competition on Software Verification* [SV-COMP 2017]. However, a comparison with these tools does not seem useful. Although we can, to some extent, tackle (memory) safety and termination, our main topic is *equivalence*, which is not studied in SV-COMP. Technically, equivalence problems can be formulated as safety problems (by self-composition [Barthe et al. 2011]: call both programs on equal inputs and assert that their results are also equal). However, none of the tools in the "recursive" category of SV-COMP 2015 could prove equivalence for our simplest (integer) example sum.

Apart from constrained rewriting, another intermediate representation for verification of imperative programs is based on (constrained) logic programs or, closely related, Horn clauses [Albert et al. 2007; Gupta et al. 2011]. It should be possible to express our contributions also in this framework, provided that constructor terms are supported.

For the setting of Example 1.1, automated grading, Vujosevic-Janicic et al. [2013] apply verification techniques like bounded model checking. Although this enables significant improvements over classic testing, there is still a nonzero risk of missing bugs due to underapproximation. Thus, it could be beneficial to add our approach to the portfolio.

For program equivalence, we discuss (fully) automated techniques for proving partial equivalence and its special case, total equivalence. Two programs P_1 and P_2 are *partially equivalent* if for the same inputs the terminating executions of P_1 and P_2 return the same value. They are *totally equivalent* if they moreover both terminate on all inputs (see Godlin and Strichman [2008] for a more extensive discussion).

This article addresses total equivalence: we require termination to analyze partial equivalence. We allow constrained equivalence queries so that only certain inputs are considered. This includes properties that cannot be checked programmatically, like the size of an array in a C program. As mentioned in Section 6.1, for nonconfluent programs P_1 and P_2, we analyze if running P_1 on the input *can* lead to the same result as P_2.

Godlin and Strichman [2008] propose a Hoare-style proof rule for partial equivalence of recursive programs (among other properties). To analyze two recursive functions f_1 and f_2, these symbols are first replaced in recursive calls in their bodies by the same *uninterpreted function symbol* f. Under this premise, it is then proved (e.g., by a bounded model checker) that the bodies of f_1 and f_2 also have equivalent results. In this sense, Godlin and Strichman [2008] also use inductive reasoning. However, our approach proves equivalence of Example 4.30 with different recursion base cases, whereas their proof rule is not applicable. Moreover, the use of uninterpreted function symbols requires that the programs must be deterministic, in contrast to our approach.

Lopes and Monteiro [2016] prove partial equivalence for programs on integers and undefined function symbols (which may arise also as abstractions of deterministic complex functions). They combine self-composition [Barthe et al. 2011], a safety-preserving transformation of undefined functions to polynomials (yielding a program on integers only), recurrence solving for loops, and a standard software model checker. However, their approach does not support *mutable* arrays, whose content can be changed during the program's execution (as in Example 4.28 for `strcpy`), in contrast to our method.

Verdoolaege et al. [2012] use widening to prove program equivalence. For validation of compiler optimizations [Necula 2000], they consider programs with (linear-)affine arithmetic and arrays. A restriction of their approach is that it does not exploit the *semantics* of arithmetic operations beyond associativity and commutativity.

Recently, *regression verification* has become an active topic of research in program equivalence proving [Godlin and Strichman 2013; Lahiri et al. 2012; Felsing et al. 2014]. As in regression testing, two programs are compared that are syntactically almost the same (e.g., different revisions of the same code base with a refactored function). Regression verification then analyzes if the two programs are semantically equivalent.

Godlin and Strichman [2013] improve modularity over their previous work [Godlin and Strichman 2008] by decomposing the proof obligations into smaller units via the call graph of the program. Hawblitzel et al. [2013] propose *mutual summaries*, relating the postconditions of two program functions. This generalizes uninterpreted functions as summaries and allows analysis of nondeterministic programs. A challenge is to find such mutual summaries automatically. Felsing et al. [2014] address this problem via Horn constraint solving to find *coupling predicates* over linear arithmetic between

program points. It would be interesting to adapt their approach for lemma generation. They also analyze total equivalence: a separate termination proof is required. The Web interface of their tool llrêve currently fails on the same example as Nakabayashi et al. [2010] (see Section 7.1). They mention an extension to arrays and heap data structures as future work.

8. DIRECTIONS FOR FUTURE WORK

This article is by no means intended as the end station for inductive theorem proving on LCTRSs, but rather as the beginning. The generalization methods that we supply are powerful together, but they do not suffice for more complicated systems or equations. A mere two methods cannot bypass the need to search for loop invariants altogether.

A natural extension would thus be both to adapt existing lemma generation techniques to the constrained setting and to adapt techniques for finding loop invariants toward the setting of rewriting induction (e.g., to suggest suitable lemmas). It might also be worthwhile to directly look at the constraints and develop advanced methods for constraint modification, which could be followed by a generalization step. Moreover, our generalization technique from Section 5.1 could be improved to generalize not only initializations with constants but also initializations with other values (e.g., copies of function parameters). This is motivated by loops that count down instead of up. Additionally, inspired by Lopes and Monteiro [2016], one might consider LCTRSs with uninterpreted functions to model functions with unknown implementations.

For a different direction, we may extend the translation from Section 3; for example, by translating structs to term data structures (see Otto et al. [2010]). The ideas from Section 3 can also be applied for languages such as Python or Java, enabling equivalence proofs between functions in different languages. This could be particularly interesting for a reference implementation in an inherently memory-safe language like F# or Java, and an efficient implementation in a language like C that has no such memory safety guarantees.

Finally, it is our hope to extend the implementation in the future, both to increase the strength of the inductive theorem proving—adding new theory and testing for more sophisticated heuristics—and to add more features to the translation from C code.

9. CONCLUSIONS

In this article, we have done two things. First, we have discussed a transformation from procedural programs to constrained term rewriting. By abstracting from the memory model underlying a particular programming language and instead encoding concepts like integers and arrays in an intuitive way, this transformation can be applied to various different (imperative) programming languages. The resulting LCTRS is close to the original program and has built-in error checking for all mistakes of interest.

Second, we have extended rewriting induction to the setting of LCTRSs. We have shown how this method can be used to prove correctness of procedural programs. The LCTRS formalism is a good analysis backend for this, since the techniques from standard rewriting can typically be extended to it, and native support for logical constraints and data types like integers and arrays is present.

We have also introduced two new techniques to generalize equations. The idea of the core method is to identify constants used as *variable initializations*, keep track of them during the proof process, and abstract from these constants when a proof attempt diverges. The LCTRS setting is instrumental in the simplicity of this method, as it boils down to dropping a (cleverly chosen) part of a constraint. The second method recognizes—and abstracts—recursive calls on semantically equivalent arguments.

In addition to the theory, we provide an implementation of these techniques. Initial results on a small database of programs from students and the literature are very promising. In future work, we aim to increase the strength of our implementations.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

APPENDIXES

A. TRANSLATING C PROGRAMS TO THE LCTRS

This appendix provides further details on the translation from C programs to LCTRSs.

A.1. Optimizing LCTRSs

After generating the LCTRS, we simplify the (left-linear) result by the following steps:

(1) *Combining unconstrained rules.* Like Falke et al. [2011], we repeat the following:
—select any unconstrained rule ρ of the form $\mathsf{u}(x_1, \ldots, x_n) \to r$ where u is not the initial symbol of a C function (like fact in Example 3.1), and u neither occurs in r nor in the left-hand side of any other rule; the repetition stops if no such ρ exists;
—rewrite all right-hand sides with ρ;
—remove both the rule ρ and the symbol u.
This process does not substantially alter the multistep reduction relation $\to^*$, as the only symbols removed are those that we think of as "intermediate" symbols.

(2) *Combining constrained rules.* If there are distinct rules $\ell \to r\ [\varphi]$ and $\ell \to r\ [\psi]$ (modulo renaming), these are combined into $\ell \to r\ [\varphi \vee \psi]$. Given rules $\ell \to \mathsf{u}(s_1, \ldots, s_m)\ [\varphi]$ and $\mathsf{u}(x_1, \ldots, x_m) \to r_i\ [\psi_i]$ for $1 \leq i \leq n$ with all x_j variables, we may replace them by $\ell \to r_i[x_1 := s_1, \ldots, x_m := s_m]\ [\varphi \wedge \psi_i[x_1 := s_1, \ldots, x_m := s_m]]$ for $1 \leq i \leq n$ if:
—u is not the initial symbol of a function and does not occur in any other rule, or ℓ;
—the terms s_j do not contain defined symbols (as then we might remove a nonterminating subterm, which would impact the multistep reduction relation).

(3) *Removing unused arguments.* For all function symbols and all of their argument positions, we mark whether the position is "used":
—all argument(s) of every return_f and initial symbols (e.g., fact) are used;
—for other symbols u_i of arity n and every $1 \leq j \leq n$: if there is a rule $\mathsf{u}_i(\ell_1, \ldots, \ell_n) \to r\ [\varphi]$ where ℓ_j is not a variable (which can arise for instance with the transformation in Section 3.5) or occurs in φ, then argument j is used in u_i;
—for all rules $\mathsf{u}_i(\ell_1, \ldots, \ell_n) \to r\ [\varphi]$ and $1 \leq j \leq n$, argument j is used in u_i if ℓ_j is a variable occurring at a *used position* in r; here, a position p is used in s if either $p = \epsilon$ or $p = i \cdot p',\ s = f(\vec{t})$, argument i is used in f and position p' is used in t_i.
The last, recursive, step essentially calculates a fixpoint; in summary, an argument position is *used* if it is possible to reduce to a term where we actually need the subterm at that position as part of a constraint or the function's return value. When a variable is not used in any later statement, we will avoid carrying it along.

(4) *Simplifying constraints.* Constraints may be brought into an equivalent form, for example by removing duplicate clauses or by replacing a clause eg $(x > y)$ with $x \leq y$. Here, φ is "equivalent" to ψ in a rule $\ell \to r\ [\varphi]$ if $\forall\vec{x}(\exists\vec{y}(\varphi) \leftrightarrow \exists\vec{z}(\psi))$ holds, where $Var(\ell) \cup Var(r) = \{\vec{x}\}$, $Var(\varphi) \setminus \{\vec{x}\} = \{\vec{y}\}$, and $Var(\psi) \setminus \{\vec{x}\} = \{\vec{z}\}$ (much like the observation on $\sim$ below Definition 2.14). We typically only remove negations and unused variables.

Example A.1. As an example, let us consider the simplification of a toy function.

```
int f(int x) {
  int y,z;
  if (x < 0) return 0;
  z = 0;
  while (x > 0) {
    x--;
    z += x;
  }
  y = z + x;
  return y;
}
```

$$
\begin{array}{rcll}
\mathsf{f}(x) & \to & \mathsf{u}_1(x,y,z) & \\
\mathsf{u}_1(x,y,z) & \to & \mathsf{u}_2(x,y,z) & [x < 0] \\
\mathsf{u}_1(x,y,z) & \to & \mathsf{u}_3(x,y,z) & [\neg(x < 0)] \\
\mathsf{u}_2(x,y,z) & \to & \mathsf{return}_\mathsf{f}(0) & \\
\mathsf{u}_3(x,y,z) & \to & \mathsf{u}_4(x,y,0) & \\
\mathsf{u}_4(x,y,z) & \to & \mathsf{u}_5(x,y,z) & [x > 0] \\
\mathsf{u}_4(x,y,z) & \to & \mathsf{u}_7(x,y,z) & [\neg(x > 0)] \\
\mathsf{u}_5(x,y,z) & \to & \mathsf{u}_6(x-1,y,z) & \\
\mathsf{u}_6(x,y,z) & \to & \mathsf{u}_4(x,y,z+x) & \\
\mathsf{u}_7(x,y,z) & \to & \mathsf{u}_8(x,z+x,z) & \\
\mathsf{u}_8(x,y,z) & \to & \mathsf{return}_\mathsf{f}(y) &
\end{array}
$$

The rule $\mathsf{f}(x) \to \mathsf{u}_1(x,y,z)$ has unconstrained variables y and z in the right-hand side that do not occur on the left. A step with this rule instantiates y and z by arbitrary type-correct values. This reflects that in the C program, the variables y and z are at first not initialized and may contain an arbitrary value (depending on the compiler). In the simplified version, this does not occur; consider the remainder obtained from combining rules.

$$
\begin{array}{rcll}
\mathsf{f}(x) & \to & \mathsf{return}_\mathsf{f}(0) & [x < 0] \\
\mathsf{f}(x) & \to & \mathsf{u}_4(x,y,0) & [\neg(x < 0)] \\
\mathsf{u}_4(x,y,z) & \to & \mathsf{u}_4(x-1,y,z+x-1) & [x > 0] \\
\mathsf{u}_4(x,y,z) & \to & \mathsf{return}_\mathsf{f}(z+x) & [\neg(x > 0)]
\end{array}
$$

Now, the first and third arguments of u_4 are used (in the constraint and return value), but the second is not: it is merely passed along in the recursive call. Removing this variable and simplifying the constraints, we obtain the following.

$$
\begin{array}{rcll}
\mathsf{f}(x) & \to & \mathsf{return}_\mathsf{f}(0) & [x < 0] \\
\mathsf{f}(x) & \to & \mathsf{u}_4(x,0) & [x \geq 0] \\
\mathsf{u}_4(x,z) & \to & \mathsf{u}_4(x-1,z+x-1) & [x > 0] \\
\mathsf{u}_4(x,z) & \to & \mathsf{return}_\mathsf{f}(z+x) & [x \leq 0]
\end{array}
$$

This system is *orthogonal* in the sense of Kop and Nishida [2013] and thus *confluent*, which is beneficial for analysis. The original LCTRS was also confluent, but this was harder to see.

Correctness relies on the fact that the LCTRSs created using the transformation described in Section 3 are "well behaved"; most importantly, all rules are left linear.

A.2. Translating C Programs with Explicit Pointers

As observed at the end of Section 3.6, the simple translation explored there has both up- and downsides. On the one hand, by abstracting from the memory model, we can simplify analysis. On the other hand, there are certain programs that we cannot handle.

For C programs with dynamically allocated arrays and/or explicit pointer use, we consider the memory model from the C standard. Declaring or allocating an array selects an amount of currently unused space in memory and designates it for use by the given array. The allocated space is not guaranteed to be at a given position in memory relative to existing declarations; when an array is indexed out of its declared bounds, the resulting behavior is undefined—so this can safely be considered an error (see paragraph 6.5.6:9 at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf).

We will think of a program's memory as a set of *blocks*, each block corresponding to a sequence of values. A pointer then becomes a location in such a block. In an LCTRS, we

will model this using a "global memory" variable, which lists the blocks as a sequence of arrays; a pointer is a pair of integers, selecting a memory block and its offset.

Limiting interest to programs on (dynamically allocated) integer or char arrays, we will use a memory variable of sort array(array(int)), which represents a sequence of integer arrays (i.e., $(\mathbb{Z}^*)^*$); the default value $0_{\mathsf{array(int)}}$ is the empty sequence $\langle\rangle \in \mathbb{Z}^*$. We use a theory signature with the array symbols introduced in Section 3.6 along with

—allocate : [array(array(int)) × array(int)] ⇒ array(array(int)), where $\mathcal{J}_{\mathsf{allocate}}(\langle a_0, \ldots, a_k\rangle, b) = \langle a_0, \ldots, a_k, b\rangle$—that is, allocate$(mem, arr)$ adds the new sequence arr to the memory;
—free : [array(array(int)) × int] ⇒ array(array(int)), where $\mathcal{J}_{\mathsf{free}}(\langle a_0, \ldots, a_k\rangle, n) = \langle a_0, \ldots, a_{n-1}, \langle\rangle, a_{n+1}, \ldots, a_k\rangle$ if $0 \le n \le k$ and $\langle a_0, \ldots, a_k\rangle$ otherwise—that is, the memory block indexed by n is considered empty, and any further attempt to address a location in that memory block should be considered an error.

A pointer is represented by a pair (b, o) of a block index and an offset within that block. The NULL-pointer is represented by $(-1, 0)$.

Example A.2. Consider the following example C++ function.

```
int *create(int k) {
  int *a = new int[k];
  int *b = a + 1;
  for (int i = 0; i < k; i += 2) b[i] = 42;
  return a;
}
```

Now, `a` and `b` share memory, and new memory is allocated. We might encode this as follows.

$$\begin{aligned}
\mathsf{create}(mem, k) &\to \mathsf{u}(\mathsf{allocate}(mem, x), k, \mathsf{size}(mem), 0) && [\mathsf{size}(x) = k]\\
\mathsf{u}(mem, k, ai, ao) &\to \mathsf{v}(mem, k, ai, ao, ai, ao + 1, 0)\\
\mathsf{v}(mem, k, ai, ao, bi, bo, i) &\to \mathsf{w}(mem, k, ai, ao, bi, bo, i) && [i < k]\\
\mathsf{v}(mem, k, ai, ao, bi, bo, i) &\to \mathsf{return}(mem, ai, ao) && [i \ge k]\\
\mathsf{w}(mem, k, ai, ao, bi, bo, i) &\to \mathsf{error} && [bo + i < 0 \vee bo + i \ge \mathsf{size}(\mathsf{select}(mem, bi))]\\
\mathsf{w}(mem, k, ai, ao, bi, bo, i) &\to \mathsf{v}(\mathsf{store}(mem, bi, \mathsf{store}(\mathsf{select}(mem, bi), bo + i, 42)), k,\\
&\qquad ai, ao, bi, bo, i + 2) && [0 \le bo + i < \mathsf{size}(\mathsf{select}(mem, bi))]
\end{aligned}$$

(For clarity, we omit the optimization step that combines the first two rules, and the one that combines the third with the last two.)

Consider how this example is executed, starting from empty memory. We will use $\langle\cdot\rangle$ to refer to specific arrays of type array(array(int)) and $[\cdot]$ for arrays of type array(int):

(1) We call create$(\langle\rangle, 2)$, representing a function call when no arrays have been allocated.
(2) By the first rule, we get u(allocate$(\langle\rangle, x)$, 2, size$(\langle\rangle)$, 0), where x is a *random* array. All we know is that it has size 2—this rule uses irregularity to represent the randomness involved in an allocation. Thus, assume that the sequence $[-4, 9]$ is chosen. Using calculation steps to evaluate allocate and size, we get u$(\langle[-4, 9]\rangle, 2, 0, 0)$. Here, the pair $(0, 0)$ represents the array a: the first block in memory, read from the start (offset 0).
(3) Then by the second rule (and a calculation), we reduce to v$(\langle[-4, 9]\rangle, 2, 0, 0, 0, 1, 0)$. The new pair $(0, 1)$ represents b: the same memory block as a but with offset 1. This location points to the sequence [9]. The final 0 is the index for the loop counter i.
(4) Entering the loop (as indeed $0 < 2$), we reduce to w$(\langle[-4, 9]\rangle, 2, 0, 0, 0, 1, 0)$.
(5) Here, we do an array store: `b[i] = 42;`. The LCTRS first tests whether `b[i]` corresponds to a position in allocated memory and reduces to an error state if

not. This is done by selecting the corresponding block from *mem*, then testing whether the offset for `b` and `i` together exceed the block's bounds. We succeed, as $0 \le 0 + 1 < \mathsf{size}(\mathsf{select}(\langle[-4, 9]\rangle, 0)) \Leftrightarrow 0 \le 1 < \mathsf{size}([-4, 9]) \Leftrightarrow 0 \le 1 < 2$.

(6) Thus, the update is done, and we reduce to
$\mathsf{v}(\mathsf{store}(\langle[-4, 9]\rangle, 0, \mathsf{store}(\mathsf{select}(\langle[-4, 9]\rangle, 0), 1 + 0, 42)), 2, 0, 0, 0, 1, 0 + 2)$
$\to^*_{\mathtt{calc}} \mathsf{v}(\mathsf{store}(\langle[-4, 9]\rangle, 0, \mathsf{store}([-4, 9], 1, 42)), 2, 0, 0, 0, 1, 2)$
$\to^*_{\mathtt{calc}} \mathsf{v}(\mathsf{store}(\langle[-4, 9]\rangle, 0, [-4, 42]), 2, 0, 0, 0, 1, 2)$
$\to_{\mathtt{calc}} \mathsf{v}(\langle[-4, 42]\rangle, 2, 0, 0, 0, 1, 2)$.
Thus, we retrieve the space for `b` from memory (getting the full block $[-4, 9]$), update the position corresponding to `b[0]` (which is the same as `a[1]`), get $[-4, 42]$, and store the result into the corresponding position in memory. Then we carry on with $i + 2$.

(7) Since $2 \ge 2$, we reduce to $\mathsf{return}(\langle[-4, 42]\rangle, 0, 0)$, returning the dynamic array $[-4, 42]$.

Note that in step 5, we do not test whether `b` corresponds to currently allocated memory. This is safe because if `b` is the NULL pointer or corresponds to previously freed memory, then $\mathsf{select}(mem, bi)$ is $\langle\rangle$, and any indexing in this array will cause an error regardless. Note also that this function gives a nonerror result only for *even k*.

Although Example A.2 considers only integer arrays, we could also handle programs with dynamically allocated arrays of varying types. In this case, we would simply use multiple memory variables with different type declarations.

B. CORRECTNESS PROOF

In this appendix, we give the full correctness proof, which was only sketched in Section 4.4.

First, we prove Lemma 4.31, reformulated as follows.

LEMMA 4.31. *The following statements are equivalent:*

—*all equations in* $\mathcal{E}$ *are inductive theorems;*
— $\leftrightarrow_{\mathcal{E}} \subseteq \leftrightarrow^*_{\mathcal{R}}$ *on ground terms (so if* s, t *are ground and* $s \leftrightarrow_{\mathcal{E}} t$*, then also* $s \leftrightarrow^*_{\mathcal{R}} t$*).*

PROOF. Suppose that $\leftrightarrow_{\mathcal{E}} \subseteq \leftrightarrow^*_{\mathcal{R}}$ on ground terms. If $s \approx t\ [\varphi] \in \mathcal{E}$ and the ground constructor substitution γ respects this equation, then $s\gamma$ and $t\gamma$ are ground (since, by definition of "respects" (Definition 4.3), $Var(s) \cup Var(t) \subseteq Dom(\gamma)$). Since obviously $s\gamma \leftrightarrow_{\mathcal{E}} t\gamma$ (with empty C), by assumption $s\gamma \leftrightarrow^*_{\mathcal{R}} t\gamma$. Thus, $s \approx t\ [\varphi]$ is an inductive theorem.

Suppose that all equations in $\mathcal{E}$ are inductive theorems, and $u \leftrightarrow_{\mathcal{E}} w$ for ground u, w; we must see that $u \leftrightarrow^*_{\mathcal{R}} t$. We have $u = C[s\gamma]$ and $w = C[t\gamma]$ for some $s \approx t\ [\varphi] \in \mathcal{E}$ and substitution γ that respects φ and maps all variables in s, t to ground terms. Let δ be a substitution such that each $\delta(x)$ is a normal form of $\gamma(x)$; by termination of $\mathcal{R}$, such a δ exists, and by quasi-reductivity, it is a ground constructor substitution. As values cannot be reduced, also δ respects φ. Therefore, $s\delta \leftrightarrow_{\mathcal{E}} t\delta$, which implies that $s\delta \leftrightarrow^*_{\mathcal{R}} t\delta$. We conclude that $C[s\gamma] \leftrightarrow^*_{\mathcal{R}} C[s\delta] \leftrightarrow^*_{\mathcal{R}} C[t\delta] \leftrightarrow^*_{\mathcal{R}} C[t\gamma]$, giving the desired result. □

Recall also the following key lemma (whose proof has been given in the main text).

LEMMA 4.32 [SAKATA ET AL. 2009]. *Let* $\to_1$ *and* $\to_2$ *be binary relations over some set A. Then* $\leftrightarrow^*_1 = \leftrightarrow^*_2$ *if all of the following hold:*

— $\to_1 \subseteq \to_2$,
— $\to_2$ *is well founded, and*
— $\to_2 \subseteq (\to_1 \cdot \to^*_2 \cdot \leftrightarrow^*_1 \cdot \leftarrow^*_2)$.

Lemma 4.33 in the main text is the combination of the following Lemmas B.1 through B.4.

LEMMA B.1. *Let s, t be terms, φ a constraint, and p a position of s such that $s_{|p}$ has the form $f(s_1, \ldots, s_n)$ with f a defined symbol and all s_i constructor terms. Suppose that the variables in s, t, φ are distinct from those in $\mathcal{R}$. Then*

(1) *For any ground constructor substitution γ that respects $s \approx t\ [\varphi]$ and any choice of $Expd(s \approx t\ [\varphi], p)$, we have*

$$s\gamma \ \left(\to_{\mathcal{R},p} \cdot \leftrightarrow_{Expd(s\approx t\ [\varphi],p)}\right)\ t\gamma.$$

Here, $\to_{\mathcal{R},p}$ indicates a reduction at position p with a rule in $\mathcal{R} \cup \mathcal{R}_{\mathtt{calc}}$.

(2) *For any $s' \to t'\ [\varphi']$ in any choice of $Expd(s \approx t\ [\varphi], p)$ and any ground constructor substitution δ that respects $s' \approx t'\ [\varphi']$, we have*

$$s'\delta \ \left(\leftarrow_{\mathcal{R},p} \cdot \leftrightarrow_{\{s \approx t\ [\varphi]\}}\right)\ t'\delta.$$

PROOF. $s\gamma_{|p} = s_{|p}\gamma = f(s_1\gamma, \ldots, s_n\gamma)$, where all $s_i\gamma$ are ground constructor terms. Since f is defined, $f(\overrightarrow{s\,\gamma})$ reduces by quasi-reductivity, which can only be a root reduction. Thus, $s\gamma = (s\gamma)[\ell\delta]_p$ for some rule $\ell \to r\ [\psi]$ and substitution δ that respects ψ. Since the rule variables are distinct from the ones in the equation, we can assume that δ is an extension of γ, so $s\gamma = s[\ell]_p\delta$. Clearly, both $\varphi\delta$ and $\psi\delta$ evaluate to $\top$.

As δ unifies $s_{|p}$ and ℓ, there is a most general unifier η, so $s_{|p}\eta = \ell\eta$ and we can write $\delta = \delta' \circ \eta$ for some δ'. Now, by definition of constrained term reduction, any choice of $Expd(s \approx t\ [\varphi], p)$ has an element $s' \approx t'\ [\varphi']$ where we can write (for suitable u, η' etc.) the following:

$$\begin{array}{rrcl} & s\eta[\ell\eta]_p & \approx & t\eta\ \ [\varphi\eta \wedge \psi\eta] \\ \sim & u[\ell\eta']_p & \approx & t''\ \ [\varphi''] \\ \to_{\ell\to r[\psi],1\cdot p} & u[r\eta']_p & \approx & t''\ \ [\varphi''] \\ \sim & s' & \approx & t'\ \ [\varphi']. \end{array}$$

Consider the "term" $s\delta \approx t\delta$. This is an instance of the first constrained term in this reduction, so by Theorem 2.19, this term reduces at position $1 \cdot p$ to $s'\delta'' \approx t'\delta''$ for some substitution δ'' that respects φ'. As the reduction happens inside $s\delta$, we see that $t\delta = t'\delta''$. Thus, $s\gamma = s\delta \to_{\mathcal{R}} s'\delta'' \leftrightarrow_{Expd(s\approx t\ [\varphi],p)} t'\delta'' = t\delta = t\gamma$.

As for the second part, note that by definition of $Expd$ there are a substitution γ and constraint ψ such that the constrained term $s\gamma \approx t\gamma\ [\varphi\gamma \wedge \psi\gamma]$ reduces to $s' \approx t'\ [\varphi']$ at position $1 \cdot p$. By Theorem 2.20, we find a substitution η that respects $\varphi\gamma \wedge \psi\gamma$ such that $s\gamma\eta \approx t\gamma\eta \to_{\mathcal{R}} s'\delta \approx t'\delta$ at position $1 \cdot p$. Since the reduction takes place in the left part of $\approx$, we have $t\gamma\eta = t'\delta$ and $s\gamma\eta \to_{\mathcal{R}} s'\delta$. We are done if also $s\gamma\eta \leftrightarrow_{s\approx t\ [\varphi]} t\gamma\eta$, which indeed holds because $\eta \circ \gamma$ respects φ (as $(\varphi\gamma \wedge \psi\gamma)\eta$ implies $\varphi\gamma\eta$). □

LEMMA B.2. *Suppose that $(\mathcal{E}, \mathcal{H}, flag) \vdash_{\mathtt{ri}} (\mathcal{E}', \mathcal{H}', flag')$ by any inference rule other than* COMPLETENESS. *Then*

$$\leftarrow\!\!\!\!\twoheadrightarrow_{\mathcal{E}} \subseteq \left(\to^*_{\mathcal{R}\cup\mathcal{H}'} \cdot \ \leftarrow\!\!\!\!\twoheadrightarrow_{\mathcal{E}'} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'}\right)$$

on ground terms.

Here, $\leftarrow\!\!\!\!\twoheadrightarrow_{\mathcal{E}'}$ denotes a *parallel* application of zero or more $\leftrightarrow_{\mathcal{E}'}$ steps.

PROOF. It suffices to show that $\leftrightarrow_{\mathcal{E}\setminus\mathcal{E}'} \subseteq (\to^*_{\mathcal{R}\cup\mathcal{H}'} \cdot \ \leftarrow\!\!\!\!\twoheadrightarrow_{\mathcal{E}'} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'})$ on ground terms: if $C[u_1, \ldots, u_n] \ \leftarrow\!\!\!\!\twoheadrightarrow_{\mathcal{E}} C[v_1, \ldots, v_n]$ because each $u_i \leftrightarrow_{\rho_i} v_i$ for some $\rho_i \in \mathcal{E}$, then this gives $u_i \to^*_{\mathcal{R}\cup\mathcal{H}'} u_i' \ \leftarrow\!\!\!\!\twoheadrightarrow_{\mathcal{E}'} v_i' \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'} v_i$ if $\rho_i \notin \mathcal{E}'$ and $u_i = u_i' \ \leftarrow\!\!\!\!\twoheadrightarrow_{\mathcal{E}'} v_i' = v_i$ if $\rho_i \in \mathcal{E}'$, so (sequentializing parallel steps) $C[u_1, \ldots, u_n] \to^*_{\mathcal{R}\cup\mathcal{H}'} C[u_1', \ldots, u_n'] \ \leftarrow\!\!\!\!\twoheadrightarrow_{\mathcal{E}'} C[v_1', \ldots, v_n'] \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'} C[v_1, \ldots, v_n]$ as desired. For all inference rules (except COMPLETENESS), either $\mathcal{E} \setminus \mathcal{E}' = \emptyset$ or we can write $\mathcal{E} \setminus \mathcal{E}' = \{s \simeq t\ [\varphi]\}$. Consider which inference rule is applied for $\vdash_{\mathtt{ri}}$:

—(SIMPLIFICATION). Suppose that $s \simeq t\ [\varphi]$ is replaced by $u \approx t\ [\psi]$, where $s \approx t\ [\varphi] \to_{\mathcal{R}\cup\mathcal{H}} u \approx t\ [\psi]$. Let $C[s\gamma] \leftrightarrow_{\{s\simeq t\ [\varphi]\}} C[t\gamma]$, where γ is a substitution that respects φ. It follows from Theorem 2.19 that $s\gamma \approx t\gamma \to_{\mathcal{R}\cup\mathcal{H}} u\delta \approx t\delta$, where δ is a substitution that respects ψ, and thus, as $\approx$ is a constructor, $C[s\gamma] \to_{\mathcal{R}\cup\mathcal{H}} C[u\delta]$ and $t\gamma = t\delta$. Then $C[u\delta] \leftrightarrow_{\{u\approx t\ [\psi]\}} C[t\delta] = C[t\gamma]$, and we have $C[s\gamma] \to_{\mathcal{R}\cup\mathcal{H}} \cdot \leftrightarrow_{\mathcal{E}'} C[t\gamma]$. Symmetrically, if $C[t\gamma] \leftrightarrow_{\{s\simeq t\ [\varphi]\}} C[s\gamma]$, then $C[t\gamma] \leftrightarrow_{\mathcal{E}'} \cdot \leftarrow_{\mathcal{R}\cup\mathcal{H}} C[s\gamma]$. Thus, $\leftrightarrow_{s\simeq t\ [\varphi]} \subseteq (\to^*_{\mathcal{R}\cup\mathcal{H}} \cdot \leftrightarrow_{\mathcal{E}'} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}})$. This suffices because in this case $\mathcal{H} = \mathcal{H}'$.

—(DELETION). In the case that $s = t$, the relation $\leftrightarrow_{\mathcal{E}\setminus\mathcal{E}'}$ is the identity. Otherwise, φ is unsatisfiable, so $s \simeq t\ [\varphi]$ is never used (i.e., $\leftrightarrow_{\mathcal{E}\setminus\mathcal{E}'} = \emptyset$).

—(EXPANSION). Suppose that $C[s\gamma] \leftrightarrow_{s\simeq t\ [\varphi]} C[t\gamma]$, where γ respects $s \simeq t\ [\varphi]$; as we only consider ground terms, $\gamma(x)$ is ground for all variables in its domain. Noting that by quasi-reductivity and termination every ground term reduces to a ground constructor term, let δ be a substitution where for each $x \in Dom(\gamma)$, $\delta(x)$ is a constructor term such that $\gamma(x) \to^*_{\mathcal{R}} \delta(x)$. Then it follows from Lemma B.1 that $C[s\gamma] \to^*_{\mathcal{R}} C[s\delta]\ (\to_{\mathcal{R}} \cdot \leftrightarrow_{\mathcal{E}'}) C[t\delta] \leftarrow^*_{\mathcal{R}} C[t\gamma]$. The situation where $C[t\gamma] \leftrightarrow_{s\simeq t\ [\varphi]} C[s\gamma]$ is symmetric.

—(EQ-DELETION). Let $s = C[s_1, \ldots, s_n]$ and $t = C[t_1, \ldots, t_n]$, where $s_1, t_1, \ldots, s_n, t_n \in \mathcal{T}erms(\Sigma_{theory}, Var(\varphi))$. Any ground substitution γ that respects φ, and whose domain contains all variables in the terms s_i and t_i, must map these variables to values. Therefore, $s_i\gamma \to^*_{\mathtt{calc}} v_i$ and $t_i\gamma \to^*_{\mathtt{calc}} w_i$, where v_i is the value of $s_i\gamma$ and w_i is the value of $t_i\gamma$. Now, suppose that $q \leftrightarrow_{s\simeq t[\varphi]} u$ for ground q, u. Then (a) $q = D[C[s_1, \ldots, s_n]]\gamma$ and $u = D[C[t_1, \ldots, t_n]]\gamma$ for some ground γ that respects φ, or (b) $u = D[C[\overrightarrow{t}]]\gamma$ and $q = D[C[\overrightarrow{s}]]\gamma$. In case (a), $q \to^*_{\mathcal{R}} D\gamma[C\gamma[v_1, \ldots, v_n]]$ and $u \to^*_{\mathcal{R}} D\gamma[C\gamma[w_1, \ldots, w_m]]$. If each $v_i = w_i$, then clearly $q \to^*_{\mathcal{R}} \cdot \leftarrow^*_{\mathcal{R}} u$. Otherwise, $(\varphi \wedge \neg(s_1 = t_1 \wedge \cdots \wedge s_n = t_n))\gamma$ is valid, so we easily get the desired $q \to^*_{\mathcal{R}} \cdot \leftrightarrow_{\mathcal{E}'} \cdot \leftarrow^*_{\mathcal{R}} u$. Case (b) is symmetric.

—(DISPROVE) In this case, we do not have $(\mathcal{E}, \mathcal{H}, b) \vdash_{\mathtt{ri}} (\mathcal{E}', \mathcal{H}', b')$.

—(CONSTRUCTOR) Let $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$, and suppose that $C[s\gamma] \leftrightarrow_{\{s\approx t\ [\varphi]\}} C[f(t\gamma)]$, where γ is a substitution that respects φ. Since $\mathcal{E}'$ contains all equations $s_i \approx t_i\ [\varphi]$, we have $C[s\gamma] = C[f(s_1\gamma, \ldots, s_n\gamma)] \leftarrow\!\!\!\!\!\!\rightarrow_{\mathcal{E}'} C[f(t_1\gamma, \ldots, t_n\gamma)] = C[t\gamma]$.

—(POSTULATE) $\mathcal{E} \setminus \mathcal{E}' = \emptyset$, so there is nothing to prove!

—(GENERALIZATION) Suppose that $s \approx t\ [\varphi]$ is replaced by $s' \approx t'\ [\psi]$. Suppose that $C[s\gamma] \leftrightarrow_{\{s\approx t\ [\varphi]\}} C[t\gamma]$ for some substitution γ that respects φ. Then there exists a substitution δ that respects ψ such that $C[s\gamma] = C[s'\delta] \leftrightarrow_{\{s'\approx t'[\varphi]\}} C[t'\delta] = C[t\gamma]$. □

LEMMA B.3. *Suppose that* $(\mathcal{E}, \mathcal{H}, flag) \vdash_{\mathtt{ri}} (\mathcal{E}', \mathcal{H}', flag')$ *by any inference rule other than* COMPLETENESS. *If*

$$\to_{\mathcal{R}\cup\mathcal{H}} \subseteq \left(\to_{\mathcal{R}} \cdot \to^*_{\mathcal{R}\cup\mathcal{H}} \cdot \ \leftarrow\!\!\!\!\!\!\rightarrow_{\mathcal{E}} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}}\right)$$

on ground terms, then

$$\to_{\mathcal{R}\cup\mathcal{H}'} \subseteq \left(\to_{\mathcal{R}} \cdot \to^*_{\mathcal{R}\cup\mathcal{H}'} \cdot \ \leftarrow\!\!\!\!\!\!\rightarrow_{\mathcal{E}'} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'}\right)$$

on ground terms.

PROOF. It suffices to consider the case that EXPANSION is applied (for the other cases, we use Lemma B.2). Suppose that $s \to_{\mathcal{H}'\setminus\mathcal{H}} t$. Using that, by quasi-reductivity and termination, every ground term reduces to a ground constructor term, it follows from Lemma B.1 that there exist ground constructor terms s', t' such that $s \to^*_{\mathcal{R}} s'\ (\to_{\mathcal{R}} \cdot \leftrightarrow_{\mathcal{E}'}) t' \leftarrow^*_{\mathcal{R}} t$, and hence

$$s\ \left(\to_{\mathcal{R}} \cdot \to^*_{\mathcal{R}\cup\mathcal{H}'} \cdot \ \leftarrow\!\!\!\!\!\!\rightarrow_{\mathcal{E}'} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'}\right)\ t. \qquad \square$$

LEMMA B.4. *Suppose that* $(\mathcal{E}, \mathcal{H}, flag) \vdash_{\mathtt{ri}} \cdots \vdash_{\mathtt{ri}} (\mathcal{E}', \mathcal{H}', flag')$. *Then*

(1) $\leftarrow\!\!\!\!\!\!\rightarrow_{\mathcal{E}} \subseteq (\to^*_{\mathcal{R}\cup\mathcal{H}'} \cdot \ \leftarrow\!\!\!\!\!\!\rightarrow_{\mathcal{E}'} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'})$ *on ground terms;*

(2) *if* $\to_{\mathcal{R}\cup\mathcal{H}} \subseteq (\to_{\mathcal{R}} \cdot \to^*_{\mathcal{R}\cup\mathcal{H}} \cdot \; \leftarrow\!\Vdash\!\to_{\mathcal{E}} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}})$ *on ground terms, then*

$$\to_{\mathcal{R}\cup\mathcal{H}'} \subseteq \left(\to_{\mathcal{R}} \cdot \to^*_{\mathcal{R}\cup\mathcal{H}'} \cdot \; \leftarrow\!\Vdash\!\to_{\mathcal{E}'} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'}\right)$$

on ground terms; and

(3) *if* $\mathcal{R}\cup\mathcal{H}$ *is terminating, then so is* $\mathcal{R}\cup\mathcal{H}'$.

Proof. In the following, we consider relations limited to ground terms only. We prove the statements by induction on the number of $\vdash_{\mathtt{ri}}$-steps, where steps in the premise of a Completeness step are also counted. The base case is evident, so suppose that $(\mathcal{E},\mathcal{H},\mathit{flag}) \vdash_{\mathtt{ri}} (\mathcal{E}_1,\mathcal{H}_1,\mathit{flag}_1) \vdash^*_{\mathtt{ri}} (\mathcal{E}',\mathcal{H}',\mathit{flag}')$:

(1) If the first step uses inference rule Completeness, then $(\mathcal{E},\mathcal{H},\mathit{flag}) \vdash^*_{\mathtt{ri}} (\mathcal{E}_1,\mathcal{H}_1,\text{INCOMPLETE})$ in fewer steps, so by the induction hypothesis,

$$\leftarrow\!\Vdash\!\to_{\mathcal{E}} \subseteq \left(\to^*_{\mathcal{R}\cup\mathcal{H}_1} \cdot \; \leftarrow\!\Vdash\!\to_{\mathcal{E}_1} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}_1}\right).$$

If the first step uses another inference rule, this same property follows from Lemma B.2. By the induction hypothesis, we have

$$\leftarrow\!\Vdash\!\to_{\mathcal{E}_1} \subseteq \left(\to^*_{\mathcal{R}\cup\mathcal{H}'} \cdot \; \leftarrow\!\Vdash\!\to_{\mathcal{E}'} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'}\right).$$

It follows from $\mathcal{H}_1 \subseteq \mathcal{H}'$ that

$$\leftarrow\!\Vdash\!\to_{\mathcal{E}} \subseteq \left(\to^*_{\mathcal{R}\cup\mathcal{H}'} \cdot \; \leftarrow\!\Vdash\!\to_{\mathcal{E}_1} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'}\right).$$

By replacing $\leftarrow\!\Vdash\!\to_{\mathcal{E}_1}$ with $(\to^*_{\mathcal{R}\cup\mathcal{H}'} \cdot \; \leftarrow\!\Vdash\!\to_{\mathcal{E}'} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'})$, we thus obtain

$$\leftarrow\!\Vdash\!\to_{\mathcal{E}} \; \subseteq \left(\to^*_{\mathcal{R}\cup\mathcal{H}'} \cdot \to^*_{\mathcal{R}\cup\mathcal{H}'} \cdot \; \leftarrow\!\Vdash\!\to_{\mathcal{E}'} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'} \cdot\cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'}\right).$$

(2) Assume that $\to_{\mathcal{R}\cup\mathcal{H}} \subseteq (\to_{\mathcal{R}} \cdot \to^*_{\mathcal{R}\cup\mathcal{H}} \cdot \; \leftarrow\!\Vdash\!\to_{\mathcal{E}} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}})$. By the induction hypothesis (in case of Completeness) or Lemma B.3 (otherwise),

$$\to_{\mathcal{R}\cup\mathcal{H}_1} \subseteq \left(\to_{\mathcal{R}} \cdot \to^*_{\mathcal{R}\cup\mathcal{H}_1} \cdot \; \leftarrow\!\Vdash\!\to_{\mathcal{E}_1} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}_1}\right).$$

We complete by the induction hypothesis on $(\mathcal{E}_1,\mathcal{H}_1,\mathit{flag}_1) \vdash^*_{\mathtt{ri}} (\mathcal{E}',\mathcal{H}',\mathit{flag}')$.

(3) Trivial with the induction hypothesis, with the first step using either the induction hypothesis again (in case of Completeness), the definition of Expansion, or the observation that other inference rules do not alter $\mathcal{H}$. □

Thus, we obtain Lemma 4.33 or, equivalently, Lemma B.5, as the first part of Theorem 4.4.

Lemma B.5. *If* $(\mathcal{E},\emptyset,\mathit{flag}) \vdash_{\mathtt{ri}} \cdots \vdash_{\mathtt{ri}} (\emptyset,\mathcal{H},\mathit{flag}')$*, then every equation in* $\mathcal{E}$ *is an inductive theorem of* $\mathcal{R}$.

Proof. It is clear that $\to_{\mathcal{R}} \subseteq \to_{\mathcal{R}\cup\mathcal{H}}$. It follows from Lemma B.4 that

— $\leftrightarrow_{\mathcal{E}} \subseteq \leftarrow\!\Vdash\!\to_{\mathcal{E}} \; \subseteq \to^*_{\mathcal{R}\cup\mathcal{H}} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}}$ on ground terms,
— $\to_{\mathcal{R}\cup\mathcal{H}} \subseteq \to_{\mathcal{R}} \cdot \to^*_{\mathcal{R}\cup\mathcal{H}} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}}$ on ground terms, and
— $\mathcal{R}\cup\mathcal{H}$ is terminating.

By Lemma 4.32 (as equality is included in $\leftrightarrow^*_{\mathcal{R}}$), we find that $\leftrightarrow^*_{\mathcal{R}} = \leftrightarrow^*_{\mathcal{R}\cup\mathcal{H}}$, and hence $\leftrightarrow_{\mathcal{E}} \subseteq \leftrightarrow^*_{\mathcal{R}}$, on ground terms. We complete with Lemma 4.31. □

Moving on to *disproving*, we need two auxiliary lemmas.

Lemma B.6. *If* $\mathcal{R}$ *is confluent and* $(\mathcal{E},\mathcal{H},\text{COMPLETE}) \vdash_{\mathtt{ri}} \bot$*, then* $\mathcal{E}$ *contains an equation* $s \approx t \; [\varphi]$ *that is not an inductive theorem.*

Proof. By confluence and termination together, we can speak of *the* normal form $u{\downarrow_{\mathcal{R}}}$ of any term u; if u is ground, then by quasi-reductivity its normal form is a ground

constructor term. A property of confluence is that if $w \leftrightarrow^*_{\mathcal{R}} q$, then $w\downarrow_{\mathcal{R}} = q\downarrow_{\mathcal{R}}$. Thus, it suffices to prove that for some $s \approx t\ [\varphi] \in \mathcal{E}$ there is a ground constructor substitution γ that respects this equation such that $s\gamma$ and $t\gamma$ have distinct normal forms.

The only inference rule that could be used to obtain $(\mathcal{E}, \mathcal{H}, \text{COMPLETE}) \vdash_{\texttt{ri}} \bot$ is DISPROVE, so $\mathcal{E} = \mathcal{E}' \cup \{s \simeq t\ [\varphi]\}$ and one of the following holds:

(1) $s, t \in \mathit{Terms}(\Sigma_{\mathit{theory}}, \mathcal{V})$ with $\varphi \wedge s \neq t$ satisfiable. In other words, there is a substitution γ mapping all variables in the equation to values, such that $\varphi\gamma$ is valid and $s\gamma$ and $t\gamma$ reduce to different values by $\rightarrow_{\texttt{calc}}$. We are done since all values are normal forms.
(2) $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ with f and g different constructors, and φ is satisfiable, so there is a substitution δ mapping all variables in φ to values, such that $\varphi\delta$ is valid. Let γ be an extension of δ that additionally maps all other variables in s, t to ground terms (by assumption, ground instances of all variables exist). Then $\varphi\gamma$ is still valid, and $s\gamma$ and $t\gamma$ are ground terms with $s\gamma \rightarrow^*_{\mathcal{R}} (s\gamma)\downarrow_{\mathcal{R}} = f((\vec{s}\gamma)\downarrow_{\mathcal{R}}) \neq g((\vec{t}\gamma)\downarrow_{\mathcal{R}}) = (t\gamma)\downarrow_{\mathcal{R}} \leftarrow^*_{\mathcal{R}} t\gamma$.
(3) $s : \iota$ is a variable not occurring in φ, φ is satisfiable, there are at least two different constructors f, g with output sort ι, and either t is a variable distinct from s or t has a constructor symbol at the root. By satisfiability of φ, a substitution δ exists whose domain does not contain s, with $\varphi\delta$ valid. If t is a variable, let γ be an extension of δ mapping s to some ground term rooted by f and t to a ground term rooted by g (by assumption, ground instances always exist). If $t = f(\vec{t})$, then let γ be an extension of δ mapping s to some ground term rooted by g and mapping all other variables in t to ground terms as well. Either way, $\varphi\gamma$ is valid and $(s\gamma)\downarrow_{\mathcal{R}} \neq (t\gamma)\downarrow_{\mathcal{R}}$. □

LEMMA B.7. *Suppose that $\rightarrow_{\mathcal{R}\cup\mathcal{H}}$ is terminating and that $\rightarrow_{\mathcal{R}\cup\mathcal{H}} \subseteq \rightarrow_{\mathcal{R}} \cdot \rightarrow^*_{\mathcal{R}\cup\mathcal{H}} \cdot \leftarrow\!\!\Vdash\!\!\rightarrow_{\mathcal{E}} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}}$. If, moreover, $\mathcal{R}$ is confluent, $(\mathcal{E}, \mathcal{H}, \text{COMPLETE}) \vdash_{\texttt{ri}} (\mathcal{E}', \mathcal{H}', \text{COMPLETE})$, and $\leftrightarrow_{\mathcal{E}} \cup \leftrightarrow_{\mathcal{H}} \subseteq \leftrightarrow^*_{\mathcal{R}}$ on ground terms, then $\leftrightarrow_{\mathcal{E}'} \cup \leftrightarrow_{\mathcal{H}'} \subseteq \leftrightarrow^*_{\mathcal{R}}$ on ground terms.*

PROOF. Assume that all conditions are satisfied; we consider the inference rule used to derive $(\mathcal{E}, \mathcal{H}, \text{COMPLETE}) \vdash_{\texttt{ri}} (\mathcal{E}', \mathcal{H}', \text{COMPLETE})$.

First, suppose that the rule used was COMPLETENESS, so $(\mathcal{E}, \mathcal{H}, \text{COMPLETE}) \vdash^*_{\texttt{ri}} (\mathcal{E}', \mathcal{H}', \text{INCOMPLETE})$ and $\mathcal{E}' \subseteq \mathcal{E}$. As we have assumed that $\leftrightarrow_{\mathcal{E}} \cup \leftrightarrow_{\mathcal{H}} \subseteq \leftrightarrow^*_{\mathcal{R}}$, certainly $\leftrightarrow_{\mathcal{E}'} \subseteq \leftrightarrow_{\mathcal{E}} \subseteq \leftrightarrow^*_{\mathcal{R}}$. As for $\leftrightarrow_{\mathcal{H}'}$, Lemma B.4 gives us that $\rightarrow_{\mathcal{R}\cup\mathcal{H}'} \subseteq \rightarrow_{\mathcal{R}} \cdot \rightarrow^*_{\mathcal{R}\cup\mathcal{H}'} \cdot \leftarrow\!\!\Vdash\!\!\rightarrow_{\mathcal{E}'} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'}$, so (using again that $\leftrightarrow_{\mathcal{E}'} \subseteq \leftrightarrow_{\mathcal{R}}$ and that $\leftarrow\!\!\Vdash\!\!\rightarrow_{\mathcal{E}'} \subseteq \leftrightarrow^*_{\mathcal{E}'}$) we can apply Lemma 4.32 and termination of $\rightarrow_{\mathcal{R}\cup\mathcal{H}'}$ to obtain $\leftrightarrow_{\mathcal{H}'} \subseteq \leftrightarrow_{\mathcal{R}\cup\mathcal{H}'} \subseteq \leftrightarrow^*_{\mathcal{R}}$.

If a different rule was applied, then each element in $\mathcal{H}'$ either also belongs to $\mathcal{H}$ or (in the case of EXPANSION) corresponds to an equation in $\mathcal{E}$. Thus, $\leftrightarrow_{\mathcal{H}'} \subseteq \leftrightarrow_{\mathcal{E}\cup\mathcal{H}} \subseteq \leftrightarrow^*_{\mathcal{R}}$. So let $s \approx t\ [\varphi] \in \mathcal{E}' \setminus \mathcal{E}$; we must see that $\leftrightarrow_{\{s\approx t\ [\varphi]\}} \subseteq \leftrightarrow^*_{\mathcal{R}}$ on ground terms. By Lemma 4.31, it suffices if for all ground constructor substitutions γ that respect this equation, $s\gamma \leftrightarrow^*_{\mathcal{R}} t\gamma$. We fix γ and use a case analysis on the applied inference rule:

—(SIMPLIFICATION). There is $s' \simeq t'\ [\varphi'] \in \mathcal{E}$ such that $s' \approx t'\ [\varphi'] \rightarrow_{\mathcal{R}\cup\mathcal{H}} s \approx t\ [\varphi]$ at position $1 \cdot p$. By Theorem 2.20, we can find δ that respects φ' such that $s'\delta \rightarrow_{\mathcal{R}\cup\mathcal{H}} s\gamma$ at position p and $t'\delta = t\gamma$. As $\rightarrow_{\mathcal{R}} \cup \rightarrow_{\mathcal{H}} \subseteq \leftrightarrow^*_{\mathcal{R}}$ by the assumption, $s\gamma \leftrightarrow^*_{\mathcal{R}} s'\delta \leftrightarrow_{\mathcal{E}} t'\delta = t\gamma$, which suffices because $\leftrightarrow_{\mathcal{E}} \subseteq \leftrightarrow^*_{\mathcal{R}}$.
—(DELETION). No equations are added in this case.
—(EXPANSION). There is $s' \simeq t'\ [\varphi'] \in \mathcal{E}$ such that $s \approx t\ [\varphi] \in \mathit{Expd}(s' \approx t'\ [\varphi'], p)$ for some p. By Lemma B.1(2), we have $s\gamma\ (\leftarrow_{\mathcal{R}} \cdot \leftrightarrow_{\mathcal{E}})\ t\gamma$, which suffices because $\leftrightarrow_{\mathcal{E}} \subseteq \leftrightarrow^*_{\mathcal{R}}$.
—(EQ-DELETION) $s \simeq t\ [\varphi'] \in \mathcal{E}$, where $\varphi = \varphi' \wedge \neg(s_1 = t_1 \wedge \cdots \wedge s_n = t_n)$, and $s = C[s_1, \ldots, s_n]$, $t = C[t_1, \ldots, t_n]$ for some C, $\vec{s}$, $\vec{t}$. Since any substitution that respects φ also respects φ', we must have $s\gamma \leftrightarrow_{\mathcal{E}} t\gamma$, so $s\gamma \leftrightarrow^*_{\mathcal{R}} t\gamma$.
—(DISPROVE) A reduction with this rule does not have the required form.

—(CONSTRUCTOR) There is $f(\ldots,s,\ldots) \approx f(\ldots,t,\ldots)\ [\varphi] \in \mathcal{E}$, and by assumption $f(\ldots,s,\ldots)\gamma \leftrightarrow_{\mathcal{R}}^{*} f(\ldots,t,\ldots)\gamma$. By confluence, this means that $f(\ldots,s\gamma,\ldots)\downarrow_{\mathcal{R}} = f(\ldots,t\gamma,\ldots)\downarrow_{\mathcal{R}}$, which implies that $(s\gamma)\downarrow_{\mathcal{R}} = (t\gamma)\downarrow_{\mathcal{R}}$.
—(POSTULATE, GENERALIZATION) A reduction with these rules does not have the form required by the lemma (as the COMPLETE flag is removed). □

This leads to the second part of Theorem 4.4, which largely corresponds to Lemma 4.34.

LEMMA B.8. *If $\mathcal{R}$ is confluent and $(\mathcal{E},\emptyset,\text{COMPLETE}) \vdash_{\mathtt{ri}} \cdots \vdash_{\mathtt{ri}} \bot$, then there is some equation in $\mathcal{E}$ that is not an inductive theorem of $\mathcal{R}$.*

PROOF. If $(\mathcal{E},\emptyset,\text{COMPLETE}) = (\mathcal{E}_1,\mathcal{H}_1,\mathit{flag}_1) \vdash_{\mathtt{ri}} \cdots \vdash_{\mathtt{ri}} (\mathcal{E}_n,\mathcal{H}_n,\mathit{flag}_n) \vdash_{\mathtt{ri}} \bot$, then we easily see that $\mathit{flag}_i = \text{COMPLETE}$ for all i. By Lemma B.6, $\mathcal{E}_n$ contains an equation $s \approx t\ [\varphi]$ that is not an inductive theorem. Then $\leftrightarrow_{\mathcal{E}_n} \not\subseteq \leftrightarrow_{\mathcal{R}}^{*}$ on ground terms. By Lemma B.7, Lemma B.4, and induction on $n-i$, this means that $\leftrightarrow_{\mathcal{E}} \cup \leftrightarrow_{\emptyset} \not\subseteq \leftrightarrow_{\mathcal{R}}^{*}$ on ground terms, so by Lemma 4.31, not all $e \in \mathcal{E}$ are inductive theorems. □

PROOF OF THEOREM 4.4. Immediately by Lemmas B.5 and B.8. □

C. SIMPLE SUM

To demonstrate the difference in power between our technique and earlier work, even when not considering advanced data structures that were not supported in Nakabayashi et al. [2010] or Falke and Kapur [2012], we have included an example that can be handled with the technique in this article (and is automatically proved by Ctrl), but not with Nakabayashi et al. [2010] or Falke and Kapur [2012] (the latter of which is not surprising, as it does not use any lemma generation at all).

Example C.1. In the programming course in Nagoya, students in the first lecture were asked to implement a function `sum` that computes the summation from 0 to a given nonnegative integer x. The teacher's reference implementation was the following.

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

Some of the students solved (or tried to solve) this in the following clever way instead.

```
int sum1(int x) {
    return x * (x + 1) / 2;
}
```

```
int sum2(int x) {
    return x * (x - 1) / 2;
}
```

To stay close to the transformation from Nakabayashi et al. [2010] (which does not use the return and error symbols), we consider the following translation.

$$\begin{array}{rcll} \mathsf{sum}(x) & \to & \mathsf{u}(x,1,0) & \\ \mathsf{u}(x,i,z) & \to & \mathsf{u}(x,i+1,z+i) & [i \le x] \\ \mathsf{u}(x,i,z) & \to & z & [i > x] \\ \mathsf{sum1}(x) & \to & x*(x+1)\ \mathsf{div}\ 2 & \\ \mathsf{sum2}(x) & \to & x*(x-1)\ \mathsf{div}\ 2 & \end{array}$$

Our implementation succeeds in proving that $\mathsf{sum}(n) \approx \mathsf{sum1}(n)$ $[n \geq 0]$ is an inductive theorem and that $\mathsf{sum}(n) \approx \mathsf{sum2}(n)$ $[n \geq 0]$ is not. We also succeed on the translation using the methods in the current work. On the other hand, the method in Nakabayashi et al. [2010] fails to prove or disprove these claims.

D. SOME EXAMPLES WE CANNOT HANDLE

To demonstrate the kind of problems that Ctrl cannot yet handle, we compare a recursive definition `sum` of the function $n \mapsto \sum_{i=1}^{n} i$ to three iterative implementations.

```
int sum(n) {
  if (n < 0) return 0;
  return n + sum(n-1);
}
```

```
int sum1(n) {
  int i = 0, j = 0, sum = 0;
  for (; i <= n; i++,j++) sum += j;
  return len;
}
```

```
int sum2(int n){
  int i,sum=0;
  for (i=n;i>=0;i--)
    sum=sum+i;
  return sum;
}
```

```
int sum3(n) {
  int ret = 0;
  for (int i = 0; i <= n; i++)
    for (int j = 0; j < i; j++)
ret++;
  return ret;
}
```

Equivalence between `sum` and each of `sum1`, `sum2`, and `sum3` fails for the three main reasons discussed in Section 6.2. For `sum1`, generalizing the initialization variables loses the information that always $i = j$. For `sum2`, our main generalization method (Section 5.1) does not apply because we do not recognize $i = n$ as an initialization. For `sum3`, our strategy fails because the two loop counters are generalized together.

ACKNOWLEDGMENTS

We are grateful to Stephan Falke, who contributed to an older version of this work, and for the helpful remarks of the reviewers for Kop and Nishida [2014] and for the present article.

REFERENCES

Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. 2008. Removing useless variables in cost analysis of java bytecode. In *Proceedings of the 2008 SAC Conference (SAC'08)*. 368–375.

Elvira Albert, Miguel Gómez-Zamalloa, Laurent Hubert, and Germán Puebla. 2007. Verification of Java bytecode using analysis and transformation of logic programs. In *Proceedings of the 2007 PADL Conference (PADL'07)*. 124–139.

Christophe Alias and Denis Barthou. 2003. Algorithm recognition based on demand-driven data-flow analysis. In *Proceedings of the 2003 WCRE Conference (WCRE'03)*. 296–305.

María Alpuente, Santiago Escobar, and Salvador Lucas. 2007. Removing redundant arguments automatically. *Theory and Practice of Logic Programming* 7, 1–2, 3–35.

Raymond Aubin. 1979. Mechanizing structural induction part I: Formal system. *Theoretical Computer Science* 9, 3, 329–345.

Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That*. Cambridge University Press.

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Mathematical Structures in Computer Science* 21, 6, 1207–1252. http://dx.doi.org/10.1017/S0960129511000193

David A. Basin and Toby Walsh. 1992. Difference matching. In *Proceedings of the 1992 CADE Conference (CADE'92)*. 295–309.

Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. 2009. Software model checking via large-block encoding. In *Proceedings of the 2009 FMCAD Conference (FMCAD'09)*. 25–32.

Adel Bouhoula. 1997. Automated theorem proving by test set induction. *Journal of Symbolic Computation* 23, 1, 47–77.

Adel Bouhoula and Florent Jacquemard. 2008a. Automated induction with constrained tree automata. In *Proceedings of the 2008 IJCAR Conference (IJCAR'08)*. 539–554.

Adel Bouhoula and Florent Jacquemard. 2008b. *Automated Induction for Complex Data Structures*. Technical Report. Available at http://arxiv.org/abs/0811.4720.

Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. 2016. T2: Temporal property verification. In *Proceedings of the 2016 TACAS Conference (TACAS'16)*. 387–393.

Alan Bundy. 2001. The automation of proof by mathematical induction. In *Handbook of Automated Reasoning*. Elsevier, 845–911.

Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. 2005. *Rippling: Meta-Level Guidance for Mathematical Reasoning*. Cambridge University Press.

Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. 1993. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence* 62, 2, 185–253.

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *Proceedings of the 2015 NFM Conference (NFM'15)*. 3–11.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the 2008 TACAS Conference (TACAS'08)*. 337–340.

Stephan Falke. 2009. *Term Rewriting with Built-In Numbers and Collection Data Structures*. Ph.D. Dissertation. University of New Mexico.

Stephan Falke and Deepak Kapur. 2009. A term rewriting approach to the automated termination analysis of imperative programs. In *Proceedings of the 2009 CADE Conference (CADE'09)*. 277–293.

Stephan Falke and Deepak Kapur. 2012. Rewriting induction + linear arithmetic = Decision procedure. In *Proceedings of the 2012 IJCAR Conference (IJCAR'12)*. 241–255.

Stephan Falke, Deepak Kapur, and Carsten Sinz. 2011. Termination analysis of C programs using compiler intermediate languages. In *Proceedings of the 2011 RTA Conference (RTA'11)*. 41–50.

Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression verification. In *Proceedings of the 2014 ASE Conference (ASE'14)*. 349–360.

Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. 2009. Proving termination of integer term rewriting. In *Proceedings of the 2009 RTA Conference (RTA'09)*. 32–47.

Yuki Furuichi, Naoki Nishida, Masahiko Sakai, Keiichiro Kusakari, and Toshiki Sakabe. 2008. Approach to procedural-program verification based on implicit induction of constrained term rewriting systems. *IPSJ Transactions on Programming* 1, 2, 100–121. In Japanese; translated summary at http://www.trs.css.i.nagoya-u.ac.jp/crisys/.

Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, et al. 2017. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning* 58, 1, 3–31.

Jürgen Giesl, Armin Kühnemann, and Janis Voigtländer. 2007. Deaccumulation techniques for improving provability. *Journal of Logic and Algebraic Programming* 71, 2, 79–113.

Benny Godlin and Ofer Strichman. 2008. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica* 45, 6, 403–439.

Benny Godlin and Ofer Strichman. 2013. Regression verification: Proving the equivalence of similar programs. *Software Testing, Verification and Reliability* 23, 3, 241–258.

Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. 2011. Predicate abstraction and refinement for verifying multi-threaded programs. In *Proceedings of the 2011 POPL Conference (POPL'11)*. 331–344.

Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. 2013. Towards modularly comparing programs using automated theorem provers. In *Proceedings of the 2013 CADE Conference (CADE'13)*. 282–299.

Gérard P. Huet and Jean-Marie Hullot. 1982. Proofs by induction in equational theories with constructors. *Journal of Computer and System Sciences* 25, 2, 239–266.

Michael Huth and Mark Ryan. 2000. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press.

Deepak Kapur and Nikita A. Sakhanenko. 2003. Automatic generation of generalization lemmas for proving properties of tail-recursive definitions. In *Proceedings of the 2003 TPHOLs Conference (TPHOLs'03)*. 136–154.

Deepak Kapur and Mahadevan Subramaniam. 1996. Lemma discovery in automated induction. In *Proceedings of the 1996 CADE Conference (CADE'96)*. 538–552.

Hirotaka Koike and Yoshihito Toyama. 2000. Comparison between inductionless induction and rewriting induction. *Computer Software* 17, 6, 1–12. In Japanese.

Cynthia Kop. 2013. Termination of LCTRSs. In *Proceedings of the 2013 WST Conference (WST'13)*. 59–63.

Cynthia Kop. 2017. *Quasi-Reductivity of Logically Constrained Term Rewriting Systems*. Technical Report. Available at https:/arxiv.org/abs/1702.02397.

Cynthia Kop and Naoki Nishida. 2013. Term rewriting with logical constraints. In *Proceedings of the 2013 FroCoS Conference (FroCoS'13)*. 343–358.

Cynthia Kop and Naoki Nishida. 2014. Automatic constrained rewriting induction towards verifying procedural programs. In *Proceedings of the 2014 APLAS Conference (ASPLAS'14)*. 334–353.

Cynthia Kop and Naoki Nishida. 2015. Constrained rewriting tool. In *Proceedings of the 2015 LPAR Conference (LPAR'15)*. 549–557.

Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 2009 PLDI Conference (PLDI'09)*. 327–337.

Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 2012 CAV Conference (CAV'12)*. 712–717.

Nuno P. Lopes and José Monteiro. 2016. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *International Journal on Software Tools for Technology Transfer* 18, 4, 359–374.

John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM* 3, 4, 184–195.

Naoki Nakabayashi, Naoki Nishida, Keiichiro Kusakari, Toshiki Sakabe, and Masahiko Sakai. 2010. Lemma generation method in rewriting induction for constrained term rewriting systems. *Computer Software* 28, 1, 173–189. In Japanese; translation at http://www.trs.css.i.nagoya-u.ac.jp/crisys/.

George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the 2000 PLDI Conference (PLDI'00)*. 83–94.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to DPLL(T). *Journal of the ACM* 53, 6, 937–977.

Carsten Otto, Marc Brockschmidt, Christan von Essen, and Jürgen Giesl. 2010. Automated termination analysis of Java bytecode by term rewriting. In *Proceedings of the 2010 RTA Conference (RTA'10)*. 259–276.

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *Proceedings of the 1998 TACAS Conference (TACAS'98)*. 151–166.

Uday S. Reddy. 1990. Term rewriting induction. In *Proceedings of the 1990 CADE Conference (CADE'90)*. 162–177.

Tsubasa Sakata, Naoki Nishida, and Toshiki Sakabe. 2011. On proving termination of constrained term rewrite systems by eliminating edges from dependency graphs. In *Proceedings of the 2011 WFLP Conference (WFLP'11)*. 138–155.

Tsubasa Sakata, Naoki Nishida, Toshiki Sakabe, Masahiko Sakai, and Keiichiro Kusakari. 2009. Rewriting induction for constrained term rewriting systems. *IPSJ Transactions on Programming* 2, 2, 80–96. In Japanese; a translated summary is available at http://www.trs.css.i.nagoya-u.ac.jp/crisys/.

Fausto Spoto, Lunjin Lu, and Fred Mesnard. 2009. Using CLP simplifications to improve Java bytecode termination analysis. *Electronic Notes in Theoretical Computer Science* 253, 5, 129–144.

SV-COMP. 2017. Competition on Software Verification. Retrieved May 1, 2017, from http://sv-comp.sosy-lab.org/

Tachio Terauchi and Alexander Aiken. 2005. Secure information flow as a safety problem. In *Proceedings of the 2005 SAS Conference (SAS'05)*. 352–367.

Pascal Urso and Emmanuel Kounalis. 2004. Sound generalizations in mathematical induction. *Theoretical Computer Science* 323, 1–3, 443–471.

Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. 2012. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Transactions on Programming Languages and Systems* 34, 3, 11.

Milena Vujosevic-Janicic, Mladen Nikolic, Dusan Tosic, and Viktor Kuncak. 2013. Software verification and graph similarity for automated evaluation of students' assignments. *Information and Software Technology* 55, 6, 1004–1016.

Toby Walsh. 1996. A divergence critic for inductive proof. *Journal of Artificial Intelligence Research* 4, 209–235.

Received December 2015; revised February 2017; accepted February 2017