Towards Verifying Procedural Programs using Constrained Rewriting Induction*

Cynthia Kop[1] and Naoki Nishida[2]

[1] Institute of Computer Science, University of Innsbruck
Cynthia.Kop@uibk.ac.at
[2] Graduate School of Information Science, Nagoya University
nishida@is.nagoya-u.ac.jp

Abstract. This paper aims at developing a verification method for procedural programs via a transformation into the recently introduced logically constrained term rewriting systems (LCTRSs). To this end, we introduce an extension of transformation methods based on integer TRSs, which can also handle global variables and arrays, and 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, involving for instance integers and arrays.

1 Introduction

A problem familiar to many computer science lecturers, is the marking of student programming assignments. This can be large time drain, as it typically involves checking dozens (or hundreds!) of unnecessarily complicated programs at once.

An obvious solution is automatic testing. For example, one might run assignments on a fixed set of input files; this quickly weeds out incorrect solutions, but has a high risk of false positives. Alternatively (or in addition), we can try to automatically prove correctness. Several methods for this have been investigated (see e.g., [9]). However, most of them require expert knowledge to use, like *assertions* in the code to trace relevant properties; this is not useful in our setting.

An interesting alternative is *inductive theorem proving*, which is well investigated in the field of functional programming (see, e.g., [2]). For a functional program f to be checked against a specification f_{spec}, it suffices if $f(\overrightarrow{x}) \approx f_{spec}(\overrightarrow{x})$ is an *inductive theorem* of the combined system of f and f_{spec}. For this initial setting, no expert knowledge is required, only the definitions of f and f_{spec}.

Recently, analyses of procedural programs (in C, Java Bytecode, etc.) via transformations into term rewrite systems have been investigated [4,6,8,18]. In particular, *constrained rewriting systems* are popular for these transformations, since logical constraints used for modeling the control flow can be separated

* This research is supported by the Austrian Science Fund (FWF) international project I963, the Japan Society for the Promotion of Science (JSPS) and *Nagoya University's Graduate Program for Real-World Data Circulation Leaders* from *MEXT*, Japan.

from terms expressing intermediate states [8,4,21,17,6]. To capture the existing approaches for constrained rewriting in one setting, the framework of *logically constrained term rewriting systems* (LCTRS) has been proposed [14].

The aim of this paper is twofold. First, we define a new transformation method from procedural programs into constrained term rewriting. This transformation – which is designed to give LCTRSs which closely follow the original control flow – will allow us to use the many methods available to term rewriting (which generally extend to constrained rewriting naturally) also to analyze imperative programs. Unlike previous methods, we do not limit interest to integer functions.

Second, we develop a verification method for LCTRSs, designed in particular for LCTRSs obtained from procedural programs. We use rewriting induction [19], one of the well-investigated methods for inductive theorem proving, together with a generalization technique that performs particularly well for transformed iterative functions. Although our examples focus on integers and static integer arrays, the results can be used with various theories.

Of course, verification also has applications outside the academic world. Although we initially focus on typical homework assignments (small programs, which require only limited language features), we hope to additionally lay a basis for more extensive program analysis using constrained term rewriting systems.

In this paper, we first recall the LCTRS formalism from [14] (Section 2), and show a way to translate procedural programs to LCTRSs (Section 3). Then we adapt existing rewriting induction methods for earlier variations of constrained rewriting [5,21] to LCTRSs (Section 4), which is strengthened with a dedicated generalization technique (Section 5). Finally, we briefly discuss implementation ideas (Section 6), give a comparison with related work (Section 7) and conclude.

This is an extended version of [15], which is presented at APLAS'14.

1.1 Motivating Example

Consider the following programming assignment:

> *Write a function* sum *which, 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 is correct. The second is not, because `ret` is not initialized to 0 – but depending on the compiler used, this error might not be noticed in standard tests. The third solution is incorrect for two reasons: first, the array is modified against the instructions, and second, if `len` $= 0$, we might get a random result or a segmentation fault. The fourth solution uses recursion, and is correct.

These implementations can be transformed into the following LCTRSs:

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

$$
\begin{aligned}
\mathsf{sum2}(arr, n) &\to \mathsf{u}(arr, n, ret, 0) \\
&\ \mathsf{u} \text{ rules as copied from above}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{sum3}(arr, len) &\to \mathsf{v}(arr, len, 0) \\
\mathsf{v}(arr, len, i) &\to \mathsf{error} && [i < len - 1 \wedge (i < 0 \vee i + 1 \geq \mathsf{size}(arr))] \\
\mathsf{v}(arr, len, i) &\to \mathsf{v}(\mathsf{store}(arr, i+1, \mathsf{select}(arr, i+1) + \mathsf{select}(arr, i)), i+1) \\
& && [i < len \wedge 0 \leq i \wedge i + 1 < \mathsf{size}(arr)] \\
\mathsf{v}(arr, len, i) &\to \mathsf{return}(arr, \mathsf{select}(arr, len - 1)) && [i \geq len - 1]
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{sum4}(arr, k) &\to \mathsf{return}(arr, 0) && [k \leq 0] \\
\mathsf{sum4}(arr, k) &\to \mathsf{error} && [k - 1 \geq \mathsf{size}(arr)] \\
\mathsf{sum4}(arr, k) &\to \mathsf{w}(\mathsf{select}(arr, k-1), \mathsf{sum4}(arr, k-1)) && [0 \leq k - 1 < \mathsf{size}(arr)] \\
\mathsf{w}(n, \mathsf{error}) &\to \mathsf{error} \\
\mathsf{w}(n, \mathsf{return}(a, r)) &\to \mathsf{return}(a, n + r)
\end{aligned}
$$

Note that arrays carry an implicit size (their allocated memory) which can be queried; this is used to automatically check for out-of-bound errors. Note also the use of a fresh variable in the right-hand side of the sum2 rule, modelling that the third parameter of u is assigned an *arbitrary* integer. The details of these transformations are discussed in Section 3.

Using inductive theorem proving, we can now prove:

$$\mathsf{sum1}(arr, len) \leftrightarrow^* \mathsf{sum4}(arr, len) \ [len \leq \mathsf{size}(arr)]$$

That is, functions sum1 and sum4, given input that makes sense (so the given length won't cause out-of-bound errors), return the same result.

We can also use inductive theorem proving to find that:

$$
\begin{aligned}
\mathsf{sum2}(arr, len) &\not\leftrightarrow^* \mathsf{sum4}(arr, len) \ [len \leq \mathsf{size}(arr)] \\
\mathsf{sum3}(arr, len) &\not\leftrightarrow^* \mathsf{sum4}(arr, len) \ [len \leq \mathsf{size}(arr)]
\end{aligned}
$$

So sum2 and sum3 do not define the same function as sum1 and sum4. Clearly, both results are useful when marking assignments.

2 Preliminaries

In this section, we briefly recall *Logically Constrained Term Rewriting Systems* (usually abbreviated as *LCTRSs*), following the definitions in [14].

2.1 Logically Constrained Term Rewriting Systems

Many-sorted Terms. 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 symbol equipped with a *sort declaration* $[\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa$, with all ι_i and κ sorts. 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:

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

Fixing Σ and $\mathcal{V}$, every term has a unique sort ι such that $\vdash s : \iota$; we say that ι is the sort of s. Let $Var(s)$ be the set of variables occurring in s. A term s is *ground* if $Var(s) = \emptyset$. A *substitution* is a sort-preserving mapping $[x_1 := s_1, \ldots, x_k := s_k]$ from variables to terms. The result $s\gamma$ of applying a substitution γ on a term s is s with all occurrences of any x_i replaced by s_i.

Given a term s, a *position* in s is a sequence p of 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 say that $s_{|p}$ is 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. We fix a signature $\Sigma = \Sigma_{terms} \cup \Sigma_{theory}$ (with possible overlap, as discussed below). The sorts occurring in Σ_{theory} are called *theory sorts*, and the symbols *theory symbols*. We assume given a mapping $\mathcal{I}$ which assigns to each theory sort ι a set $\mathcal{I}_\iota$, and a mapping $\mathcal{J}$ which maps each $f : [\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa \in \Sigma_{theory}$ to a function $\mathcal{J}_f$ in $\mathcal{I}_{\iota_1} \times \cdots \times \mathcal{I}_{\iota_n} \Longrightarrow \mathcal{I}_\kappa$. For all theory sorts ι we also fix a set $\mathcal{V}al_\iota \subseteq \Sigma_{theory}$ of *values*: function symbols $a : [] \Rightarrow \iota$, where $\mathcal{J}$ gives a bijective mapping from $\mathcal{V}al_\iota$ to $\mathcal{I}_\iota$. We require that $\Sigma_{terms} \cap \Sigma_{theory} \subseteq \mathcal{V}al = \bigcup_\iota \mathcal{V}al_\iota$.

A term in $\mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$ is called a *logical term*. For ground logical terms, let $[\![f(s_1, \ldots, s_n)]\!] := \mathcal{J}_f([\![s_1]\!], \ldots, [\![s_n]\!])$. Every ground logical term s corresponds to 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*. We say φ is *valid* if $[\![\varphi\gamma]\!] = \top$ for *all* substitutions γ which map $Var(\varphi)$ to values, and *satisfiable* if $[\![\varphi\gamma]\!] = \top$ for *some* substitutions γ which map $Var(\varphi)$ to values. A substitution γ *respects* φ if $\gamma(x)$ is a value for all $x \in Var(\varphi)$ and $[\![\varphi\gamma]\!] = \top$.

Formally, terms in $\mathcal{T}erms(\Sigma_{terms}, \mathcal{V})$ have no special function, but we see them as the primary objects of the term rewriting system: a reduction would typically begin and end with such terms, with elements of $\Sigma_{theory} \setminus \mathcal{V}al$ (also called *calculation symbols*) only used in intermediate terms; their function is 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 the core theory symbols: $\mathsf{true}, \mathsf{false} : \mathsf{bool}, \wedge, \vee, \Rightarrow\colon [\mathsf{bool} \times \mathsf{bool}] \Rightarrow \mathsf{bool}$, $\neg\colon [\mathsf{bool}] \Rightarrow \mathsf{bool}$, and, for all sorts ι, symbols $=_\iota, \neq_\iota\colon [\iota \times \iota] \Rightarrow \mathsf{bool}$, and an evaluation function $\mathcal{J}$ that interprets these symbols as expected. We omit the sort subscripts from $=$ and $\neq$ when they can be derived from context.

The standard integer signature Σ^{int}_{theory} is $\Sigma^{core}_{theory} \cup \{+, -, *, \mathsf{exp}, \mathsf{div}, \mathsf{mod} : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int}; \leq, <\colon [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{bool}\} \cup \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\}$. Here, values are true, false and n for all $n \in \mathbb{Z}$. We let $\mathcal{J}$ be defined in the natural way, with one alteration: since all $\mathcal{J}_f$ are required to be total functions, we define $\mathcal{J}_{\mathsf{div}}(n, 0) = \mathcal{J}_{\mathsf{mod}}(n, 0) = \mathcal{J}_{\mathsf{exp}}(n, k) = 0$ for all n and all $k < 0$. Of course, when constructing LCTRSs we normally avoid such calls.

Rules and Rewriting. A *rule* is a triple $\ell \to r\ [\varphi]$ where ℓ and r are terms of the same sort and φ is a constraint. Here, ℓ is not a logical term (so also not a variable, as all variables are elements of $\mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$). If $\varphi = \mathsf{true}$ with $\mathcal{J}(\mathsf{true}) = \top$, the rule is usually just denoted $\ell \to r$. 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 $\gamma(x)$ is a value for all $x \in \mathit{LVar}(\ell \to r\ [\varphi])$, and $\varphi\gamma$ is valid. Note that it is allowed that $\mathit{Var}(r) \not\subset \mathit{Var}(\ell)$, but fresh variables in the right-hand side may only be instantiated with *values*. This is done to model user input or random choice, both of which would typically produce a value. Variables in the left-hand-sides do not need to be instantiated with values (unless they also occur in the constraint); this is needed for instance to support a lazy evaluation strategy.

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:

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

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 $\to_{\mathcal{R}}$. If $f(\ell_1, \ldots, \ell_n) \to r\ [\varphi] \in \mathcal{R}$ we call f a *defined symbol*; non-defined 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 all constructors. A *logically constrained term rewriting system* (LCTRS) is defined as the abstract rewriting system $(\mathcal{T}erms(\Sigma, \mathcal{V}), \to_{\mathcal{R}})$, and usually given by supplying Σ, $\mathcal{R}$, and also $\mathcal{I}$ and $\mathcal{J}$ if these are not clear from context.

Example 1. To implement an LCTRS calculating the *factorial* function, we let $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$, $\mathcal{I}_{\mathsf{bool}} = \mathbb{B}$, $\Sigma_{theory} = \Sigma^{int}_{theory}$, $\mathcal{J}$ defined as discussed above, and:

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

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

Writing $=$ for $=_{\mathsf{int}}$ and using infix notation, examples of logical terms are $0 = 0 + -1$ and $x + 3 \geq y + -42$. Both are constraints. $5 + 9$ is also a (ground) logical term, but not a constraint. We can reduce $5 + 9$ to 14 with a calculation (using $x + y \to z\ [z = x + y]$), and $\mathsf{fact}(3)$ reduces in ten steps to 6.

Example 2. 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 $\Sigma_{theory} = \Sigma^{int}_{theory} \cup \{\mathsf{size} : [\mathsf{array(int)}] \Rightarrow \mathsf{int},\ \mathsf{select} : [\mathsf{array(int)} \times \mathsf{int}] \Rightarrow \mathsf{int}\}\ \cup\ \{\mathsf{a} \mid a \in \mathbb{Z}^*\}$. (So we do not encode arrays as lists: every array 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:

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

In addition, we have:

$$\begin{array}{rl} \Sigma_{terms} = & \{\ \mathsf{sum} : [\mathsf{array(int)}] \Rightarrow \mathsf{int},\ \mathsf{sum1} : [\mathsf{array(int)} \times \mathsf{int}] \Rightarrow \mathsf{int}\ \} \cup \\ & \{\ \mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\ \} \cup \{\ \mathsf{a} \mid a \in \mathbb{Z}^*\ \} \end{array}$$

$$\mathcal{R}_{\mathsf{sum}} = \left\{ \begin{array}{rl} \mathsf{sum}(x) \to \mathsf{sum1}(x, \mathsf{size}(x) - 1) & \\ \mathsf{sum1}(x,k) \to \mathsf{select}(x,k) + \mathsf{sum1}(x, k-1) & [k \geq 0] \\ \mathsf{sum1}(x,k) \to 0 & [k < 0] \end{array} \right\}$$

Note that this implementation differs from the ones we saw in Section 1.1, because there we were proving encodings of imperative programs; here, we use a similar theory signature, but refrain from including error checking.

Note the special role of *values*, which are new in LCTRSs compared to older styles of constrained rewriting. Values are the representatives of the underlying theory. All values are constants (constructor symbols $v()$ which do not take arguments), even if they represent complex structures, as seen in Example 2. However, not all constants are values; for instance a constant constructor $\mathsf{error} \in \Sigma_{terms}$ would not be a value. Because, unlike traditional TRSs and e.g. [6,8], values are not term-generated, we can easily have uncountably many of them (for example an LCTRS over the real number field $\mathbb{R}$), and do not have to match modulo theories (for example equating $\mathsf{0} + (x + y)$ with $y + x$).

Moreover, note that the restriction that variables in the constraint may only be instantiated with values imposes a sort of strategy on $\to_{\mathcal{R}}$. In Example 1, for instance, the fact-rules may only be applied to a term $\mathsf{fact}(n)$ whose immediate argument n is a value. So, e.g., $\mathsf{fact}(\mathsf{fact}(\mathsf{3}))$ reduces only at position 1.

Differences to [14]. In [14], where LCTRSs are first defined, we assume that $\mathcal{V}$ contains unsorted variables, and use a separate *variable environment* for typing terms. Also, $\to_{\mathcal{R}}$ is there defined as the union of $\to_{\mathtt{rule}}$ (using rules in $\mathcal{R}$) and $\to_{\mathtt{calc}}$ (using calculations), whereas here we simply consider calculations as being given by an additional set of rules. These changes give equivalent results, but the current definitions cause a bit less bookkeeping.

A non-equivalent change is the requirement on rules: in [14] left-hand sides must have a root symbol in $\Sigma_{terms} \setminus \Sigma_{theory}$. We follow [13] in weakening this.

Note that, like [14], we refrain from imposing several natural restrictions, such as *regularity* (for all $\ell \to r\ [\varphi] \in \mathcal{R}$: $Var(\varphi) \subseteq Var(\ell)$), *standardness* (for all $\ell \to r\ [\varphi] \in \mathcal{R}$: $\ell \in \mathcal{T}erms(\Sigma_{terms}, \mathcal{V})$) and *value-safety* (for all theory sorts ι: any constructor with output sort ι is a value). In the case of regularity, this is crucial: the transformation method from Section 3 sometimes creates irregular rules (when given uninitialized variables or user input). Furthermore, "dirty" features may arise during analysis. For example the rewriting induction method of Section 4 might introduce rules which need neither be regular or standard.

2.2 Quantification

The definition of LCTRSs does *not* permit quantifiers. In for instance an LCTRS over integers and arrays, we cannot specify a rule like:

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

(Where $\mathsf{addtoend} : [\mathsf{int}{\times}\mathsf{array}(\mathsf{int})] \Rightarrow \mathsf{array}(\mathsf{int}) \in \Sigma_{theory}$ and $\mathsf{extend} : [\mathsf{array}(\mathsf{int}){\times}\ \mathsf{int}] \Rightarrow \mathsf{array}(\mathsf{int}) \in \mathcal{D}$.)

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

Thus, we can permit quantifiers in the constraints of rules, as intuitive notation for fresh predicates. However, as the reduction relation $\to_{\mathcal{R}}$ is only decidable if all $\mathcal{J}_f$ are, an *unbounded* quantification would likely not be useful in practice.

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*: pairs $s\,[\varphi]$ of a term s and a constraint φ. A constrained term $s\,[\varphi]$ represents all terms $s\gamma$ where γ respects φ, and can be used to reason about such terms.

Different constrained terms might represent the same terms; for example $f(\mathsf{0})\,[\mathsf{true}]$ and $f(x)\,[x = \mathsf{0}]$, or $g(x, y)\,[x > y]$ and $g(z, u)\,[u \leq z - \mathsf{1}]$. We consider these terms *equivalent*. Formally, $s\,[\varphi] \sim t\,[\psi]$ if for all substitutions γ which respect φ there is a substitution δ which respects ψ such that $s\gamma = t\delta$, and vice versa. 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 $Var(s) = \{\overrightarrow{x}\}$, $Var(\varphi) \setminus Var(s) = \{\overrightarrow{y}\}$ and $Var(\psi) \setminus Var(s) = \{\overrightarrow{z}\}$.

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 $s_{|q} = \ell\gamma$ and $t = s[r\gamma]_q$ for some substitution γ such that $\gamma(x)$ is a variable in $Var(\varphi)$ or a value for all $x \in LVar(\ell \to r\ [\psi])$ 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 3. In the factorial LCTRS from Example 1, we have that $\mathsf{fact}(x)\,[x > 3] \to_{\mathcal{R}} x * \mathsf{fact}(x-1)\,[x > 3]$. This constrained term can be further reduced using the calculation rule $x - y \to z\ [z = x - y]$, but here we must use the $\sim$ relation, as follows: $x * \mathsf{fact}(x-1)\,[x > 3] \sim x * \mathsf{fact}(x-1)\,[x > 3 \wedge z = x - 1] \to_{\mathsf{base}} x * \mathsf{fact}(z)\,[x > 3 \wedge z = x - 1]$, as $\forall x[x > 3 \leftrightarrow \exists z[x > 3 \wedge z = x - 1]]$.

Example 4. The $\sim$ relation also allows us to reformulate the constraint after a reduction, which is in particular useful if some variables are not used any longer. For example, with the rule $f(x) \to g(y)\ [y > x]$, we have: $f(x)\,[x > 3] \sim f(x)\,[x > 3 \wedge y > x] \to_{\mathsf{base}} g(y)\,[x > 3 \wedge y > x] \sim g(y)\,[y > 4]$. We do *not* have that $f(x)\ [\mathsf{true}] \to_{\mathcal{R}} g(x+1)\ [\mathsf{true}]$, as $x + 1$ cannot be instantiated to a value.

Example 5. A constrained term does not always need to be reduced in the most general way. With the rule $f(x) \to g(y)\ [y > x]$, we have $f(0)\ [\mathsf{true}] \sim f(0)\ [y > 0] \to_{\mathsf{base}} g(y)\ [y > 0]$, but we also have $f(0)\ [\mathsf{true}] \sim f(0)\ [1 > 0] \to_{\mathsf{base}} g(1)$.

As intended, constrained reductions give information about usual reductions:

Theorem 6. *If $s\,[\varphi] \to_{\mathcal{R}} t\,[\psi]$, then for all substitutions γ which respect φ there is a substitution δ which respects ψ such that $s\gamma \to_{\mathcal{R}} t\delta$. If the former step uses rule ρ at position q, then so does the latter.*

Proof. We first observe (**): *If $s\,[\varphi] \to_{\mathsf{base}} t\,[\psi]$, then $\varphi = \psi$, and for any substitution γ which respects φ also $s\gamma \to_{\mathcal{R}} t\gamma$.*

To see that this holds, we first observe that $\varphi = \psi$ by definition of $\to_{\mathsf{base}}$. Furthermore, there are some position p, rule $\ell \to r\ [c]$ and substitution δ such that $s_{|p} = \ell\delta$ and $t = s[r\delta]_p$. Writing ϵ for the composition $\gamma \circ \delta$ (so $Dom(\epsilon) = Dom(\delta) \cup Dom(\gamma)$ and $\epsilon(x) = (x\delta)\gamma$ for all variables in its domain), we have on the one hand $(s\gamma)_{|p} = s_{|p}\gamma = \ell\delta\gamma = \ell\epsilon$. On the other hand, $t\gamma = s[r\delta]_p\gamma = (s\gamma)[r\delta\gamma]_p = (s\gamma)[r\epsilon]_p$. So indeed $s\gamma \to_{\mathcal{R}} t\gamma$.

Towards proving the theorem, suppose $s\,[\varphi] \to_{\mathcal{R}} t\,[\varphi]$, so $s\,[\varphi] \sim s'\,[\varphi'] \to_{\mathsf{base}} t'\,[\psi'] \sim t\,[\psi]$, and let γ respect φ. By definition of $\sim$, there is some substitution ϵ which respects φ' such that $s\gamma = s'\epsilon$. By (**) also ϵ respects ψ' and $s'\epsilon \to_{\mathcal{R}} t'\epsilon$. Again by definition of $\sim$, we find δ which respects ψ such that $t'\epsilon' = t\delta$. □

Theorem 7. *If $s\,[\varphi] \to_{\mathcal{R}} t\,[\psi]$, then for all substitutions δ which respect ψ there is a substitution γ which respects φ such that $s\gamma \to_{\mathcal{R}} t\delta$. If the former step uses rule ρ at position q, then so does the latter.*

Proof. Parallel to the proof of Theorem 6: if $s\,[\varphi] \sim s'\,[\varphi'] \to_{\mathsf{base}} t'\,[\psi'] \sim t\,[\psi]$, then by definition of $\sim$ there are suitable ϵ, γ such that $t\delta = t'\epsilon \leftarrow_{\mathcal{R}} s'\epsilon = s\gamma$. □

3 Transforming Imperative Programs into LCTRSs

Transformations of imperative programs into constrained rewriting systems operating on integers have been investigated in e.g. [4,6,8]. Even though these papers use different definitions of constrained rewriting, the proposed transformations can easily be adapted to produce LCTRSs that operate on integers, i.e., use Σ_{theory} as in Example 1. What is more, we can extend the ideas to also handle more advanced programming structures, such as arrays and exceptions.

3.1 Transforming Simple Integer Functions

The transformations proposed in [4,6,8] are all based on the same underlying principle. Each function is transformed separately. We let $\overrightarrow{v}$ be a vector containing all parameters and local variables in the function (for simplicity, we ignore global variables for now). For all basic blocks in the function, we introduce a new function symbol, which operates on the variables $\overrightarrow{v}$. Here, a *basic block* is a code segment with a single entry and exit point, e.g. a sequence of assignments. The transition from one basic block to another is encoded as a rule, with assignments being reflected by argument updates in the right-hand side, and conditions by the constraint. Return statements are encoded by reducing to the return value.

Example 8. Consider the following small imperative program:

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

Here, $\overrightarrow{v} = \langle x, i, z \rangle$. There are three basic blocks: u_1 (the initialization of the local variables, which includes both `int z = 1` and `int i = 1`), u_2 (the loop body), and u_3 (the block containing the return-statement). We obtain:

$$\begin{array}{rll} \mathsf{fact}(x) & \to \mathsf{u}_1(x,i,z) & \\ \mathsf{u}_1(x,i,z) & \to \mathsf{u}_2(x,1,1) & \\ \mathsf{u}_2(x,i,z) & \to \mathsf{u}_2(x,i+1,z*i) & [i \leq x] \\ \mathsf{u}_2(x,i,z) & \to \mathsf{u}_3(x,i,z) & [\neg(i \leq x)] \\ \mathsf{u}_3(x,i,z) & \to z & \end{array}$$

Finally, the generated LCTRS is optimized to make it more amenable to analysis: we combine rules whose root symbols occur only once in left-hand sides [6], remove unused parameters (in particular, variables which are not in scope at a given location) and, if appropriate, simplify the constraint (e.g. by removing duplicate clauses or replacing $\neg\varphi$ by the corresponding non-negated atomic formula).

Example 9. Applying these optimizations to the LCTRS from Example 8, we obtain the following, simpler LCTRS:

$$\begin{array}{rll} \mathsf{fact}(x) & \to \mathsf{u}_2(x,1,1) & \\ \mathsf{u}_2(x,i,z) & \to \mathsf{u}_2(x,i+1,z*i) & [i \leq x] \\ \mathsf{u}_2(x,i,z) & \to z & [i > x] \end{array}$$

Note that there is nothing special about the integers; the definition of LCTRSs allows values from all kinds of underlying domains. So, with a suitable theory signature, we could also handle e.g. doubles, encoded as real numbers. If the v_i are typed with arbitrary sorts (not just int), then we can handle programs with various basic data types. More sophisticated structures are discussed in Section 3.5.

3.2 Error Handling

Of course, the transformation of Section 3.1 doesn't quite correspond to what actually happens in a programming language. As computers have limited memory, they do not really work with integers, but rather with *bitvectors*.

There are various ways to deal with this issue. First, we could simply change the theory. Rather than including all integers as values, we let $\mathcal{V}al_{\mathsf{int}} = \{\mathtt{MININT}, \ldots, \mathtt{MAXINT}\}$. Moreover, we alter $\mathcal{J}$ to make $\mathcal{J}_+$, $\mathcal{J}_-$ and $\mathcal{J}_*$ wrap around (e.g. $\mathcal{J}_-(\mathtt{MININT}, 1) = \mathtt{MAXINT}$). The LCTRS we thus create has the same rules, but acts more closely to the real program behavior.

However, in practice we usually do not want integers wrapping around; if we encounter integer overflow, this is typically an *error*, which leads to incorrect results. In order to model this, it must be possible to reduce to a special error state. Although we *could* simply do this by introducing a new constructor $\mathsf{error} : \mathsf{int}$ in Σ_{terms}, this would be inconvenient with an eye on applications and further extensions: doing so, we would also have to add propagation rules such as $\mathsf{error} + x \to \mathsf{error}$. Instead, we will change the way returns are handled. A function f in the original program is assigned output sort result_f. We introduce constructors $\mathsf{error}_f : \mathsf{result}_f$ and $\mathsf{return}_f : [\mathsf{int}] \Rightarrow \mathsf{result}_f$. Now, a statement `return e` is encoded as a reduction to $\mathsf{return}_f(e)$.

Example 10. The system from Example 8 becomes (without yet handling errors):

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

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

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

To avoid symbol density, we will typically use the negation of τ rather than $\neg\tau$.

Example 11. Continuing Example 10, we generate the following rewrite rules:

$$\begin{array}{rll} \mathsf{fact}(x) & \to \mathsf{u}_1(x,i,z) & \\ \mathsf{u}_1(x,i,z) & \to \mathsf{u}_2(x,1,1) & \\ \mathsf{u}_2(x,i,z) & \to \mathsf{u}_2(x,i+1,z*i) & [i \le x \wedge i+1 \le \mathtt{MAXINT} \wedge z*i \le \mathtt{MAXINT} \wedge \\ & & \ z*i \ge \mathtt{MININT}] \\ \mathsf{u}_2(x,i,z) & \to \mathsf{error}_{\mathsf{fact}} & [i \le x \wedge (i+1 > \mathtt{MAXINT} \vee z*i > \mathtt{MAXINT} \vee \\ & & \ z*i < \mathtt{MININT})] \\ \mathsf{u}_2(x,i,z) & \to \mathsf{u}_3(x,i,z) & [\neg(i \le x)] \\ \mathsf{u}_3(x,i,z) & \to \mathsf{return}_{\mathsf{fact}}(z) & \end{array}$$

This system has the following simplified version:

$$\begin{aligned}
\mathsf{fact}(x) &\to \mathsf{u}_2(x,1,1) \\
\mathsf{u}_2(x,i,z) &\to \mathsf{u}_2(x,i+1,z*i) && [i \le x \wedge i+1 \le \texttt{MAXINT} \wedge z*i \le \texttt{MAXINT} \wedge z*i \ge \texttt{MININT}] \\
\mathsf{u}_2(x,i,z) &\to \mathsf{error}_{\mathsf{fact}} && [i \le x \wedge (i+1 > \texttt{MAXINT} \vee z*i > \texttt{MAXINT} \vee z*i < \texttt{MININT})] \\
\mathsf{u}_2(x,i,z) &\to \mathsf{return}_{\mathsf{fact}(z)} && [i > x]
\end{aligned}$$

Note that with the transformation thus modified, we could easily model asserts and a `throw` statement as well. Similarly, we can handle division by zero: whenever s div t or s mod t is used in a basic block, we add $t \neq 0$ to the constraint of the corresponding rule, and add a new rule to reduce to error if $t = 0$.

Note: when transforming a function into an LCTRS, we can *choose* what errors to model. For instance, we could ignore overflows as possible sources of errors (effectively assuming unbounded integers), but still add error-checking for division by zero. Note also that we could let error_f be a constructor which takes an argument, i.e. $\mathsf{error}_f : [\mathsf{Errors}] \Rightarrow \mathsf{result}_f \in \Sigma_{terms}$, where Errors is a sort with constructors IntegerOverflow, DivisionByZero and so on.

3.3 Global Variables

These transformations are all designed for very *local* code: a function never calls other functions, or modifies global variables. Now that we have introduced separate return constructors, the latter is easily changed: we assume that a function symbol is given all global variables that it uses as input, and returns these same global variables as output, along with its return value.

Example 12. Consider the following short C-program, and its translation:

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

$$\begin{aligned}
\mathsf{up}(best,x) &\to \mathsf{u}_1(best,x) \\
\mathsf{u}_1(best,x) &\to \mathsf{return}_{\mathsf{up}}(x,1) && [x > best] \\
\mathsf{u}_1(best,x) &\to \mathsf{u}_2(best,x) && [\neg(x > best)] \\
\mathsf{u}_2(best,x) &\to \mathsf{return}_{\mathsf{up}}(best,0)
\end{aligned}$$

The simplified version of this LCTRS:

$$\begin{aligned}
\mathsf{up}(b,x) &\to \mathsf{return}_{\mathsf{up}}(x,1)\ [x > b] \\
\mathsf{up}(b,x) &\to \mathsf{return}_{\mathsf{up}}(b,0)\ [x \le b]
\end{aligned}$$

3.4 Function Calls

Now, to handle *function calls*, we have to be a little careful. If a function call is used in an expression, say $\mathsf{fact}(3) + 5$, then this expression is not well-sorted in the corresponding LCTRS: $\mathsf{fact}(3)$ has sort $\mathsf{result}_{\mathsf{fact}}$, not int. To avoid this issue, and propagate errors as needed, we do two things:

– Expressions involving function calls which are not just of the form $var =$

$func(arg_1, \ldots, arg_n)$, are split up: the return value of each function call is assigned to a temporary variable before using. For example,

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

is read as:

```
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;
}
```

– Having done this, we consider every function call (assignment) as a separate program point. The call is executed in a separate parameter, and its results tested for errors, and assigned to the relevant variable(s).

Example 13. Following Example 12, suppose we add the following function.

```
void nag() {
  for (int i = 0; i < 10; i++)
    int k;
    cout << "Please type an integer: ";
    cin >> k;
    up(k);
  }
}
```

Note that we do not need to change this program, because the return value of `up(k)` is not used in a complex expression (in fact, is is not used at all!). Considering user input as a random number, and ignoring the observation that no errors can occur in `up`, this procedure gives the following additional rules:

$$\begin{aligned}
\mathsf{nag}(b) &\to \mathsf{v}_1(b, 0, k) \\
\mathsf{v}_1(b, i, k) &\to \mathsf{v}_2(b, i, k', \mathsf{up}(k')) \quad [i < 10] \\
\mathsf{v}_1(b, i, k) &\to \mathsf{return}_{\mathsf{nag}}(b) \quad [\neg(i < 10)] \\
\mathsf{v}_2(b, i, k, \mathsf{error}_{\mathsf{up}}) &\to \mathsf{error}_{\mathsf{nag}} \\
\mathsf{v}_2(b, i, k, \mathsf{return}_{\mathsf{up}}(b', j)) &\to \mathsf{v}_1(b', i + 1, k)
\end{aligned}$$

Note the way the second rule uses a fresh variable in the right-hand side to create an arbitrary integer, which simulates user input. The simplified version:

$$\begin{aligned}
\mathsf{nag}(b) &\to \mathsf{v}_1(b, 0) \\
\mathsf{v}_1(b, i) &\to \mathsf{v}_2(b, i, \mathsf{up}(k')) \quad [i < 10] \\
\mathsf{v}_1(b, i) &\to \mathsf{return}_{\mathsf{nag}}(b) \quad [i \geq 10] \\
\mathsf{v}_2(b, i, \mathsf{error}_{\mathsf{up}}) &\to \mathsf{error}_{\mathsf{nag}} \\
\mathsf{v}_2(b, i, \mathsf{return}_{\mathsf{up}}(b', j)) &\to \mathsf{v}_1(b', i + 1)
\end{aligned}$$

Example 14. The `ncr` program given above is transformed to the following LCTRS (where we test for division by zero but not integer overflow for simplicity):

$$\begin{aligned}
\mathsf{ncr}(x,y) &\to \mathsf{u}_1(x,y,a,t_1,t_2,b)\\
\mathsf{u}_1(x,y,a,t_1,t_2,b) &\to \mathsf{u}_2(x,y,a,t_1,t_2,b,\mathsf{fact}(x))\\
\mathsf{u}_2(x,y,a,t_1,t_2,b,\mathsf{error}_{\mathsf{fact}}) &\to \mathsf{error}_{\mathsf{ncr}}\\
\mathsf{u}_2(x,y,a,t_1,t_2,b,\mathsf{return}_{\mathsf{fact}}(k)) &\to \mathsf{u}_3(x,y,k,t_1,t_2,b,\mathsf{fact}(y))\\
\mathsf{u}_3(x,y,a,t_1,t_2,b,\mathsf{error}_{\mathsf{fact}}) &\to \mathsf{error}_{\mathsf{ncr}}\\
\mathsf{u}_3(x,y,a,t_1,t_2,b,\mathsf{return}_{\mathsf{fact}}(k)) &\to \mathsf{u}_4(x,y,a,k,t_2,b,\mathsf{fact}(x-y))\\
\mathsf{u}_4(x,y,a,t_1,t_2,b,\mathsf{error}_{\mathsf{fact}}) &\to \mathsf{error}_{\mathsf{ncr}}\\
\mathsf{u}_4(x,y,a,t_1,t_2,b,\mathsf{return}_{\mathsf{fact}}(k)) &\to \mathsf{error}_{\mathsf{ncr}} && [t_1 * k = 0]\\
\mathsf{u}_4(x,y,a,t_1,t_2,b,\mathsf{return}_{\mathsf{fact}}(k)) &\to \mathsf{return}_{\mathsf{ncr}}(a \ \mathsf{div}\ (t_1 * k)) && [t_1 * k \neq 0]
\end{aligned}$$

In simplified form, this LCTRS becomes:

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

3.5 Handling Arrays

Finally, let us consider how to handle arrays. We have already hinted at the idea of how to do this in Section 1.1, and now that we have seen how to handle global variables (and therefore a form of side effects), there should be little novelty in the more complete treatment. To start, we need to fix a theory signature and corresponding interpretations. For now, let us handle arrays *without aliasing*.

For a theory sort ι which has at least one value, say $\mathsf{0}_\iota$, we let the data type $\mathsf{array}(\iota)$ be mapped onto finite-length sequences of $\mathcal{I}_\iota$; that is, $\mathcal{I}_{\mathsf{array}(\iota)} = \mathcal{I}_\iota^*$. Additionally, in the theory of arrays over ι, we introduce the following theory symbols (in addition to Σ^{int}_{theory} and whichever other theories we want to support):

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

Note that this notion of arrays is different from the one in SMT-LIB [**?**], where arrays are functions from one domain to another (the domain does not need to be finite, and is often taken to consist of all integers or bitvectors). For program analysis, the alternative use of finite-length sequences seems practical.

To implement array *lookups* $a[i]$ in an LCTRS, we simply use $\mathsf{select}(a, i)$; to guarantee correctness of this test, we add bound checks to the constraint, and reduce to error_f if such a check is not satisfied. To implement array *assignments* $a[i] = e$ in an LCTRS, we replace a by $\mathsf{store}(a, i, e)$; here, too, we add bound checks and possible error reduction. In addition, since the underlying memory of the array was altered, the updated variable must be included in the return value.

Example 15. 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 simply as integer arrays (although alternative choices for $\mathcal{I}_{\mathsf{char}}$ make little difference). Noting that the function does not update `original` at any point, but does potentially update `goal`, the return value must include `goal`. We find the following implementation as an LCTRS:

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

Now it is 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 $\mathsf{sum1}, \mathsf{sum2}$ and $\mathsf{sum4}$, which is not necessary as the array is not modified in these cases. This was done to allow a direct comparison with $\mathsf{sum3}$, where the array *is* modified.

3.6 Handling Pointers

The transformation from Section 3.5 is convenient, because it allows us to abstract from the underlying memory model. The downside is that, by limiting every array to a unique name, we cannot handle programs with array aliasing or pointer arithmetic. Aside from explicit aliasing, this also means that for instance properties we prove about strcpy from Example 15 might fail to hold for a call like $\mathsf{strcpy}(a, a)$, where `goal` and `original` refer to the same memory.

As we only intend to handle *part* of the language, this does not need to be an issue, certainly not for the purpose of testing student programs; in practice, many programs are written without explicit use of pointers. Nevertheless, we *can* also handle programs with more sophisticated pointer use, by including one or more "memory" variable and considering pointers as indexing this memory.

Example 16. Consider the following example C++ function:

```
char *create() {
  int *a = new int[10];
  int *b = a + 1;
  for (int i = 0; i < 10; i += 2) b[i] = 42;
  return a;
}
```

Now, `a` and `b` share memory, and new memory is allocated.

To encode this example, let mem be a variable of sort $\mathsf{array}(\mathsf{array}(\mathsf{int}))$, so the variable represents a sequence of ι-arrays. We introduce a new symbol $\mathsf{allocate} : [\mathsf{array}(\mathsf{array}(\mathsf{int}))\times\mathsf{array}(\mathsf{int})] \Rightarrow \mathsf{array}(\mathsf{array}(\mathsf{int}))$, and define $\mathcal{J}_{\mathsf{allocate}}(\langle a_0,\ldots,a_k\rangle, b) = \langle a_1,\ldots,a_n,b\rangle$. A `char*` variable is represented as a tuple (i,j) of two integers; i indexes mem directly, pointing to the array we want to index, and j gives an offset in this array. The `NULL`-pointer is represented by $(-1,0)$.

The example function is encoded as follows:

$$\begin{aligned}
\mathsf{create}(mem) &\to \mathsf{u}(\mathsf{allocate}(mem,x),\mathsf{size}(mem),0) && [\mathsf{size}(x)=10]\\
\mathsf{u}(mem,ai,ao) &\to \mathsf{v}(mem,ai,ao,ai,ao+1,0)\\
\mathsf{v}(mem,ai,ao,bi,bo,i) &\to \mathsf{error} && [i<10\ \wedge\\
&&& (bo+i<0\ \vee bo+i\geq\mathsf{size}(\mathsf{select}(mem,bi)))]\\
\mathsf{v}(mem,ai,ao,bi,bo,i) &\to \mathsf{v}(\mathsf{store}(mem,bi,\mathsf{store}(\mathsf{select}(mem,bi),bo+i,42)),\\
&&& ai,ao,bi,bo,i+2) && [i<10\ \wedge\\
&&& 0\leq bo+i<\mathsf{size}(\mathsf{select}(mem,bi))]\\
\mathsf{v}(mem,ai,ao,bi,bo,i) &\to \mathsf{return}(mem,ai,ao) && [\neg(i<10)]
\end{aligned}$$

Note how we use irregularity in the first rule to represent the randomness involved in an allocation. Note also that we do not do bound checks for the $\mathsf{select}(mem,bi)$ expressions; this is redundant if we simply take $\mathsf{0}_{\mathsf{array}(\iota)} = \langle\rangle$ (the empty sequence). For the same reason, we do not have to check whether a pointer is null.

Of course, Example 16 is not a formal discussion of how to handle pointers in general; it is meant as an idea on how it *could* be done in future extensions. In this paper, we limit interest to *static* arrays, without aliases and pointer arithmetic. This is because a more sophisticated translation has the downside of creating more complex LCTRSs, which it is likely to be harder to prove properties about!

4 Rewriting Induction for LCTRSs

In this section, we adapt the inference rules from [19,5,21] to inductive theorem proving with LCTRSs. This provides the core theory to use rewriting induction, which will be strengthened with a lemma generalization technique in Section 5.

We start by listing some restrictions 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 prove its correctness (Section 4.3). Compared to [19,5,21], we make several changes to optimally handle the new formalism. We complete with three illustrative examples (Sections 4.4 and 4.5).

4.1 Restrictions

In order for rewriting induction to be successful, we need to impose certain restrictions. We limit interest to LCTRSs which satisfy the following properties:

1. all core theory symbols ($\wedge, \vee, \Rightarrow, \neg$ and each $=_\iota, \neq_\iota$) are present in Σ_{theory};
2. the LCTRS is terminating, so there is no infinite reduction $s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} \ldots$;
3. the system is *quasi-reductive*, i.e., for every term s either $s \in \mathcal{T}erms(\mathcal{C}ons, \emptyset)$ (we say s is a *ground constructor term*), or there is some t such that $s \to_{\mathcal{R}} t$;[3]
4. there are ground terms of every sort occurring in Σ.

Property 1 is just the standard assumption we saw in Section 2. We will need these symbols, for instance, to add new information to a constraint. Termination (property 2) is crucial in the inductive derivation, as the method uses induction on terms, oriented with an extension of $\to_{\mathcal{R}}$. Property 3 which, together with termination, provides *sufficient completeness*, makes it possible to do an exhaustive case analysis on the rules applicable to an equation. It also allows us to assume that variables are always instantiated by ground constructor terms.

The last property is natural, since inductive theorem proving makes a statement about *ground* terms. Function symbols which cannot be assigned ground arguments can simply be omitted. Note that, together with quasi-reductivity and termination, this implies that there are ground *constructor* terms of all sorts.

Methods to prove both quasi-reductivity and termination have previously been published for different styles of constrained rewriting; see e.g. [5] for quasi-reductivity and [7,20] for termination. These methods are easily adapted to LCTRSs. Quasi-reductivity is handled in Appendix A, and is moreover always satisfied by systems obtained from the transformations in Section 3. Some basics of termination analysis for LCTRSs are discussed in [13].

4.2 Rewriting Induction

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

Definition 17. *A* (constrained) equation *is a triple $s \approx t\ [\varphi]$ with s and t terms and φ a constraint. Let $s \simeq t\ [\varphi]$ denote either $s \approx t\ [\varphi]$ or $t \approx s\ [\varphi]$. A substitution* γ respects $s \approx t\ [\varphi]$ *if γ respects φ and $Var(s) \cup Var(t) \subseteq Dom(\gamma)$. We say γ is a* ground constructor substitution *if all $\gamma(x)$ 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).

To prove that an equation is an inductive theorem, we consider eight inference rules. Four of them originate in [19]; three are based on extensions [1,5,21]. The

[3] A more standard definition of this propertywould be that for every defined or calculation symbol f and suitable ground constructor terms $s_1, \ldots, s_n$ the term $f(s_1, \ldots, s_n)$ reduces. As observed in Appendix A, this definition is equivalent.

last, Generalization, is new; it essentially offers a shortcut for lemma equations. All these rules modify a triple $(\mathcal{E}, \mathcal{H}, b)$, called a *proof state*. Here, $\mathcal{E}$ is a set of equations, $\mathcal{H}$ a set of constrained rewrite 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 three *core* inference rules from [19] are adapted to LCTRSs as follows.

Simplification 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,[4] then we may derive:

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

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

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

Expansion Let $\mathit{Expd}(s, t, \varphi, p)$ be a set containing, for all rules $\ell \to r[\psi] \in \mathcal{R}$[5] such that ℓ is unifiable with $s_{|p}$ with most general unifier γ and $\varphi\gamma \wedge \psi\gamma$ is (or may be)[6] satisfiable, an equation $s' \approx t'\ [\varphi']$ where $s[\ell]_p\gamma \approx t\gamma\ [(\varphi\gamma) \wedge (\psi\gamma)] \to_{\mathcal{R}} s' \approx t'\ [\varphi']$ with rule $\ell \to r\ [\psi]$ at position $1 \cdot p$. Here, as in Simplification, $\approx$ is seen as a fresh constructor for the purpose of constrained term reduction. Intuitively, *Expd* generates all resulting equations if a ground constructor instance of $s \approx t\ [\varphi]$ is reduced at position p of s.
Now, if p is a position of s such that $s_{|p}$ is *basic* (i.e., $s_{|p} = f(s_1, \ldots, s_n)$ with f a defined symbol and all s_i constructor terms), we may derive:

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

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

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

Note that, if $\to_{\mathcal{R}}$ is non-deterministic (which may happen when considering irregular rules), we can choose how to build *Expd*.

These inference rules differ from [19] in several ways. Partially, this is because we consider LCTRSs rather than plain TRSs, and have to handle the constraints: hence we use constrained reduction rather than normal reduction in Simplification, and include an unsatisfiability case in Deletion.

[4] It is not enough if $s\,[\varphi] \to_{\mathcal{R}} u\,[\psi]$: when reducing constrained terms, we may manipulate unused variables at will, which causes problems if they are used in t. For example, $f(x + 0)\,[x > y] \sim f(x + 0)\,[z = x + 0] \to_{\mathtt{base}} f(z)\,[z = x + 0] \sim f(x)\,[y < x]$, but we do not want to replace an equation $f(x + 0) \approx g(y)\ [x > y]$ by $f(x) \approx g(y)\ [x < y]$!

[5] Here, we assume that the variables in the rules are distinct from the ones in s, t, φ.

[6] Although we do not *have* to include equations in $\mathit{Expd}(s, t, \varphi, p)$ which correspond to rules that give an unsatisfiable constraint, it is sometimes convenient to postpone the satisfiability check; if this creates additional equations with unsatisfiable constraints, they can afterwards be removed with Deletion anyway.

In Expansion, we have made more structural changes; our definition also differs from the corresponding rules in [5,21], where the method is defined for different styles of constrained rewriting.

To start, we use constrained reduction, whereas the authors of [19,5,21] use direct instantiation (e.g. $Expd(s,t,p)$ contains elements $s[r\gamma]_p \approx t$ when $\ell \to r \in \mathcal{R}$ and $s_{|p}$ unifies with ℓ with most general unifier γ). This was changed to better handle irregular rules, especially those where the right-hand side introduces fresh variables, i.e. $\ell \to r\ [\varphi]$ where $Var(r) \cap Var(\varphi) \not\subseteq Var(\ell)$. Such rules occur for example in transformed iterative functions where variables are declared but not immediately initialized. The alternative formulation of $\mathcal{R}$ in Section 5, which is essential for our lemma generalization technique, also uses such irregular rules.

Second, the case where no rule is added is new. This is needed to allow progress in cases when adding the rule might cause loss of termination. It somewhat corresponds to, but is strictly stronger than, Case-Simplify in [5].

The next two inference rules are adapted from both [21] and [5]. Most importantly, they provide a link between the equation part $s \approx t$ and the constraint.

EQ-deletion If all $s_i, t_i \in \mathcal{T}erms(\Sigma_{theory}, Var(\varphi))$, then we can derive:

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

$C[\,]$ is allowed to contain symbols in Σ_{theory}. Intuitively, if $\bigwedge_{i=1}^{n} s_i = t_i$ holds, then $C[s_1,\ldots,s_n]\gamma \leftrightarrow^*_{\mathcal{R}_{\mathtt{calc}}} C[t_1,\ldots,t_n]\gamma$ and thus, we are done. We exclude this case from the equation by adding $\neg(\bigwedge_{i=1}^{n} s_i = t_i)$ to the constraint. This inference rule originates in [21] and can, in combination with Deletion, be seen as a generalized variant of Theory$_\top$ in [5].

Disprove If $\vdash s : \iota$ and one of the following holds:

- $s, t \in \mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$, ι is a theory sort,[7] and $\varphi \wedge s \neq t$ is satisfiable;
- $s = f(\overrightarrow{s})$ and $t = g(\overrightarrow{t})$ with f, g distinct constructors;
- s is a variable not in $Var(\varphi)$, φ is satisfiable, there are at least two different constructors with output sort ι and either t is a variable distinct from s, or t has the form $f(\overrightarrow{s})$ with f a constructor.

Then we may derive:

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

The first case of this rule ($s, t \in \mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$) corresponds to Theory$_\top$ in [5]; the latter two cases are new. Note that here we require completeness.

Where [5] and [21] focus on systems which only have theory symbols and defined symbols (so where most of the reasoning can be moved to the constraint), in the LCTRS setting we are also interested in non-theory constructors, such as error_f and return_f. In order to support reasoning about systems where such constructors are admitted, we add one further rule:

[7] Note that this (only) excludes the case where s and t are both non-logical variables.

CONSTRUCTOR If f is a constructor, we can derive:

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

The CONSTRUCTOR rule originates in [1], where it is called POSITIVE DECOMPOSITION. It is used to split up a complicated equation into smaller problems.

Finally, let us consider the last inference rule from [19], and a final new rule.

POSTULATE For any set of equations $\mathcal{E}'$, we can derive:

$$(\mathcal{E}, \mathcal{H}, b) \vdash_{\mathtt{ri}} (\mathcal{E} \cup \mathcal{E}', \mathcal{H}, \text{INCOMPLETE})$$

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

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

The POSTULATE rule allows us to add additional equations (although at the price of losing completeness: we cannot conclude non-equivalence because we added an unsound equation). The reason for doing so is that in proving that the equations in $\mathcal{E}'$ are inductive theorems, we may obtain additional induction rules. These rules can then be used to simplify the equations of $\mathcal{E}$.

Since complex theorems typically require more than straight induction (both in the current setting and in mathematical proofs in general), the generation of suitable *lemma equations* $\mathcal{E}'$ is not only part, but even at the heart, of inductive theorem proving. Consequently, this subject has been extensively investigated [3,10,11,17,22,23], and a large variety of lemma generation techniques exist.

The new GENERALIZATION rule has a similar purpose: a more general equation is sometimes easier to handle than the original, as it gives stronger induction rules. This inference rule is not *necessary*: we could alternatively add $s' \approx t' \ [\psi]$ using POSTULATE: induction rules which are useful for $s' \approx t' \ [\psi]$ will also help for $s \approx t \ [\varphi]$. By generalizing instead, we just avoid some extra steps, which is useful because the lemma generation technique introduced in this paper is a generalization technique. This inference rule can be seen as a combination of POSTULATE and the SUBSUMPTION rule in [1].

There are many other potential inference rules we could consider, as various extensions of the base method have been studied in the literature (see e.g. [1]). For now, we stick to these eight rules, as they suffice on our example set.

4.3 Soundness and Completeness of Rewriting Induction

These inference rules are used for *rewriting induction* by the following theorem:

Theorem 18. *Let an LCTRS with rules $\mathcal{R}$ and signature Σ, satisfying the restrictions from Section 4.1, be given.*

Let $\mathcal{E}$ be a finite set of equations and let flag = COMPLETE *if $\mathcal{R}$ is* confluent *(that is, if $s \leftrightarrow^*_{\mathcal{R}} t$, then there is some u with $s \to^*_{\mathcal{R}} u$ and $t \to^*_{\mathcal{R}} u$) and*

flag = INCOMPLETE *otherwise. 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* $\mathcal{R}$*. If* $(\mathcal{E}, \emptyset, \mathit{flag}) \vdash_{\mathtt{ri}} \cdots \vdash_{\mathtt{ri}} \bot$*, then there is some equation in* $\mathcal{E}$ *which is not an inductive theorem of* $\mathcal{R}$*.*

In Section 4.4, we will see how we can use this result in practice. Now, let us first prove the theorem. To this end, we follow the proof method of [21], which builds on the original proof idea in [19]. Given a set $\mathcal{E}$ of equations, we will write $\leftrightarrow_{\mathcal{E}}$ for the *derivation relation of* $\mathcal{E}$, defined as follows:

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

The proof is split up into several auxiliary lemmas. To start, we observe:

Lemma 19. *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* $s \leftrightarrow_{\mathcal{R}}^* t$*).*

Proof. Suppose $\leftrightarrow_{\mathcal{E}} \subseteq \leftrightarrow_{\mathcal{R}}^*$ on ground terms. Then every equation in $\mathcal{E}$ is an inductive theorem: if $s \approx t\ [\varphi] \in \mathcal{E}$ and γ is a ground constructor substitution that respects this equation, then $s\gamma$ and $t\gamma$ are ground terms, and since obviously $s\gamma \leftrightarrow_{\mathcal{E}} t\gamma$ (with empty C), by assumption $s\gamma \leftrightarrow_{\mathcal{R}}^* t\gamma$.

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

Lemma 20 ([19,12]). *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*
- $\to_2 \subseteq (\to_1 \cdot \to_2^* \cdot \leftarrow_2^*)$.

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 $x \to_2^* x$ is clear, we suppose $x \to_2 y \to_2^* z$. Then, it follows from $\to_2 \subseteq (\to_1 \cdot \to_2^* \cdot \leftarrow_2^*)$ that there are some a, b such that $x \to_1 a \to_2^* b \leftarrow_2^* y$. Since $\to_1 \subseteq \to_2$ (i.e., $x \to_2 a$), we can apply the induction hypothesis both on a and on y, so $a \leftrightarrow_1^* b \leftrightarrow_1^* y$ and $y \leftrightarrow_1^* z$. Therefore, $x \leftrightarrow_1^* z$. □

Lemma 21. *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 the rules. Then:*

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

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

   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 $Expd(s,t,\varphi,p)$ and any ground constructor substitution δ which 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 $s_i\gamma$ must be ground constructor terms. Since f is a defined symbol, quasi-reductivity provides that $f(\overrightarrow{s}\gamma)$ reduces and, since all its strict subterms are constructor terms, the only possible reduction is at the root. Thus, there is a rule $\ell \to r\ [\psi]$ and a substitution δ which respects ψ such that $s\gamma = (s\gamma)[\ell\delta]_p$. Since the rule variables are distinct from the ones in the equation, we can assume that δ is an extension of γ, so $s\gamma = s[l]_p\delta$. Clearly, both $\varphi\delta$ and $\psi\delta$ evaluate to $\top$.

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

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

Consider the "term" $s\delta \approx t\delta$. This is an instance of the first constrained term in this reduction, so by Theorem 6, this "term" reduces at position $1\cdot p$ to $s'\delta'' \approx t'\delta''$ for some substitution δ'' which respects φ'. As the reduction happens inside $s\delta$, we see that $t\delta = t'\delta''$. Thus, $s\gamma = s\delta \to_{\mathcal{R}} s'\delta'' \leftrightarrow_{Expd(s,t,\varphi,p)} t'\delta'' = t\delta = t\gamma$.

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 7, we can find a substitution ϵ which respects $\varphi\gamma \wedge \psi\gamma$ such that $s\gamma\epsilon \approx t\gamma\epsilon \to_{\mathcal{R}} s'\delta \approx t'\delta$ at position $1 \cdot p$. Since the reduction takes place in the left part of $\approx$, we have $t\gamma\epsilon = t'\delta$ and $s\gamma\epsilon \to_{\mathcal{R}} s'\delta$. We are done if also $s\gamma\epsilon \leftrightarrow_{s\approx t\ [\varphi]} t\gamma\epsilon$. But this is clearly the case, since the composed substitution $\epsilon \circ \gamma$ respects φ (as $(\varphi\gamma \wedge \psi\gamma)\epsilon$ implies $\varphi\gamma\epsilon$). □

Lemma 22. *Suppose that $(\mathcal{E}, \mathcal{H}, b)\ \vdash_{\mathtt{ri}}\ (\mathcal{E}', \mathcal{H}', b')$. Then,*

$$\leftrightarrow_{\mathcal{E}} \subseteq (\to^*_{\mathcal{R}\cup\mathcal{H}'} \cdot (\leftrightarrow_{\mathcal{E}'} \cup =) \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'})$$

on ground terms.

Proof. It suffices to show that $\leftrightarrow_{\mathcal{E}\setminus\mathcal{E}'} \subseteq (\to^*_{\mathcal{R}\cup\mathcal{H}'} \cdot (\leftrightarrow_{\mathcal{E}'} \cup =) \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'})$ on ground terms. For all inference rules either $\mathcal{E} \setminus \mathcal{E}' = \emptyset$ or we can write $\mathcal{E} \setminus \mathcal{E}' = \{s \simeq t\ [\varphi]\}$. We make a case analysis depending on which inference rule is applied for $\vdash_{\mathtt{ri}}$.

- (Simplification). Suppose that $s \simeq t\ [\varphi]$ is replaced by $u \approx t\ [\psi]$ with $s \approx t\,[\varphi] \to_{\mathcal{R}\cup\mathcal{H}} u \approx t\,[\psi]$. Suppose that $C[s\gamma] \leftrightarrow_{\{s\simeq t\ [\varphi]\}} C[t\gamma]$, where γ is a substitution which respects φ. It follows from Theorem 6 that $s\gamma \approx t\gamma \to_{\mathcal{R}\cup\mathcal{H}} u\delta \approx t\delta$ where δ is a substitution which 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). Consider the case that $s = t$. In this case we have that $\leftrightarrow_{\mathcal{E}\setminus\mathcal{E}'}$ is the identity. Otherwise, φ is unsatisfiable. In this case, $s \simeq t\ [\varphi]$ is never used, i.e., $\leftrightarrow_{\mathcal{E}\setminus\mathcal{E}'} = \emptyset$.
- (Expansion). Suppose $C[s\gamma] \leftrightarrow_{s\simeq t\ [\varphi]} C[t\gamma]$, where γ respects $s \simeq t\ [\varphi]$; as we only consider ground terms, we can safely assume that all $\gamma(x)$ are ground. Noting that by quasi-reductivity and termination every term reduces to a ground constructor term, let δ be a substitution where each $\delta(x)$ is a constructor term such that $\gamma(x) \to^*_{\mathcal{R}} \delta(x)$. It follows from Lemma 21 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 $s_1, t_1, \ldots, s_n, t_n$ are terms in $\mathcal{T}erms(\Sigma_{theory}, Var(\varphi))$. Then a ground substitution γ which respects φ and whose domain contains all variables in the 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 $q \leftrightarrow_{s\simeq t[\varphi]} u$ for ground q, u. Then $q = D[C[s_1, \ldots, s_n]]\gamma$ and $u = D[C[t_1, \ldots, t_n]]\gamma$ for some ground γ which respects φ, or $u = D[C[\overrightarrow{t}]]\gamma$ and $q = D[C[\overrightarrow{s}]]\gamma$; as the latter situation is symmetric, we will not discuss it. In the former, $q \to^*_{\mathcal{R}} D\gamma[C\gamma[v_1, \ldots, v_n]]$ 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 $q \to^*_{\mathcal{R}} \cdot \leftrightarrow_{\mathcal{E}'} \cdot \leftarrow^*_{\mathcal{R}} u$.
- (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 which respects φ. Since $\mathcal{E}'$ contains all equations $s_i \approx t_i\ [\varphi]$, we have $C[s\gamma] = C[f(s_1\gamma, \ldots, s_n\gamma)] \leftrightarrow_{\mathcal{E}'} C[f(t_1\gamma, \ldots, s_n\gamma)] \leftrightarrow_{\mathcal{E}'} \ldots \leftrightarrow_{\mathcal{E}'} C[f(t_1\gamma, \ldots, t_n\gamma)] = C[t\gamma]$.
- (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 which respects φ. Then there exists a substitution δ which respects ψ such that $C[s\gamma] = C[s'\delta] \leftrightarrow_{\{s'\approx t'[\varphi]} C[t'\delta] = C[t\gamma]$. □

Lemma 23. *Suppose that* $(\mathcal{E}, \mathcal{H}, \mathit{flag}) \vdash_{\mathtt{ri}} (\mathcal{E}', \mathcal{H}', \mathit{flag}')$. *If*

$$\to_{\mathcal{R}\cup\mathcal{H}} \subseteq (\to_{\mathcal{R}} \cdot \to^*_{\mathcal{R}\cup\mathcal{H}} \cdot (\leftrightarrow_{\mathcal{E}} \cup =) \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}})$$

on ground terms, then

$$\to_{\mathcal{R}\cup\mathcal{H}'} \subseteq (\to_{\mathcal{R}} \cdot \to^*_{\mathcal{R}\cup\mathcal{H}'} \cdot (\leftrightarrow_{\mathcal{E}'} \cup =) \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}'})$$

on ground terms.

Proof. It suffices to consider the case that Expansion is applied (for the other cases, we use Lemma 22). Suppose that $s \to_{\mathcal{H}' \setminus \mathcal{H}} t$. Then, it follows from Lemma 21 that $s\ (\to_{\mathcal{R}}^* \cdot \to_{\mathcal{R}} \cdot \leftrightarrow_{\mathcal{E}'} \cdot \leftarrow_{\mathcal{R}}^*)\ t$, and hence:

$$s\ \ (\to_{\mathcal{R}} \cdot \to_{\mathcal{R} \cup \mathcal{H}'}^* \cdot (\leftrightarrow_{\mathcal{E}'} \cup =) \cdot \leftarrow_{\mathcal{R} \cup \mathcal{H}'}^*)\ \ t$$

□

Lemma 24. *Suppose that* $(\mathcal{E}, \mathcal{H}, \mathit{flag}) \vdash_{\mathtt{ri}} \cdots \vdash_{\mathtt{ri}} (\mathcal{E}', \mathcal{H}', \mathit{flag}')$*. Then, all of the following hold:*

1. $\leftrightarrow_{\mathcal{E}} \subseteq (\to_{\mathcal{R} \cup \mathcal{H}'}^* \cdot (\leftrightarrow_{\mathcal{E}'} \cup =) \cdot \leftarrow_{\mathcal{R} \cup \mathcal{H}'}^*)$ *on ground terms,*
2. *if* $\to_{\mathcal{R} \cup \mathcal{H}} \subseteq (\to_{\mathcal{R}} \cdot \to_{\mathcal{R} \cup \mathcal{H}}^* \cdot (\leftrightarrow_{\mathcal{E}} \cup =) \cdot \leftarrow_{\mathcal{R} \cup \mathcal{H}}^*)$ *on ground terms, then*

$$\to_{\mathcal{R} \cup \mathcal{H}'} \subseteq (\to_{\mathcal{R}} \cdot \to_{\mathcal{R} \cup \mathcal{H}'}^* \cdot (\leftrightarrow_{\mathcal{E}'} \cup =) \cdot \leftarrow_{\mathcal{R} \cup \mathcal{H}'}^*)$$

   on ground terms, and
3. *if* $\mathcal{R} \cup \mathcal{H}$ *is terminating, then so is* $\mathcal{R} \cup \mathcal{H}'$*.*

Proof.

1. By induction on the number of $\vdash_{\mathtt{ri}}$ steps. The base case is evident, so suppose $(\mathcal{E}, \mathcal{H}, \mathit{flag}) \vdash_{\mathtt{ri}} (\mathcal{E}_1, \mathcal{H}_1, \mathit{flag}_1) \vdash_{\mathtt{ri}}^* (\mathcal{E}', \mathcal{H}', \mathit{flag}')$. It follows from Lemma 22 that

$$\leftrightarrow_{\mathcal{E}} \subseteq \big(\to_{\mathcal{R} \cup \mathcal{H}_1}^* \cdot (\leftrightarrow_{\mathcal{E}_1} \cup =) \cdot \leftarrow_{\mathcal{R} \cup \mathcal{H}_1}^*\big)$$

   and by the induction hypothesis we have:

$$\leftrightarrow_{\mathcal{E}_1} \subseteq (\to_{\mathcal{R} \cup \mathcal{H}'}^* \cdot (\leftrightarrow_{\mathcal{E}'} \cup =) \cdot \leftarrow_{\mathcal{R} \cup \mathcal{H}'}^*)$$

   on ground terms. It follows from $\mathcal{H}_1 \subseteq \mathcal{H}'$ that

$$\leftrightarrow_{\mathcal{E}} \subseteq (\to_{\mathcal{R} \cup \mathcal{H}'}^* \cdot (\leftrightarrow_{\mathcal{E}_1} \cup =) \cdot \leftarrow_{\mathcal{R} \cup \mathcal{H}'}^*)$$

   By replacing $\leftrightarrow_{\mathcal{E}_1}$ with $(\to_{\mathcal{R} \cup \mathcal{H}'}^* \cdot (\leftrightarrow_{\mathcal{E}'} \cup =) \cdot \leftarrow_{\mathcal{R} \cup \mathcal{H}'}^*)$, we have that

$$\leftrightarrow_{\mathcal{E}} \subseteq \left(\to_{\mathcal{R} \cup \mathcal{H}'}^* \cdot \left(\begin{array}{c} (\to_{\mathcal{R} \cup \mathcal{H}'}^* \cdot (\leftrightarrow_{\mathcal{E}'} \cup =) \cdot \leftarrow_{\mathcal{R} \cup \mathcal{H}'}^*) \\ \cup \\ = \end{array} \right) \cdot \leftarrow_{\mathcal{R} \cup \mathcal{H}'}^* \right)$$

   and hence $\leftrightarrow_{\mathcal{E}} \subseteq (\to_{\mathcal{R} \cup \mathcal{H}'}^* \cdot (\leftrightarrow_{\mathcal{E}'} \cup =) \cdot \leftarrow_{\mathcal{R} \cup \mathcal{H}'}^*)$.
2. Trivial by applying Lemma 23 step by step.
3. Trivial by definition of Expansion, the only inference rule adding to $\mathcal{H}$. □

This leaves the way free for the first part of Theorem 18.

Lemma 25. *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 24 that:

– $\leftrightarrow_{\mathcal{E}} \subseteq \to^*_{\mathcal{R}\cup\mathcal{H}} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}}$ 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 20 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 19. □

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

Lemma 26. *If $\mathcal{R}$ is confluent and $(\mathcal{E}, \mathcal{H}, \text{COMPLETE}) \vdash_{\texttt{ri}} \bot$, then $\mathcal{E}$ contains an equation $s \approx t\ [\varphi]$ which 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 well-known property of confluence is that if $w \leftrightarrow^*_{\mathcal{R}} q$, then necessarily $w{\downarrow_{\mathcal{R}}} = q{\downarrow_{\mathcal{R}}}$. So, it suffices to prove that for some $s \approx t\ [\varphi]$ there is a ground constructor substitution γ which 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 one of the following holds:

1. $s, t \in \mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$ with $\varphi \wedge s \neq t$ satisfiable. That is, 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 with $\to_{\texttt{calc}}$. Since values are always in normal form by the definition of a rule, we are done.
2. $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_n)$ with f and g different constructors, and φ is satisfiable. That is, there is a substitution δ mapping all variables in φ to values such that $\varphi\delta$ is valid. Let γ be some extension of δ which 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 $f(\overrightarrow{s}){\downarrow_{\mathcal{R}}} = (s\gamma){\downarrow_{\mathcal{R}}} \neq (t\gamma){\downarrow_{\mathcal{R}}} = g(\overrightarrow{t}{\downarrow_{\mathcal{R}}})$.
3. $s : \iota$ is a variable not occurring in φ and φ is satisfiable and 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 $t = f(\overrightarrow{t})$, 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, $(s\gamma){\downarrow_{\mathcal{R}}} \neq (t\gamma){\downarrow_{\mathcal{R}}}$, and $\varphi\gamma$ is valid. □

Lemma 27. *If $\mathcal{R}$ 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. Suppose $\leftrightarrow_{\mathcal{E}} \cup \leftrightarrow_{\mathcal{H}} \subseteq \leftrightarrow^*_{\mathcal{R}}$ on ground terms. Then certainly $\leftrightarrow_{\mathcal{H}'} \subseteq \leftrightarrow^*_{\mathcal{R}}$: whichever rule was applied, each element in $\mathcal{H}'$ is either also in $\mathcal{H}$ or (in the case of EXPANSION) corresponds to an equation in $\mathcal{E}$.

So let $s \approx t\ [\varphi] \in \mathcal{E}' \setminus \mathcal{E}$; we must see that $\leftrightarrow_{\{s\approx t\ [\varphi]\}} \subseteq \leftrightarrow^*_{\mathcal{R}}$. By Lemma 19, it suffices if for all ground constructor substitutions γ which 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 $s' \approx t'\ [\varphi'] \to_{\mathcal{R}\cup\mathcal{H}} s \approx t\ [\varphi]$ at position $1 \cdot p$. By Theorem 7, we can find δ which respects φ' such that $s'\delta \to_{\mathcal{R}\cup\mathcal{H}} s\gamma$ at position p and $t'\delta = t\gamma$. As $\to_{\mathcal{R}} \cup \to_{\mathcal{H}} \subseteq \leftrightarrow^*_{\mathcal{R}}$ 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 $s \approx t\ [\varphi] \in \mathit{Expd}(s', t', \varphi', p)$ for some p. By Lemma 21(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 $\varphi = \varphi' \wedge \neg(s_1 = t_1 \wedge \ldots \wedge s_n = t_n)$, and $s = C[s_1, \ldots, s_n],\ t = C[t_1, \ldots, t_n]$ for some $C, \overrightarrow{s}, \overrightarrow{t}$. Since any substitution which 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). □

Now we are ready for the second part of Theorem 18:

Lemma 28. *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 $\mathcal{E}$ which is not an inductive theorem of $\mathcal{R}$.*

Proof. If $(\mathcal{E}, \emptyset, \text{COMPLETE}) = (\mathcal{E}_1, \mathcal{H}_1, \mathit{flag}_1) \vdash_{\mathtt{ri}} \ldots \vdash_{\mathtt{ri}} (\mathcal{E}_n, \mathcal{H}_n, \mathit{flag}_n) \vdash_{\mathtt{ri}} \bot$, then we easily see that $\mathit{flag}_n = \text{COMPLETE}$ for all n. By Lemma 26, $\mathcal{E}_n$ contains an equation $s \approx t\ [\varphi]$ which is not an inductive theorem. Then $\leftrightarrow_{\mathcal{E}_n} \not\subseteq \leftrightarrow^*_{\mathcal{R}}$ on ground terms. By Lemma 27 and induction on n, this means that $\leftrightarrow_{\mathcal{E}} \cup \leftrightarrow_{\emptyset} \not\subseteq \leftrightarrow^*_{\mathcal{R}}$ on ground terms, so by Lemma 19, not all $e \in \mathcal{E}$ are inductive theorems. □

Thus we obtain:

Proof of Theorem 18. Immediately by Lemmas 25 and 28. □

4.4 An Illustrative Example

To show how the method works, consider $\mathcal{R}_{\mathsf{sum}}$, the LCTRS with the two correct implementations of the motivating example in Section 1.1.

$$
\begin{array}{lll}
(1) & \mathsf{sum1}(arr, n) \to \mathsf{u}(arr, n, 0, 0) & \\
(2) & \mathsf{u}(arr, n, ret, i) \to \mathsf{error} & [i < n \wedge (i < 0 \vee i \geq \mathsf{size}(arr))] \\
(3) & \mathsf{u}(arr, n, ret, i) \to \mathsf{u}(arr, n, ret + \mathsf{select}(arr, i), i + 1) & \\
 & & [i < n \wedge 0 \leq i < \mathsf{size}(arr)] \\
(4) & \mathsf{u}(arr, n, ret, i) \to \mathsf{return}(arr, ret) & [i \geq n] \\
 & & \\
(5) & \mathsf{sum4}(arr, k) \to \mathsf{return}(arr, 0) & [k \leq 0] \\
(6) & \mathsf{sum4}(arr, k) \to \mathsf{error} & [k - 1 \geq \mathsf{size}(arr)] \\
(7) & \mathsf{sum4}(arr, k) \to \mathsf{w}(\mathsf{select}(arr, k - 1), \mathsf{sum4}(arr, k - 1)) & \\
 & & [0 \leq k - 1 < \mathsf{size}(arr)] \\
(8) & \mathsf{w}(n, \mathsf{error}) \to \mathsf{error} & \\
(9) & \mathsf{w}(n, \mathsf{return}(a, r)) \to \mathsf{return}(a, n + r) &
\end{array}
$$

Here, note that a constraint $x \leq y < b$ should be read as: $x \leq y \wedge y < b$.

We want to see that these two implementations are equivalent, at least when the input makes sense, so the given length is at least 0 and does not exceed the array size. This is the case if the following equation is an inductive theorem:

$$\text{(A)} \quad \mathsf{sum1}(a,k) \approx \mathsf{sum4}(a,k) \ [0 \leq k \leq \mathsf{size}(a)]$$

Thus, we start the procedure with $(\{\text{(A)}\}, \emptyset, \textsc{complete})$. Since the aim in this case is to prove (rather than disprove) equivalence, we will henceforth omit the completeness flag. To start, we apply Simplification and obtain:

$$(\{\text{(B)} \quad \mathsf{u}(a,k,0,0) \approx \mathsf{sum4}(a,k) \ [0 \leq k \leq \mathsf{size}(a)]\}, \emptyset)$$

None of Simplification, EQ-deletion, Deletion and Constructor is applicable, so we apply Expansion to the right-hand side of (B) at the root. Since $k \leq \mathsf{size}(a)$ and $k - 1 \geq \mathsf{size}(a)$ cannot both hold, the error rule leads to an unsatisfiable constraint. Therefore, this step only gives two new equations:

$$\left(\left\{ \begin{array}{l} \text{(C)} : \mathsf{return}(a,0) \approx \mathsf{u}(a,k,0,0) \ [0 \leq k \leq \mathsf{size}(a) \wedge k \leq 0] \\ \text{(D)} : \quad \mathsf{w}(\mathsf{select}(a,k-1), \mathsf{sum4}(a,k-1)) \approx \mathsf{u}(a,k,0,0) \\ \qquad\qquad [0 \leq k \leq \mathsf{size}(a) \wedge 0 \leq k-1 < \mathsf{size}(a)] \end{array} \right\}, \{ \text{(B}^{-1}\text{)} \} \right)$$

Here, $\text{(B}^{-1}\text{)}$ should be read as the rule generated from the inverse of (B), so:

$$\text{(B}^{-1}\text{)} \ \mathsf{sum4}(a,k) \to \mathsf{u}(a,k,0,0) \ [0 \leq k \leq \mathsf{size}(a)]$$

We use Simplification with rule (4) to reduce (C) to $\mathsf{return}(a,0) \approx \mathsf{return}(a,0)$ $[\ldots]$, which we quickly delete. $(\{\text{(D)}\},\{\text{(B}^{-1}\text{)}\})$ remains. Simplifying the right-hand side of (D) with rule (3), we obtain $(\{\text{(E)}\},\{\text{(B}^{-1}\text{)}\})$, with:

$$\text{(E)} : \mathsf{w}(\mathsf{select}(a,k-1), \mathsf{sum4}(a,k-1)) \approx \mathsf{u}(a,k,0+\mathsf{select}(a,0),0+1)$$
$$[0 \leq k \leq \mathsf{size}(a) \wedge 0 \leq k-1 < \mathsf{size}(a)]$$

Next we use Simplification with the calculation rules. As these rules are irregular, this requires some care. There are three standard ways to do this:

- if $s \to_{\mathtt{calc}} t$ then $s\,[\varphi] \to_{\mathcal{R}} t\,[\varphi]$, e.g. $\mathsf{f}(0+1) \approx r \ [\varphi]$ reduces to $\mathsf{f}(1) \approx r \ [\varphi]$;
- a calculation can be replaced by a fresh variable, which is defined in the constraint, e.g. $\mathsf{f}(x+1) \approx r \ [\varphi]$ reduces to $\mathsf{f}(y) \approx r \ [\varphi \wedge y = x+1]$;
- a calculation *already* defined in the constraint can be replaced by the relevant variable, e.g. $\mathsf{f}(x+1) \approx r \ [\varphi \wedge y = x+1]$ reduces to $\mathsf{f}(y) \approx r \ [\varphi \wedge y = x+1]$.

These ways are not functionally different; if an equation e reduces both to e_1 and e_2 with a calculation at the same position, then it is easy to see that $e_1 \sim e_2$.

We can do more: recall that, by the definition of constrained term reduction, we can rewrite a constraint φ with variables $\overrightarrow{x}, \overrightarrow{y}$ in a constrained term $s\,[\varphi]$, to any constraint ψ over $\overrightarrow{x}, \overrightarrow{z}$ such that $\exists \overrightarrow{y}[\varphi]$ is equivalent to $\exists \overrightarrow{z}[\psi]$ (if $\mathit{Var}(s) = \{\overrightarrow{x}\}$). We primarily use this observation to write the constraint in

a simpler form after Simplification or Expansion, for instance by removing redundant clauses.

Using six more Simplification steps with the calculation rules on (E), and writing the constraint in a simpler form, we obtain:

$$\left(\left\{\begin{array}{ll}(\mathrm{F}): & \mathsf{w}(n,\mathsf{sum4}(a,k')) \approx \mathsf{u}(a,k,r,1) \;\; [k' = k-1 \wedge \\ & 0 \le k' < \mathsf{size}(a) \wedge n = \mathsf{select}(a,k') \wedge r = 0 + \mathsf{select}(a,0)]\end{array}\right\}, \{\,(\mathrm{B}^{-1})\,\}\right)$$

Then, using Simplification with the induction rule (B^{-1}):

$$\left(\left\{\begin{array}{ll}(\mathrm{G}): & \mathsf{w}(n,\mathsf{u}(a,k',0,0)) \approx \mathsf{u}(a,k,r,1) \;\; [k' = k-1 \wedge \\ & 0 \le k' < \mathsf{size}(a) \wedge n = \mathsf{select}(a,k') \wedge r = 0 + \mathsf{select}(a,0)]\end{array}\right\}, \{\,(\mathrm{B}^{-1})\,\}\right)$$

Once more, our only choice (aside from generalizing or adding additional equations) is Expansion. We expand in the right-hand side, obtaining:

$$\left(\left\{\begin{array}{ll}(\mathrm{H}): & \mathsf{u}(a,k,r+\mathsf{select}(a,1),1+1) \approx \mathsf{w}(n,\mathsf{u}(a,k',0,0)) \\ & [k' = k-1 \wedge 0 \le k' < \mathsf{size}(a) \wedge n = \mathsf{select}(a,k') \wedge \\ & \quad r = 0 + \mathsf{select}(a,0) \wedge 1 < k \wedge 0 \le 1 < \mathsf{size}(a)] \\ (\mathrm{I}): & \mathsf{return}(a,r) \approx \mathsf{w}(n,\mathsf{u}(a,k',0,0)) \;\; [k' = k-1 \wedge \\ & 0 \le k' < \mathsf{size}(a) \wedge n = \mathsf{select}(a,k') \wedge r = 0 + \mathsf{select}(a,0) \\ & \quad \wedge\, 1 \ge k]\end{array}\right\}, \left\{\begin{array}{l}(\mathrm{B}^{-1}) \\ (\mathrm{G}^{-1})\end{array}\right\}\right)$$

We have again omitted the error rule, as the corresponding constraint is not satisfiable. For (I), the constraint implies that $k = 1$, so Simplification with rule (4) followed by (9) and some prettifying of the constraint, gives:

$$\mathsf{return}(a,r) \approx \mathsf{return}(a,n+0)\ [k' = 0 < \mathsf{size}(a) \wedge n = \mathsf{select}(a,k') \wedge r = \mathsf{select}(a,0)]$$

Using EQ-deletion, we can reduce this to an equation with a constraint $k' = 0 < \mathsf{size}(a) \wedge n = \mathsf{select}(a,k') \wedge r = \mathsf{select}(a,0) \wedge \neg(r = n+0)$, which is unsatisfiable. The resulting equation is removed with Deletion.

We continue with $(\{(\mathrm{H})\},\{(\mathrm{B}^{-1}),(\mathrm{G}^{-1})\})$. After applying Simplification with (3) and calculation rules a few times, we have $(\{(\mathrm{J})\},\{(\mathrm{B}^{-1}),(\mathrm{G}^{-1})\})$:

$$\begin{array}{ll}(\mathrm{J}): & \mathsf{u}(a,k,r_1,2) \approx \mathsf{w}(n,\mathsf{u}(a,k',r,1)) \\ & [k' = k-1 \wedge 0 \le k' \wedge 1 < k \le \mathsf{size}(a) \wedge n = \mathsf{select}(a,k') \wedge \\ & \quad r = 0 + \mathsf{select}(a,0) \wedge r_1 = r + \mathsf{select}(a,1)]\end{array}$$

Here, we have used the third style of calculation simplification to reuse r.

Again we use Expansion, now on the left-hand side; two of the resulting equations are quickly deleted; for the third, repeated Simplification leads to:

$$\begin{array}{ll}(\mathrm{K}): & \mathsf{u}(a,k,r_2,3) \approx \mathsf{w}(n,\mathsf{u}(a,k',r_1,2)) \\ & [k' = k-1 \wedge 0 \le k' \wedge 2 < k \le \mathsf{size}(a) \wedge n = \mathsf{select}(a,k') \wedge \\ & \quad r = 0 + \mathsf{select}(a,0) \wedge r_1 = r + \mathsf{select}(a,1) \wedge r_2 = r_1 + \mathsf{select}(a,2)]\end{array}$$

But now a pattern starts to arise. If we continue like this, simplifying as long as we can, and then using whichever of the other core rules is applicable, we get:

$$\begin{array}{l}(\mathrm{L}):\ \mathsf{u}(a,k,r_3,4) \approx \mathsf{w}(n,\mathsf{u}(a,k',r_2,3))\ [k' = k-1 \wedge 3 < k \le \mathsf{size}(a) \wedge \ldots] \\ (\mathrm{M}):\ \mathsf{u}(a,k,r_4,5) \approx \mathsf{w}(n,\mathsf{u}(a,k',r_3,4))\ [k' = k-1 \wedge 4 < k \le \mathsf{size}(a) \wedge \ldots]\end{array}$$

That is, 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 [23]). Yet the previous induction rules never apply to the new equation. We will need to generalize the equation to finish this proof.

So, consider the following equation (we will say more about it in Section 5):

$$\text{(N): } \mathsf{u}(a,k,r,i) \approx \mathsf{w}(n,\mathsf{u}(a,k',r',i'))\ [k' = k-1 \wedge 0 \leq i' < k \leq \mathsf{size}(a) \wedge i' = i-1 \wedge r = r' + \mathsf{select}(a,i') \wedge n = \mathsf{select}(a,k')]$$

It is easy to see that (K) is an instance of (N); we apply Generalization and continue with $(\{\text{(N)}\},\{(\text{B}^{-1}),(\text{G}^{-1}),(\text{J})\})$. Using Expansion, we obtain:

$$\left(\left\{\begin{array}{ll} \text{(O):} & \mathsf{error} \approx \mathsf{w}(n,\mathsf{u}(a,k',r',i'))\ \ [k' = k-1 \wedge \\ & 0 \leq i' < k \leq \mathsf{size}(a) \wedge i' = i-1 \wedge r = r' + \mathsf{select}(a,i') \wedge \\ & n = \mathsf{select}(a,k') \wedge i < k \wedge (i < 0 \vee i \geq \mathsf{size}(a))] \\ \text{(P):} & \mathsf{u}(a,k,r+\mathsf{select}(a,i),i+1) \approx \mathsf{w}(n,\mathsf{u}(a,k',r',i')) \\ & [k' = k-1 \wedge 0 \leq i' < k \leq \mathsf{size}(a) \wedge i' = i-1 \wedge r = r' + \\ & \mathsf{select}(a,i') \wedge n = \mathsf{select}(a,k')\ \wedge i < k\ \wedge 0 \leq i < \mathsf{size}(a)] \\ \text{(Q):} & \mathsf{return}(a,r) \approx \mathsf{w}(n,\mathsf{u}(a,k',r',i')) \\ & [k' = k-1 \wedge 0 \leq i' < k \leq \mathsf{size}(a) \wedge i' = i-1 \wedge \\ & r = r' + \mathsf{select}(a,i') \wedge n = \mathsf{select}(a,k') \wedge i \geq k] \end{array}\right\}, \left\{\begin{array}{c}(\text{B}^{-1})\\(\text{G}^{-1})\\(\text{J})\\(\text{N})\end{array}\right\}\right)$$

Here, we included the equation (O) for the error rule, because it might not be immediately obvious whether we can omit it. However, consider the constraint of (O). $i < 0$ cannot hold if we have both $0 \leq i'$ and $i' = i - 1$. Also $i \geq \mathsf{size}(a)$ cannot hold if we have both $i < k$ and $k \leq \mathsf{size}(a)$. Thus, this constraint is unsatisfiable; we quickly remove the equation with Deletion.

Consider (Q). The clauses $i \geq k$, $k' = k - 1$ and $i' = i - 1$ together give $i' \geq k'$, so we can simplify the right-hand side with rules (4) and (9), giving:

$$\mathsf{return}(a,r) \approx \mathsf{return}(a,n+r')\ [k' = k-1 \wedge 0 \leq i' < k \leq \mathsf{size}(a) \wedge i' = i-1 \wedge r = r' + \mathsf{select}(a,i') \wedge n = \mathsf{select}(a,k') \wedge i \geq k]$$

We apply EQ-deletion. Since the constraint implies that $i' = k'$ and therefore $r = r' + n$, we can also remove the resulting equation using Deletion.

Only (P) remains. We simplify this equation with the normal rules, giving:

$$\text{(Q)}\quad \mathsf{u}(a,k,r'',i'') \approx \mathsf{w}(n,\mathsf{u}(a,k',r,i))\ [k' = k-1 \wedge 0 \leq i' < k \leq \mathsf{size}(a) \wedge i' = i-1 \wedge r = r' + \mathsf{select}(a,i') \wedge n = \mathsf{select}(a,k') \wedge i < k \wedge 0 \leq i < \mathsf{size}(a) \wedge i'' = i+1 \wedge r'' = r + \mathsf{select}(a,i)]$$

But now note that the induction rule (N) applies! Because this rule is irregular, we have to be a little careful. We use the following substitution for the constrained reduction step: $\gamma = [a := a, k := k, r := r'', i := i'', n := n, k' := k', r' := r, i' := i]$. Note that this substitution also affects variables not occurring in the left-hand side of the rule! Using Simplification, the equation is reduced to $\mathsf{w}(n,\mathsf{u}(a,k',r,i)) \approx \mathsf{w}(n,\mathsf{u}(a,k',r,i))\ [\ldots]$, which is removed using Deletion.

Since we have succesfully reduced $(\{\text{(A)}\},\emptyset)$ to $(\emptyset,\mathcal{H})$ for some $\mathcal{H}$, the original equation (A) is an inductive theorem.

4.5 Correctness of String Functions

For another example, let us look at an assignment to implement `strlen`, a string function which operates on 0-terminated `char` arrays. Following Section 3.5, we limit interest to *static* arrays, so do not use pointer arithmetic. As `char` is a numeric data type, the LCTRS translation can implement this as integer arrays again (although using another underlying sort $\mathcal{I}_{\mathsf{char}}$ would make little difference).

The example function and its LCTRS translation are as follows:

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

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

Note that the overflow 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 $a[i] = 0$. This corresponds to the following equation being an inductive theorem:

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

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 instead use an additional symbol nonzero_until, and replace the $\forall i \in \{0, n-1\}[\mathsf{select}(x, i) \neq 0]$ part of the constraint by $\mathsf{nonzero\_until}(x, n)$.

Starting with ($\{(\mathrm{A})\}$,$\emptyset$), we first use Simplification using rule (1):

$$
\left(\left\{ \begin{array}{ll}
(\mathrm{A}) & \mathsf{u}(x, 0) \approx \mathsf{return}(n) \\
& [0 \leq n < \mathsf{size}(x) \land \forall i \in \{0, n-1\}[\mathsf{select}(x, i) \neq 0] \land \\
& \mathsf{select}(x, n) = 0]
\end{array} \right\}, \{\emptyset\} \right)
$$

We continue with Expansion, again on the left-hand side (as there is little we can do on the right-hand side at all); since the constraint implies that $0 < \mathsf{size}(x)$ the error case (2) gives an unsatisfiable constraint, so we only get two new equations:

$$
\left(\left\{ \begin{array}{ll}
(\mathrm{B}) & \mathsf{return}(0) \approx \mathsf{return}(n) \\
& [0 \leq n < \mathsf{size}(x) \land \forall i \in \{0, n-1\}[\mathsf{select}(x, i) \neq 0] \land \\
& \mathsf{select}(x, n) = 0 \land 0 \leq 0 < \mathsf{size}(x) \land \mathsf{select}(x, 0) = 0] \\
(\mathrm{C}) & \mathsf{u}(x, 0 + 1) \approx \mathsf{return}(n) \\
& [0 \leq n < \mathsf{size}(x) \land \forall i \in \{0, n-1\}[\mathsf{select}(x, i) \neq 0] \land \\
& \mathsf{select}(x, n) = 0 \land 0 \leq 0 < \mathsf{size}(x) \land \mathsf{select}(x, 0) \neq 0]
\end{array} \right\}, \{(\mathrm{A})\} \right)
$$

As the constraint of (B) implies that $n = 0$ (because of the quantification and $\mathsf{select}(x, 0) = 0$), we can remove (B) using EQ-DELETION and DELETION.

As for (C), we simplify with a calculation:

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

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

$$\left(\left\{ \begin{array}{ll} \text{(E)} & \mathsf{return}(1) \approx \mathsf{return}(n) \\ & [0 \leq n < \mathsf{size}(x) \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x,i) \neq 0] \wedge \\ & \mathsf{select}(x,n) = 0 \wedge 0 < \mathsf{size}(x) \wedge \mathsf{select}(x,0) \neq 0 \wedge \\ & 0 \leq 1 < \mathsf{size}(x) \wedge \mathsf{select}(x,1) = 0] \\ \text{(F)} & \mathsf{u}(x,1+1) \approx \mathsf{return}(n) \\ & [0 \leq n < \mathsf{size}(x) \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x,i) \neq 0] \wedge \\ & \mathsf{select}(x,n) = 0 \wedge 0 < \mathsf{size}(x) \wedge \mathsf{select}(x,0) \neq 0 \wedge \\ & 0 \leq 1 < \mathsf{size}(x) \wedge \mathsf{select}(x,1) \neq 0] \end{array} \right\}, \left\{ \begin{array}{l} \text{(A)} \\ \text{(D)} \end{array} \right\} \right)$$

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

$$\left(\left\{ \begin{array}{ll} \text{(G)} & \mathsf{return}(2) \approx \mathsf{return}(n) \\ & [0 \leq n < \mathsf{size}(x) \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x,i) \neq 0] \wedge \\ & \mathsf{select}(x,n) = 0 \wedge 0 < \mathsf{size}(x) \wedge \mathsf{select}(x,0) \neq 0 \wedge 1 < \\ & \mathsf{size}(x) \wedge \mathsf{select}(x,1) \neq 0 \ \wedge 2 < \mathsf{size}(x) \wedge \mathsf{select}(x,2) = 0] \\ \text{(H)} & \mathsf{u}(x,2+1) \approx \mathsf{return}(n) \\ & [0 \leq n < \mathsf{size}(x) \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x,i) \neq 0] \wedge \\ & \mathsf{select}(x,n) = 0 \wedge 0 < \mathsf{size}(x) \wedge \mathsf{select}(x,0) \neq 0 \wedge 1 < \\ & \mathsf{size}(x) \wedge \mathsf{select}(x,1) \neq 0 \wedge 2 < \mathsf{size}(x) \wedge \mathsf{select}(x,2) \neq 0] \end{array} \right\}, \left\{ \begin{array}{l} \text{(A)} \\ \text{(D)} \\ \text{(F}') \end{array} \right\} \right)$$

Again, we get rid of (G) easily. Now, simplifying (H) and reformulating its constraint, we obtain:

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

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's generalize! We obtain ({(J},{...}), where (J) is $\mathsf{u}(x,k) \approx \mathsf{return}(n)$ $[\varphi]$ with:

$$\begin{array}{l} \varphi : [k = m+1 \wedge 0 \leq n < \mathsf{size}(x) \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x,i) \neq 0] \wedge \\ \qquad \mathsf{select}(x,n) = 0 \wedge 0 \leq m < \mathsf{size}(x) \wedge \forall j \in \{0, m\}[\mathsf{select}(x,j) \neq 0] \end{array}$$

Obviously, (I) is an instance of (J); we use EXPANSION to obtain:

$$\left(\left\{ \begin{array}{ll} \text{(K)} & \mathsf{error} \approx \mathsf{return}(n) \ [\varphi \wedge (k < 0 \vee k \geq \mathsf{size}(x))] \\ \text{(L)} & \mathsf{return}(k) \approx \mathsf{return}(n) \\ & [\varphi \wedge 0 \leq k < \mathsf{size}(x) \wedge \mathsf{select}(x,k) = 0] \\ \text{(M)} & \mathsf{u}(x,k+1) \approx \mathsf{return}(n) \\ & [0 \leq k < \mathsf{size}(x) \wedge \mathsf{select}(x,k) \neq 0] \end{array} \right\}, \left\{ \begin{array}{l} \text{(A)} \\ \text{(D)} \\ \text{(F}') \\ \text{(J)} \end{array} \right\} \right)$$

Now, for all cases, note that the two $\forall$ statements, together with $\mathsf{select}(x,n)=\mathsf{0}$, imply that $m < n$, so that $k \leq n$. Consequently, the constraint of (K) is unsatisfiable: $k = m+1$ and $\mathsf{0} \leq m$ together imply that $k < \mathsf{0}$ cannot hold, and since $k \leq n$, we see that $k \geq \mathsf{size}(x)$ cannot hold in combination with $n < \mathsf{size}(x)$. By DELETION, we may remove (K).

For (L), we use EQ-DELETION. Note that the two $\forall$ statements, together with $\mathsf{select}(x,k)=\mathsf{0}$, imply that $n-\mathsf{1} < k$, so $n \leq k$. Since we also have that $k \leq n$, the resulting constraint is unsatisfiable; we use DELETION again.

Finally, consider (M). Simplifying this equation with a calculation, and reformulating the constraint, we have:

$$\begin{aligned}(\mathrm{M}')\ \mathsf{u}(x,p) \approx \mathsf{return}(n)\ [&p = k+\mathsf{1} \wedge \mathsf{select}(x,n)=\mathsf{0} \wedge \mathsf{0} \leq n < \mathsf{size}(x) \wedge\\ &\forall i \in \{\mathsf{0}, n-\mathsf{1}\}[\mathsf{select}(x,i) \neq \mathsf{0}] \wedge \mathsf{0} \leq k < \mathsf{size}(x) \wedge\\ &\forall j \in \{\mathsf{0}, k\}[\mathsf{select}(x,j) \neq \mathsf{0}] \wedge \text{some constraints on } m]\end{aligned}$$

This equation is simplified to an equation of the form $\mathsf{return}(n) \approx \mathsf{return}(n)$ $[\ldots]$ using the induction rule (J); we are done with DELETION.

Now recall the strcpy function from Example 15.

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

$$\begin{aligned}
&(5)\ \ \mathsf{strcpy}(x,y) \to \mathsf{v}(x,y,\mathsf{0})\\
&(6)\ \ \mathsf{v}(x,y,i) \to \mathsf{error}_{\mathsf{strcpy}}\ [i < \mathsf{0} \vee i \geq \mathsf{size}(y)]\\
&(7)\ \ \mathsf{v}(x,y,i) \to \mathsf{w}(x,y,i)\ [\mathsf{0} \leq i < \mathsf{size}(y) \wedge \mathsf{select}(y,i) = \mathsf{0}]\\
&(8)\ \ \mathsf{v}(x,y,i) \to \mathsf{error}_{\mathsf{strcpy}}\ [\mathsf{0} \leq i < \mathsf{size}(y) \wedge \mathsf{select}(y,i) \neq \mathsf{0} \wedge i \geq \mathsf{size}(x)]\\
&(9)\ \ \mathsf{v}(x,y,i) \to \mathsf{v}(\mathsf{store}(x,i,\mathsf{select}(y,i)),y,i+\mathsf{1})\\
&\qquad\qquad [\mathsf{0} \leq i < \mathsf{size}(x) \wedge i < \mathsf{size}(y) \wedge \mathsf{select}(y,i) \neq \mathsf{0}]\\
&(10)\ \ \mathsf{w}(x,y,i) \to \mathsf{error}_{\mathsf{strcpy}}\ [i < \mathsf{0} \vee i \geq \mathsf{size}(x)]\\
&(11)\ \ \mathsf{w}(x,y,i) \to \mathsf{return}_{\mathsf{strcpy}}(\mathsf{store}(x,i,\mathsf{0}))\ [\mathsf{0} \leq i < \mathsf{size}(x)]
\end{aligned}$$

To see that this implementation is correct, there are two things we can do. The obvious thing is to compare with an existing, trusted implementation of strcpy. Alternatively, like with strlen, we could formally prove correctness. This means that we must show that if a is a valid C-string, and b an array with size greater than the length of a, then $\mathsf{strcpy}(a,b)$ updates a to correspond, as a C-string, to b. To this end, let us introduce a new symbol test, and two rules:

$$\begin{aligned}
&(12)\ \ \mathsf{test}(x,n,\mathsf{error}) \to \mathsf{false}\\
&(13)\ \mathsf{test}(x,n,\mathsf{return}(y)) \to \forall i \in \{\mathsf{0},\ldots,n\}[\mathsf{select}(x,i) = \mathsf{select}(y,i)]
\end{aligned}$$

Correctness of strcpy follows if the following equation is an inductive theorem:

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

The proof of this follows roughly the same lines as the recursive proof of strlen, and is found automatically by our tool (see Section 6); let us present here the main steps (but without keeping track of induction rules which aren't used).

The first step is SIMPLIFICATION at position 3.

$$\text{(O)} \quad \mathsf{test}(x, n, \mathsf{v}(y, x, 0)) \approx \mathsf{true} \;[0 \leq n < \mathsf{size}(x) \wedge n < \mathsf{size}(y) \wedge \mathsf{select}(x, n) = 0 \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x, i) = 0]]$$

We apply EXPANSION at the same position; the error rules both give an unsatisfiable constraint (because of $0 \leq n < \mathsf{size}(x)$ and $n < \mathsf{size}(y)$), so can be omitted. We obtain two new equations:

$$\text{(P)} \quad \mathsf{test}(x, n, \mathsf{w}(y, x, 0)) \approx \mathsf{true} \;[0 \leq n < \mathsf{size}(x) \wedge n < \mathsf{size}(y) \wedge \mathsf{select}(x, n) = 0 \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x, i) = 0] \wedge 0 < \mathsf{size}(x) \wedge \mathsf{select}(x, 0) = 0]$$

$$\text{(Q)} \quad \mathsf{test}(x, n, \mathsf{v}(\mathsf{store}(y, 0, \mathsf{select}(x, 0)), x, 0+1)) \approx \mathsf{true} \;[0 \leq n < \mathsf{size}(x) \wedge n < \mathsf{size}(y) \wedge \mathsf{select}(x, n) = 0 \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x, i) = 0] \wedge 0 < \mathsf{size}(y) \wedge 0 < \mathsf{size}(x) \wedge \mathsf{select}(x, 0) \neq 0]$$

Using SIMPLIFICATION with rules (11) and (13) on (P), we obtain an equation that can easily be eq-deleted / deleted away, as the constraint implies that $n = 0$.

As for (Q), note that the $0 < \mathsf{size}(x)$ and $0 < \mathsf{size}(y)$ clauses are redundant as $0 \leq n$. We simplify the equation with calculations, obtaining:

$$\text{(R)} \quad \mathsf{test}(x, n, \mathsf{v}(z, x, 1)) \approx \mathsf{true} \;[0 \leq n < \mathsf{size}(x) \wedge n < \mathsf{size}(y) \wedge \mathsf{select}(x, n) = 0 \wedge \mathsf{select}(x, 0) \neq 0 \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x, i) = 0] \wedge k = \mathsf{select}(x, 0) \wedge z = \mathsf{store}(y, 0, k)]$$

Now, this constraint is somewhat awkward to reason with; in particular the store relation between y and z. So, noting that y only occurs in the constraint now, and not in the main part of the equation, we can reformulate the constraint:

$$\text{(R}') \quad \mathsf{test}(x, n, \mathsf{v}(z, x, 1)) \approx \mathsf{true} \;[0 \leq n < \mathsf{size}(x) \wedge n < \mathsf{size}(z) \wedge \mathsf{select}(x, n) = 0 \wedge \mathsf{select}(x, 0) \neq 0 \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x, i) = 0] \wedge k = \mathsf{select}(x, 0) \wedge \mathsf{select}(z, 0) = k]$$

Or, even simpler, since k also plays only a limited role:

$$\text{(R}'') \quad \mathsf{test}(x, n, \mathsf{v}(z, x, 1)) \approx \mathsf{true} \;[0 \leq n < \mathsf{size}(x) \wedge n < \mathsf{size}(z) \wedge \mathsf{select}(x, n) = 0 \wedge \mathsf{select}(x, 0) \neq 0 \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x, i) = 0] \wedge \mathsf{select}(x, 0) = \mathsf{select}(z, 0)]$$

We use EXPANSION again. Noting that the $\mathsf{select}(x, 0) \neq 0$ clause together with the quantification implies that $n \geq 1$, neither of the error-rules gives a satisfiable constraint, and the equation resulting from rule (7) is quickly simplified, eq-deleted and deleted away. This leaves the equation resulting from rule (9):

$$\text{(S)} \quad \mathsf{test}(x, n, \mathsf{v}(\mathsf{store}(z, 1, \mathsf{select}(x, 1)), x, 1+1)) \approx \mathsf{true} \;[0 \leq n < \mathsf{size}(x) \wedge n < \mathsf{size}(z) \wedge \mathsf{select}(x, n) = 0 \wedge \mathsf{select}(x, 0) \neq 0 \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x, i) = 0] \wedge \mathsf{select}(x, 0) = \mathsf{select}(z, 0) \wedge 0 \leq 1 < \mathsf{size}(z) \wedge 1 < \mathsf{size}(x) \wedge \mathsf{select}(x, 1) \neq 0]$$

Simplifying with calculations, and then reformulating the constraint as before:

$$\begin{aligned}(\mathrm{S}')\qquad & \mathsf{test}(x,n,\mathsf{v}(a,x,2)) \approx \mathsf{true}\\ & [0 \le n < \mathsf{size}(x) \wedge n < \mathsf{size}(a) \wedge \mathsf{select}(x,n) = 0 \wedge \mathsf{select}(x,0) \neq 0 \wedge\\ & \quad \forall i \in \{0, n-1\}[\mathsf{select}(x,i) = 0] \wedge \mathsf{select}(x,0) = \mathsf{select}(a,0) \wedge\\ & \quad \mathsf{select}(x,1) \neq 0 \wedge \mathsf{select}(a,1) = \mathsf{select}(x,1)]\end{aligned}$$

Note that here $a = \mathsf{store}(z, 1, \mathsf{select}(x, 1))$; the clause $\mathsf{select}(x, 0) = \mathsf{select}(z, 0)$ is replaced by $\mathsf{select}(x, 0) = \mathsf{select}(a, 0)$ because z and a correspond at position 0. Hence, the updated variable z plays no role, and was removed from the constraint.

But now we see a divergence arising. Using Expansion again, followed by Simplification as much as possible and reformulating the constraint, we obtain:

$$\begin{aligned}(\mathrm{T})\qquad & \mathsf{test}(x,n,\mathsf{v}(b,x,3)) \approx \mathsf{true}\\ & [0 \le n < \mathsf{size}(x) \wedge n < \mathsf{size}(b) \wedge \mathsf{select}(x,n) = 0 \wedge \mathsf{select}(x,0) \neq 0 \wedge\\ & \quad \forall i \in \{0, n-1\}[\mathsf{select}(x,i) = 0] \wedge \mathsf{select}(x,0) = \mathsf{select}(b,0) \wedge\\ & \quad \mathsf{select}(x,1) \neq 0 \wedge \mathsf{select}(b,1) = \mathsf{select}(x,1) \wedge\\ & \quad \mathsf{select}(x,2) \neq 0 \wedge \mathsf{select}(b,2) = \mathsf{select}(x,2)]\end{aligned}$$

Let us reformulate this, by taking the similar statements into quantifications!

$$\begin{aligned}(\mathrm{T}')\qquad & \mathsf{test}(x,n,\mathsf{v}(b,x,3)) \approx \mathsf{true}\\ & [0 \le n < \mathsf{size}(x) \wedge n < \mathsf{size}(b) \wedge \mathsf{select}(x,n) = 0 \wedge\\ & \quad \forall i \in \{0, n-1\}[\mathsf{select}(x,i) = 0] \wedge\\ & \quad \forall i \in \{0,2\}[\mathsf{select}(x,i) \neq 0] \wedge \forall i \in \{0,2\}[\mathsf{select}(b,i) = \mathsf{select}(x,i)]]\end{aligned}$$

We use Generalization, and create:

$$(\mathrm{U})\ \mathsf{test}(x,n,\mathsf{v}(b,x,k)) \approx \mathsf{true}\ [\varphi]$$

Where φ is the constraint:

$$\begin{aligned}& 0 \le n < \mathsf{size}(x) \wedge n < \mathsf{size}(b) \wedge \mathsf{select}(x,n) = 0 \wedge 0 \le k \wedge\\ & \quad \forall i \in \{0, n-1\}[\mathsf{select}(x,i) = 0] \wedge\\ & \forall i \in \{0, k-1\}[\mathsf{select}(x,i) \neq 0] \wedge \forall i \in \{0, k-1\}[\mathsf{select}(b,i) = \mathsf{select}(x,i)]\end{aligned}$$

Now, aside from the induction rule (U), Expansion gives:

$$\begin{aligned}&(\mathrm{V})\ \mathsf{test}(x,n,\mathsf{error}_{\mathsf{strcpy}}) \approx \mathsf{true}\ [\varphi \wedge (k < 0 \vee k \ge \mathsf{size}(x))]\\ &(\mathrm{W})\ \mathsf{test}(x,n,\mathsf{w}(b,x,k)) \approx \mathsf{true}\ [\varphi \wedge 0 \le k < \mathsf{size}(x) \wedge \mathsf{select}(x,k) = 0]\\ &(\mathrm{X})\ \mathsf{test}(x,n,\mathsf{error}_{\mathsf{strcpy}}) \approx \mathsf{true}\ [\varphi \wedge \ldots \wedge k \ge \mathsf{size}(b))]\\ &(\mathrm{Y})\ \mathsf{test}(x,n,\mathsf{v}(\mathsf{store}(b,k,\mathsf{select}(x,k)),x,k+1)) \approx \mathsf{true}\\ &\qquad\qquad [\varphi \wedge 0 \le k < \mathsf{size}(x) \wedge \mathsf{select}(x,k) \neq 0]\end{aligned}$$

As in the strlen case, we note that φ implies that $0 \le k \le n$. As φ additionally implies that $n < \mathsf{size}(x), \mathsf{size}(b)$, equations (V) and (X) both have unsatisfiable constraints. Thus, we remove them using Deletion.

For (W), note that the added clause $\mathsf{select}(x,k) = \mathsf{0}$ together with the $\forall$ clauses implies $n \leq k$; thus we have $n = k$. This allows us to simplify with a calculation, and follow up with SIMPLIFICATIONs with rule (11) and (13), giving:

$$\forall i \in \{0,\ldots,k\}[\mathsf{select}(x,i) = \mathsf{select}(y,i)] \approx \mathsf{true}$$
$$[\varphi \wedge \mathsf{0} \leq k < \mathsf{size}(x) \wedge \mathsf{select}(x,k) = \mathsf{0} \wedge y = \mathsf{store}(b,k,\mathsf{0})]$$

We can reduce this with a calculation to $\mathsf{true} \approx \mathsf{true}\ [\ldots]$, as the constraint implies that the quantification evaluates to true (the case for $i < k$ is given by the last quantification of φ, and the case for $i = k$ because $\mathsf{select}(x,k) = \mathsf{select}(y,k) = \mathsf{0}$).

As for (Y), let us simplify and reformulate the equation:

(Z)
$$\mathsf{test}(x,n,\mathsf{v}(y,x,k')) \approx \mathsf{true}$$
$$[\mathsf{0} \leq n < \mathsf{size}(x) \wedge n < \mathsf{size}(y) \wedge \mathsf{select}(x,n) = \mathsf{0} \wedge$$
$$\forall i \in \{\mathsf{0}, n-\mathsf{1}\}[\mathsf{select}(x,i) = \mathsf{0}] \wedge$$
$$\forall i \in \{\mathsf{0}, k-\mathsf{1}\}[\mathsf{select}(x,i) \neq \mathsf{0}] \wedge \forall i \in \{\mathsf{0}, k-\mathsf{1}\}[\mathsf{select}(y,i) = \mathsf{select}(x,i)] \wedge$$
$$\mathsf{0} \leq k < \mathsf{size}(x) \wedge \mathsf{select}(x,k) \neq \mathsf{0} \wedge k' = k+\mathsf{1} \wedge \mathsf{select}(y,k) = \mathsf{select}(x,k)]$$

Or, taking clauses into the quantifications where possible:

(Z′)
$$\mathsf{test}(x,n,\mathsf{v}(y,x,k')) \approx \mathsf{true}$$
$$[\mathsf{0} \leq n < \mathsf{size}(x) \wedge n < \mathsf{size}(y) \wedge \mathsf{select}(x,n) = \mathsf{0} \wedge$$
$$\forall i \in \{\mathsf{0}, n-\mathsf{1}\}[\mathsf{select}(x,i) = \mathsf{0}] \wedge \forall i \in \{\mathsf{0}, k'-\mathsf{1}\}[\mathsf{select}(x,i) \neq \mathsf{0}] \wedge$$
$$\forall i \in \{\mathsf{0}, k'-\mathsf{1}\}[\mathsf{select}(y,i) = \mathsf{select}(x,i)] \wedge \mathsf{0} \leq k' \leq \mathsf{size}(x)]$$

Simplifying with induction rule (U), we obtain a trivial equation $\mathsf{true} \approx \mathsf{true}\ [\ldots]$.

5 Lemma Generalization by Dropping Initializations

Divergence, like we encountered in all three examples in Section 4, is very common in inductive theorem proving. As explained before, this is only natural: in mathematical proofs, when basic induction is insufficient to prove a theorem, 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 [3,10,11,17,22,23]. Lemma equations are typically derived during solving: we generate candidates when the proof state is in divergence.

Here, we propose a new method, specialized for systems with constraints. The generalizations from Sections 4.4 and 4.5 were found using this technique. Although the method is very simple (at its core, we just drop a part of the constraint), it is particularly effective for LCTRSs obtained from procedural programs.

5.1 Generalizing Initializations

Towards a new lemma generalization technique, let us state the rules of $\mathcal{R}_{\mathsf{sum}}$ in an alternative way. Whenever the right-hand side of a rule has a subterm $f(\ldots,n,\ldots)$ with f a defined symbol and n a value, then we replace n by a fresh variable v_i, and add $v_i = n$ to the constraint. We call this an *initialization-free* representation of the rules. In $\mathcal{R}_{sum}$, rules (2)–(9) are left unmodified, but (1) is replaced by:

$$(1')\ \mathsf{sum1}(arr, n) \to \mathsf{u}(arr, n, v_1, v_2)\ \ [v_1 = 0 \wedge v_2 = 0]$$

Evidently, these rules generate the same rewrite relation as the original rules.

Let us consider what happens now if we use the same steps as in Section 4.4. We make one change to the definitions from Section 4.2: since the variables v_i are associated to fixed values, they are not renamed in Expansion. Also, when simplifying the presentation of a constrained term, we ignore $v_i = n$ clauses, since for instance translating $f(v_i)$ $[v_i = 0]$ back to $f(0)$ [true] would defeat the point. The resulting induction has the same shape, but with more complex equations. Some instances:

$$\begin{aligned}
(\mathrm{B}')&: \quad \mathsf{u}(a,k,v_1,v_2) \approx \mathsf{sum4}(a,k) \\
&\quad [0 \le k \le \mathsf{size}(a) \wedge v_1 = 0 \wedge v_2 = 0] \\
(\mathrm{E}')&: \mathsf{w}(\mathsf{select}(a,k-1), \mathsf{sum4}(a,k-1)) \approx \mathsf{u}(a,k,v_1 + \mathsf{select}(a,v_2), v_2 + 1) \\
&\quad [0 \le k \le \mathsf{size}(a) \wedge v_1 = 0 \wedge v_2 = 0 \wedge 0 \le k-1 < \mathsf{size}(a)] \\
(\mathrm{F}')&: \quad \mathsf{w}(n, \mathsf{sum4}(a,k')) \approx \mathsf{u}(a,k,r_0,i_0) \\
&\quad [k' = k-1 \wedge 0 \le k' < \mathsf{size}(a) \wedge v_1 = 0 \wedge v_2 = 0 \wedge n = \mathsf{select}(a,k') \wedge \\
&\qquad r_0 = v_1 + \mathsf{select}(a,v_2) \wedge i_0 = v_2 + 1] \\
(\mathrm{H}')&: \quad \mathsf{u}(a,k,r_0 + \mathsf{select}(a,i_0), i_0 + 1) \approx \mathsf{w}(n, \mathsf{u}(a,k',v_1,v_2)) \\
&\quad [k' = k-1 \wedge 0 \le k' < \mathsf{size}(a) \wedge v_1 = 0 \wedge v_2 = 0 \wedge n = \mathsf{select}(a,k') \wedge \\
&\qquad r_0 = v_1 + \mathsf{select}(a,v_2) \wedge i_0 = v_2 + 1 \wedge i_0 < k \wedge 0 \le i_0 < \mathsf{size}(a)] \\
(\mathrm{J}')&: \quad \mathsf{u}(a,k,r_1,i_1) \approx \mathsf{w}(n, \mathsf{u}(a,k',r_0,i_0)) \\
&\quad [k' = k-1 \wedge 0 \le k' < \mathsf{size}(a) \wedge v_1 = 0 \wedge v_2 = 0 \wedge n = \mathsf{select}(a,k') \wedge \\
&\qquad r_0 = v_1 + \mathsf{select}(a,v_2) \wedge i_0 = v_2 + 1 \wedge i_0 < k \wedge 0 \le i_0 < \mathsf{size}(a) \wedge \\
&\qquad i_1 = i_0 + 1 \wedge r_1 = r_0 + \mathsf{select}(a,i_0)] \\
(\mathrm{K}')&: \quad \mathsf{u}(a,k,r_2,i_2) \approx \mathsf{w}(n, \mathsf{u}(a,k',r_1,i_1)) \\
&\quad [k' = k-1 \wedge 0 \le k' < \mathsf{size}(a) \wedge v_1 = 0 \wedge v_2 = 0 \wedge n = \mathsf{select}(a,k') \wedge \\
&\qquad r_0 = v_1 + \mathsf{select}(a,v_2) \wedge i_0 = v_2 + 1 \wedge 0 \le i_0 \wedge i_1 < k \wedge \\
&\qquad i_1 = i_0 + 1 \wedge r_1 = r_0 + \mathsf{select}(a,i_0) \wedge i_2 = i_1 + 1 \wedge r_2 = r_1 + \mathsf{select}(a,i_1)] \\
(\mathrm{L}')&: \quad \mathsf{u}(a,k,r_3,i_3) \approx \mathsf{w}(n, \mathsf{u}(a,k',r_2,i_2)) \\
&\quad [k' = k-1 \wedge 0 \le k' < \mathsf{size}(a) \wedge v_1 = 0 \wedge v_2 = 0 \wedge n = \mathsf{select}(a,k') \wedge \\
&\qquad r_0 = v_1 + \mathsf{select}(a,v_2) \wedge i_0 = v_2 + 1 \wedge 0 \le i_0 \wedge i_2 < k \wedge \\
&\qquad i_1 = i_0 + 1 \wedge r_1 = r_0 + \mathsf{select}(a,i_0) \wedge i_2 = i_1 + 1 \wedge r_2 = r_1 + \mathsf{select}(a,i_1) \wedge \\
&\qquad i_3 = i_2 + 1 \wedge r_3 = r_2 + \mathsf{select}(a,i_2)]
\end{aligned}$$

Notice that, if we ignore the constraint, each of the divergent equations (J′), (K′), (L′) is the same modulo renaming of variables, while the constraint grows. Essentially, we keep track of parts of the history of an equation in its constraint.

Backtracking to the proof state $(\{(\mathrm{J}')\}, \{(\mathrm{B}'^{-1}),(\mathrm{G}'^{-1})\})$, we generalize the equation by dropping all clauses $v_i = q_i$, where v_i is an initialization variable. To make it explicit that these variables no longer play a special role, we rename them. Thus, we generate the following generalized equation:

$$\begin{aligned}
&\mathsf{u}(a,k,r_1,i_1) \approx \mathsf{w}(n, \mathsf{u}(a,k',r_0,i_0))\ [k' = k-1 \wedge 0 \le k' < \mathsf{size}(a) \wedge n = \\
&\quad \mathsf{select}(a,k') \wedge r_0 = x_1 + \mathsf{select}(a,x_2) \wedge i_0 = x_2 + 1 \wedge i_0 < k \wedge \\
&\quad 0 \le i_0 < \mathsf{size}(a) \wedge i_1 = i_0 + 1 \wedge r_1 = r_0 + \mathsf{select}(a,i_0)]
\end{aligned}$$

Reformulating this, to remove those variables which do not contribute anything to the equation (e.g. suitable x_1, x_2 always exist, taking $x_2 = i_0 - 1$ and $x_1 = \mathsf{select}(a, x_2) - r_0$), we obtain:

$$(\mathrm{N}') \qquad \begin{array}{c} \mathsf{u}(a, k, r_1, i_1) \approx \mathsf{w}(n, \mathsf{u}(a, k', r_0, i_0)) \\ [k' = k - 1 \wedge 0 \leq k' < \mathsf{size}(a) \wedge n = \mathsf{select}(a, k') \wedge 0 \leq i_0 < k \wedge \\ i_1 = i_0 + 1 \wedge r_1 = r_0 + \mathsf{select}(a, i_0)] \end{array}$$

Note that $(\mathrm{N}') \sim (\mathrm{N})$. We can now continue the proof process with $(\{(\mathrm{N}')\}, \{(\mathrm{B}'^{-1}), (\mathrm{G}'^{-1})\})$, and complete much like we did in Section 4.4.

5.2 Discussion

Thus, the generalization technique in this paper is straightforward to use:

- replace initializations by variables in the original rules;
- remove the definitions of those initializations when a divergence is detected.

The only downside is that, in order to use this technique, we have to use the altered rules from the beginning, so we keep track of the v_i variables throughout the recursive procedure. For an automatic analysis this is no problem, however.

Note that we can only use this method if the equation part of the divergence has the same shape every time. This is the case for sum, because the rule that causes the divergence has the form $\mathsf{u}(x_1, \ldots, x_n) \to \mathsf{u}(r_1, \ldots, r_n)\ [\varphi]$, preserving its outer shape. In general, the generalization method is most likely to be successful when analyzing tail-recursive functions (with accumulators), such as those obtained from procedural programs. We should also be able to handle mutually recursive functions, like $\mathsf{u}(x_1, \ldots, x_n) \to \mathsf{w}(r_1, \ldots, r_m)\ [\varphi]$ and $\mathsf{w}(y_1, \ldots, y_m) \to \mathsf{u}(q_1, \ldots, q_n)\ [\psi]$. To analyze systems with general recursion, however, we will need different generalization or lemma generation techniques.

The given generalization method also works for strlen and strcpy from Section 4.5. In these cases, we additionally have to collect multiple clauses into a quantified clause before generalizing, as was done for equation (I) in Section 4.5.

6 Implementation

The method for program verification, as described in this paper, can be broken down into two parts:

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

An initial implementation of part 1, limited to integer functions without function calls or error checking, is available at:

`http://www.trs.cm.is.nagoya-u.ac.jp/c2rs/`

We are still working on an implementation of the full transformation.

Part 2, the core method on LCTRSs, has been implemented in Ctrl, our tool for analyzing constrained term rewriting. As prerequisites, we have also implemented basic termination proving techniques and automatic detection of quasi-reductivity and confluence. To deal with constraints, the tool is coupled both with a small internal reasoner and the external SMT-solver Z3 [16]. Here, Z3 has the advantage of being quite strong at *unsatisfiability* as well as satisfiability, which is essential to automatically test *validity* of constraints.

The internal reasoner has two functions:

- automatically detecting satisfiability or validity of simple statements quickly, without a call to an SMT-solver; this both gives a speedup, and allows us to optimize for often recurring questions (e.g. "find $n_1, \ldots, n_k \in \{-2, \ldots, 2\}$ such that φ is valid", a satisfiability question that we need for *polynomial interpretations*, to prove termination);
- simplifying the constraints of equations, by for instance combining statements into quantifications; as this affects the shape of generalized equations, this significantly increases the power of the tool for functions like strlen and strcpy.

In addition, our notion of arrays is not supported by mainstream SMT-solvers, so we translate our array formulas into the SMT-LIB array-format; an array is encoded as a function from $\mathbb{Z}$ to $\mathbb{Z}$, with an additional variable encoding its size.

At the moment, Postulate is not implemented (but Generalization is), and Disprove is present only in a limited form. The non-rule adding case of Expansion is instead called `case`. A complete implementation of the method described in this technical report is still work in progress.

To obtain Ctrl, see: `http://cl-informatik.uibk.ac.at/software/ctrl/`.

6.1 Strategy

Even assuming an oracle for determining satisfiability and validity of the constraints, it is not immediately clear how to best implement rewriting induction. There are many choices to be made during a derivation: what inference rule to apply, what position and rule to use in Simplification, what substitution to use (as we saw when simplifying equation (Q) in Section 4.4, this is not always obvious), how to simplify the constraint in between rewriting steps...

For the first choices, Ctrl uses a simple strategy: we try, in the following order: Disprove, EQ-deletion and Deletion together, Constructor, Simplification, Expansion, and Generalization (simply omitting all v_i definitions). Whenever an inference rule applies, we continue from the start of the list. If $b =$ incomplete but Disprove would otherwise succeed, or if we have gone too deep without removing any of the main equations, we backtrack and try something else. At the moment, divergence is not automatically detected, although this is an obvious extension.

To rewrite an equation, in Simplification and Expansion, we try to instantiate as many variables in the rule by variables in the equation as possible. That is, to rewrite an equation $s \approx t\ [\varphi_1 \wedge \cdots \wedge \varphi_n]$ at the root with a rule

$l \to r\ [\psi_1 \wedge \cdots \wedge \psi_m]$, we first determine some γ such that $s = \ell\gamma$, and $\gamma(v_i) = v_i$ for all the special variables. Additionally, if any ψ_i has the form $C[x, y_1, \ldots, y_k]$ with $x \in Dom(\gamma)$ and all $y_i \notin Dom(\gamma)$, and there is some $\psi_j = C\gamma[\gamma(x), s_1, \ldots, s_k]$, then let $\gamma(y_i) := s_i$ for all i. This is exactly what we did for equation (Q) in Section 4.4 and (M'), (Z') in Section 4.5. Other variables can be chosen fresh.

When simplifying constraints, the $v_i = n$ definitions are ignored. Otherwise, when a clause is clearly implied by other clauses, it may be removed; we also remove clauses for variables which do not play a role anymore (such as m in equation (M) in Section 4.5). Most importantly, Ctrl tries to introduce *ranged quantifications* $\forall x \in \{k_1, \ldots, k_n\}[\varphi(x)]$ whenever possible (as we also saw in Section 4.5), provided $n \geq$. If one of the boundaries of the range is a special variable v_i, we replace it by the value it is defined as, since it is typically better not to generalize the starting point of a quantification, only the ending point.

6.2 Experiments

To test performance of Ctrl, we used assignments from a group of students in the first-year programming course in Nagoya. Unfortunately, although we know how to translate C-programs to LCTRSs, we do not yet have a complete implementation. Therefore, we translated five groups by hand: `sum` (given n, implement $\sum_{i=1}^{n} i$), `fib` (calculate the first n Fibonacci numbers), `sumfrom` (given n and m, implement $\sum_{i=n}^{m} i$), `strlen` and `strcpy`. Due to the large effort of manually translating, we only use this small sample space. We considered two further assignments, with our own implementations: `arrsum` (the array summation from Section 4.4), and `fact` (the factorial function from examples 1 and 9).

We quickly found that many implementations were incorrect: the students had often forgotten to account for, e.g., negative input or empty strings. Correcting for this (by altering the constraint, or excluding the benchmark), Ctrl automatically verified most of the queries, as summarized to the right. Here, for instance "3 / 5" means that 3 out of the 5 different correct functions could automatically be verified. The runtime is in seconds, and includes only benchmarks where Ctrl succeeded.

function	verified	time
`sum`	9 / 13	4.8
`fib`	10 / 12	11.4
`strlen`	3 / 5	16.2
`strcpy`	3 / 6	30.0
`sumfrom`	2 / 5	5.6
`arrsum`	1 / 1	14.2
`fact`	1 / 1	4.3

Investigating the failures, we found that the main weakness of the method was not the simple generalization method, but the termination requirement! As Ctrl's termination module is not very strong yet, several times the initial LCTRS could not be handled; also, sometimes a natural induction rule was not introduced because it would cause non-termination (although in most of these cases, expanding at a different position still led to a proof). Another weakness is that sometimes, generalizing removes the relation between two variables (e.g. both x and y are initialized to 0, and are both increased by 1 in every loop iteration). This suggests a natural direction for improvements to the technique.

An evaluation page, including exact problem statements, is given in:

`http://cl-informatik.uibk.ac.at/software/ctrl/aplas14/`

7 Related Work

The related work can be split into two categories. First, the literature on rewriting induction; and second, the work on program verification.

7.1 Rewriting Induction

The inductive theorem proving method in this paper is not new; building on a long literature about rewriting induction (see e.g. [1,5,19,21]), the method should mostly be seen as an adaptation of existing techniques to the LCTRS formalism introduced in [14], which therefore generalizes the possibilities of earlier work.

The most relevant related works are [5,21], where rewriting induction is defined for different styles of constrained rewriting. In both cases, the formalisms used are restricted to *integer* functions and predicates; it is not clear how they can be generalized to handle more advanced theories. LCTRSs offer a more general setting, which allows us to use rewriting induction also for systems with for instance arrays, bitvectors or real numbers. Additionally, by not restricting the predicates in Σ_{theory}, we can handle (a limited form of) quantifiers in constraints.

To enable these advantages, we had to make subtle changes to the inference rules, in particular Simplification and Expansion. Our changes make it possible to modify constraints of an equation, and to handle *irregular* rules, where the constraint introduces fresh variables. This has the additional advantage that it enables Expansion steps when this would create otherwise infeasible rules.

Furthermore, the method requires a very different implementation from previous definitions: we need separate strategies to simplify constraints (e.g. deriving quantified statements), and, in order to permit the desired generality, must rely primarily on external solvers to manipulate constraints.

In addition to the adaptation of rewriting induction, we have introduced a completely new lemma generalization technique, which offers a powerful tool for analyzing loops in particular. A similar idea (abstracting the initialization values) is used in [17], but the execution is very different. In [17], an equation $s \approx t\ [\varphi]$ is generalized by first adapting $s \approx t$ using templates obtained from the rules, then by generalizing φ using a set of relations between positions, which the proof process keeps track of. In our method, the constraint carries all the information. Our method succeeds on all examples in [17], and on some where [17] fails (cf. Appendix B),

For *unconstrained* systems, there are several generalization methods in the literature, e.g., [10,11,22]. Mostly, these approaches are very different from our method. Most similar, perhaps, is [10], which also proposes a method to generalize initial values. As observed in [17], this method is not sufficient to handle 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 far as we are aware, there is no other work for lemma generation of rewrite systems (or functional programs) obtained from procedural programs.

7.2 Automatic Program Verification

Although the current paper is a primarily theoretical contribution to the field of constraint rewriting induction, our intended goal is to (automatically) verify correctness properties of procedural programs.

As mentioned in the introduction, however, most existing verifiers require human interaction. Exceptions are the fully automated tools in the *Competition on Software Verification* (SV-COMP, `http://sv-comp.sosy-lab.org/`), which verify program properties like reachability, termination and memory-safety.

However, comparing our approach to these tools does not seem useful. While we can, to some extent, tackle termination and memory-safety, the main topic of this paper is *equivalence*, which is not studied in SV-COMP. And while technically equivalence problems can be formulated as reachability queries (e.g., $f(x) \approx g(x)$ [c] is handled by the `main` function to the right), neither of the top two tools in the "recursive" category of SV-COMP halts succesfully (in two hours) for our simplest (integer) example sum.

```
int main() {
  int x =
    __VERIFIER_nondet_int();
  if (c && f(x) != g(x)) {
    ERROR: goto ERROR;
  }
  return 0;
}
```

8 Conclusions

In this paper, 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 in-built error checking for all mistakes we are interested in.

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 a new technique to generalize equations. The idea of this 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.

In addition to the theory of these techniques, we provide an implementation that automatically verifies inductive theorems. Initial results on a small database of student programs are very promising. In future work, we will aim to increase the strength of this implementation and couple it with an automatic transformation tool which converts procedural programs into LCTRSs.

Acknowledgements: we are grateful to Stephan Falke, who contributed to an older version of this work, and for the helpful remarks of the reviewers for [15].

References

1. Bouhoula, A.: Automated theorem proving by test set induction. Journal of Symbolic Computation, vol. 23(1), pp. 47–77. Elsevier (1997)
2. Bundy, A.: The automation of proof by mathematical induction. In: Voronkov, A., Robinson, A. (eds.) Handbook of Automated Reasoning, pp. 845–911. Elsevier (2001)
3. Bundy, A., Basin, D., Hutter, D., Ireland, A.: Rippling: Meta-Level Guidance for Mathematical Reasoning. Cambridge University Press (2005)
4. Falke, S., Kapur, D.: A term rewriting approach to the automated termination analysis of imperative programs. In: Schmidt. R.A. (ed.) Automated Deduction (CADE). LNCS, vol. 5663, pp. 277–293. Springer, Heidelberg (2009)
5. Falke, S., Kapur, D.: Rewriting induction + linear arithmetic = decision procedure. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning (IJCAR). LNAI, vol. 7364, pp. 241–255. Springer, Heidelberg (2012)
6. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: Schmidt-Schauß, M. (ed.) 22nd International Conference on Rewriting Techniques and Applications (RTA). LIPIcs, vol. 10, pp. 41–50. Dagstuhl, Leibniz (2011)
7. Falke, S.: Term Rewriting with Built-In Numbers and Collection Data Structures. Ph.D. thesis, University of New Mexico, Albuquerque, NM, USA (2009)
8. Furuichi, Y., Nishida, N., Sakai, M., Kusakari, K., Sakabe, T.: Approach to procedural-program verification based on implicit induction of constrained term rewriting systems. In: IPSJ Transactions on Programming, vol. 1(2), pp. 100–121 (2008), in Japanese. (**)
9. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press (2000)
10. Kapur, D., Sakhanenko, N.A.: Automatic generation of generalization lemmas for proving properties of tail-recursive definitions. In: Basin, D., Wolff, B. (eds.) Theorem Proving in Higher Order Logics (TPHOLs). LNCS, vol. 2758, pp. 136–154. Springer, Heidelberg (2003)
11. Kapur, D., Subramaniam, M.: Lemma discovery in automated induction. In: McRobbie, M.A., Slaney, J.K. (eds.) Automated Deduction (CADE). LNCS, vol. 1104, pp. 538–552. Springer, Heidelberg (1996)
12. Koike, H., Toyama, Y.: Comparison between inductionless induction and rewriting induction. In: Computer Software, vol. 17(6), pp. 1–12 (2000), in Japanese.
13. Kop, C.: Termination of LCTRSs. In: 13th International Workshop on Termination (WST), pp. 59–63 (2013)
14. Kop, C., Nishida, N.: Term rewriting with logical constraints. In: Fontaine, P. and Ringeissen, C. and Schmidt, R.A. (eds.) Frontiers of Combining Systems (FroCoS). LNCS, vol. 8152, pp. 343–358. Springer, Heidelberg (2013)
15. Kop, C., Nishida, N.: Automatic constrained rewriting induction towards verifying procedural programs. In: Programming Languages and Systems (APLAS). LNCS, vol. 8301. To Appear. Springer, Heidelberg (2014)
16. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.H., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
17. Nakabayashi, N., Nishida, N., Kusakari, K., Sakabe, T., Sakai, M.: Lemma generation method in rewriting induction for constrained term rewriting systems. In: Computer Software, vol. 28(1), pp. 173–189 (2010), in Japanese. (**)

18. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Automated termination analysis of Java bytecode by term rewriting. In: Lynch, C. (ed.) 21st International Conference on Rewriting Techniques and Applications (RTA). LIPIcs, vol. 6, pp. 259–276. Dagstuhl, Leibniz (2010)
19. Reddy, U.S.: Term rewriting induction. In: Stickel, M.E. (ed.) 10th International Conference on Automated Deduction (CADE). LNCS, vol. 449, pp. 162–177. Springer, Heidelberg (1990)
20. Sakata, T., Nishida, N., Sakabe, T.: On proving termination of constrained term rewrite systems by eliminating edges from dependency graphs. In: Kuchen, H. (ed.) Functional and Constraint Logic Programming (WFLP). LNCS, vol. 6816, pp. 138–155. Springer, Heidelberg (2011)
21. Sakata, T., Nishida, N., Sakabe, T., Sakai, M., Kusakari, K.: Rewriting induction for constrained term rewriting systems. In: IPSJ Transactions on Programming, vol. 2(2), pp. 80–96 (2009), in Japanese. (**)
22. Urso, P., Kounalis, E.: Sound generalizations in mathematical induction. In: Theoretical Computer Science, vol. 323(1-3), pp. 443–471. Elsevier (2004)
23. Walsh, T.: A divergence critic for inductive proof. In: Journal of Artificial Intelligence Research, vol. 4, pp. 209–235. (1996)

(**) Translations or summaries of marked Japanese papers are available at:
`http://www.trs.cm.is.nagoya-u.ac.jp/crisys/`

A Sufficient Conditions for Quasi-Reductivity

Definition 29 (Quasi-reductivity). *An LCTRS* $(\Sigma_{terms}, \Sigma_{theory}, \mathcal{I}, \mathcal{J}, \mathcal{R})$ *is quasi-reductive if for all* $s \in \mathcal{T}erms(\Sigma, \emptyset)$ *one of the following holds:*

- $s \in \mathcal{T}erms(\mathcal{C}ons, \emptyset)$ *(we say:* s *is a* constructor-term*);*
- *there is a* t *such that* $s \to_{\mathcal{R}} t$ *(we say:* s reduces*).*

Note the use of $\emptyset$ in this definition: we only consider *ground* terms, so a term $f(x)$ with $f \in \mathcal{D}$ is not required to reduce.

As observed in Footnote 3, this is equivalent to a more common definition:

Lemma 30. *An LCTRS is quasi-reductive if and only if all terms* $f(s_1, \ldots, s_n)$ *with* f *a defined or calculation symbol and all* $s_i \in \mathcal{T}erms(\mathcal{C}ons, \emptyset)$*, reduce.*

Proof. If the LCTRS is quasi-reductive, then each such $f(\overrightarrow{s})$ reduces, as it is not a constructor term. If the LCTRS is not quasi-reductive, then let $f(\overrightarrow{s})$ be a minimal ground irreducible non-constructor term. By minimality, all s_i must be constructor terms. If f is a constructor, then the whole term is a constructor term, contradiction, so f is either a defined symbol or a calculation symbol. □

Now, to prove quasi-reductivity, let us impose several necessary restrictions:

Definition 31 (Restrictions for quasi-reductivity). *We say that an LCTRS* $(\Sigma_{terms}, \Sigma_{theory}, \mathcal{I}, \mathcal{J}, \mathcal{R})$ *is:*

- left-linear *if for all rules* $\ell \to r\ [\varphi] \in \mathcal{R}$*: every variable in* ℓ *occurs only once;*
- constructor-sound *if there are ground constructor-terms for every sort* ι *such that some* $f : [\ldots \times \iota \times \ldots] \Rightarrow \kappa \in \mathcal{D}$ *(so* ι *is an input sort of a defined symbol);*
- left-value-free *if the left-hand sides of rules do not contain any values.*

Note that any LCTRS can be turned into a left-value-free one, by replacing a value v by a fresh variable and adding a constraint $x = v$ instead. Constructor-soundness seems quite natural, with a sort representing the set of ground constructor terms of that sort. Left-linearity is probably the greatest limitation; however, note that non-left-linear systems impose *syntactic* equality. In a rule

$$\mathsf{addtoset}(x, \mathsf{setof}(x, rest)) \to \mathsf{setof}(x, rest)$$

we can reduce $\mathsf{addtoset}(3{+}4, \mathsf{setof}(3{+}4, s))$ immediately to $\mathsf{setof}(3{+}4, s)$. However, we cannot reduce $\mathsf{addtoset}(3 + 4, \mathsf{setof}(4 + 3, s))$ with this rule. There is also no syntactic way to check for *inequality*. Therefore, it seems like we could better formulate this rule and its complement using constraints, or (if the sort of x has non-value constructors) by a structural check. The rule above and its complement could for instance become:

$$\begin{aligned} &\mathsf{addtoset}(x, \mathsf{setof}(y, rest)) \to \mathsf{setof}(y, rest) && [x = y] \\ &\mathsf{addtoset}(x, \mathsf{setof}(y, rest)) \to \mathsf{setof}(y, \mathsf{addtoset}(x, rest)) && [x \neq y] \end{aligned}$$

In this light, left-linearity also seems like a very natural restriction.

Note: in [5] a similar method is introduced to prove quasi-reductivity of a different style of constrained rewriting. There, however, the systems are additionally restricted to be value-safe*: the only constructors of sorts occurring in Σ_{theory} are values. Although all our current examples satisfy this requirement, We drop it here, because it is not necessary in the definition of LCTRSs.*

Lemma 32. *An LCTRS is either quasi-reductive, or it admits an irreducible term $f(\overrightarrow{s})$ with each $s_i \in \mathcal{T}erms(\mathcal{C}ons, \emptyset)$ and f a defined or calculation symbol.*

Proof. We prove: any ground term s is a constructor-term, or reduces, or has a subterm of the form required by the lemma (this subterm need not be strict). We prove this by induction on the form of s.

As s must be ground, it cannot be a variable, so can only have the form $f(s_1, \ldots, s_n)$. By the induction hypothesis, each s_i is either a constructor-term, reduces, or has a subterm of the required form. If any of the s_i reduces, then so does s and we are done. If any of the s_i has a subterm of the required form, then so does s. Otherwise, all s_i are constructor-terms.

Consider f. If f is a constructor, then s is a constructor-term. Otherwise, f is a defined or calculation symbol and s therefore has the required form. □

Theorem 33. *A constructor-sound LCTRS with rules $\mathcal{R}$ is quasi-reductive if and only if the following conditions both hold:*

- *the same LCTRS restricted to constructor-rules $\mathcal{R}' := \{f(\overrightarrow{\ell}) \to r[\varphi] \in \mathcal{R} \mid \forall i[\ell_i \in \mathcal{T}erms(\mathcal{C}ons, \mathcal{V})]\}$ is quasi-reductive;*
- *all constructor symbols with respect to $\mathcal{R}'$ are also constructors w.r.t. $\mathcal{R}$.*

Proof. Suppose $\mathcal{R}'$ is quasi-reductive, and constructor-terms are the same in either LCTRS. Then also $\mathcal{R}$ is quasi-reductive, as anything which reduces under $\mathcal{R}'$ also reduces under $\mathcal{R}$. Alternatively, suppose $\mathcal{R}$ is quasi-reductive.

Towards a contradiction, suppose $\mathcal{R}'$ admits constructor symbols which are not constructors in $\mathcal{R}$, so there is a symbol $f \in \Sigma_{terms}$ which is defined in $\mathcal{R}$ but not in $\mathcal{R}'$. Then there are no rules $f(\overrightarrow{l}) \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{\texttt{calc}}$ which match terms of the form $f(\overrightarrow{s})$ with all s_i ground constructor-terms. As we have limited interest to constructor-sound systems, and f is a defined symbol, such terms exist. As nothing matches $f(\overrightarrow{s})$ itself, and its strict subterms are constructor-terms so cannot be reduced, this term contradicts quasi-reductivity of $\mathcal{R}$!

For the first point, suppose towards a contradiction that $\mathcal{R}'$ is not quasi-reductive, yet $\mathcal{R}$ is, and the same terms are constructor-terms in either. By Lemma 32 there is some irreducible $f(s_1, \ldots, s_n)$ with all s_i constructor-terms and f not a constructor. As the s_i are constructor-terms, the rules in $\mathcal{R} \setminus \mathcal{R}'$ also cannot match! Thus, the term is also irreducible with $\to_{\mathcal{R}}$, contradiction. □

Thus, in the following algorithm we limit interest to *constructor TRSs*, where the immediate arguments in the left-hand sides of rules are all constructor-terms.

Main Algorithm. We assume given sequences $\iota_1, \ldots, \iota_n$ of theory sorts, $\kappa_1, \ldots, \kappa_m$ of sorts, and $x_1, \ldots, x_n$ of variables, with each $x_i : \iota_i \in \mathcal{V}$. Moreover, we assume given a set A of pairs $(\overrightarrow{s}, \varphi)$. Here, $\overrightarrow{s}$ is a sequence $s_1, \ldots, s_m$ of constructor-terms which do not contain values, such that $\vdash s_i : \kappa_i$, and φ is a

logical constraint. The s_i have no overlapping variables with each other or the x_j; that is, a term $f(x_1, \ldots, x_n, s_1, \ldots, s_m)$ would be linear. Variables in $\overrightarrow{x}$ and $\overrightarrow{s}$ may occur in φ, however.

Now, for $b \in \{\mathsf{term}, \mathsf{value}, \mathsf{either}\}$[8], define the function $OK(\overrightarrow{x}, A, b)$ as follows; this construction is well-defined by induction first on the number of function symbols occurring in A, second by the number of variables occurring in A, and third by the flag b (with $\mathsf{either} > \mathsf{term}, \mathsf{value}$). Only symbols in the s_i are counted for the first induction hypothesis, so not those in the constraints.

- if $m = 0$: let $\{y_1, \ldots, y_k\} = (\bigcup_{((),\varphi)\in A} Var(\varphi)) \setminus \{x_1, \ldots, x_n\}$;
  - if $\exists y_1 \ldots y_k. \bigvee_{((),\varphi)\in A} \varphi$ is valid, then **true**
  - else **false**

  Note that if $A = \emptyset$, this returns **false**.
- if $m > 0$ and $b = \mathsf{either}$, then consider κ_1. If κ_1 does not occur in Σ_{theory}, the result is:
  $\boldsymbol{OK}(\overrightarrow{x}, A, \mathsf{term})$
  If κ_1 occurs in Σ_{theory} and all constructors with output sort κ_1 are values, then the result is:
  $\boldsymbol{OK}(\overrightarrow{x}, A, \mathsf{value})$
  If κ_1 occurs in Σ_{theory} but there are also non-value constructors of sort κ_1, then let $V := \{(\overrightarrow{s}, \varphi) \in A \mid s_1 \text{ is a variable}\}$ and $T := \{(\overrightarrow{s}, \varphi) \in A \mid s_1$ is not a variable in $Var(\varphi)\}$. Note that V and T overlap in cases where s_1 is a variable not occurring in φ. The result of the function is:
  $\boldsymbol{OK}(\overrightarrow{x}, V, \mathsf{value}) \wedge \boldsymbol{OK}(\overrightarrow{x}, T, \mathsf{term})$
  In all cases, the recursive calls are defined, by the decrease in the third argument (and in the last possibly also in the first and second argument).
- if $m > 0$ and $b = \mathsf{value}$, then we assume that κ_1 occurs in Σ_{theory} and for all $(\overrightarrow{s}, \varphi) \in A$ the first term, s_1, is a variable (if not, the function result is **false**). Then let x_{n+1} be a fresh variable of sort κ_1, and let $A' := \{((s_2, \ldots, s_m), \varphi[s_1 := x_{n+1}]) \mid ((s_1, \ldots, s_n), \varphi) \in A\}$; the result is:
  $\boldsymbol{OK}(\boldsymbol{(}x_1\boldsymbol{,\ldots,}x_{n+1}\boldsymbol{)}\mathbf{,A',}\ \mathsf{either})$
  Note that A' has equally many function symbols as and fewer variables than A, and that we indeed have suitable sort sequences $(\iota_1, \ldots, \iota_n, \kappa_1)$ for the variables and $\kappa_2, \ldots, \kappa_m$ for the term sequences);
- if $m > 0$ and $b = \mathsf{term}$ and for all $(\overrightarrow{s}, \varphi) \in A$ the first term, s_1, is a variable, then let $A' := \{((s_2, \ldots, s_m), \varphi) \mid ((s_1, \ldots, s_m), \varphi) \in A \mid s_1 \notin Var(\varphi)\}$; the result is:
  $\boldsymbol{OK}(\boldsymbol{(}\overrightarrow{x}\mathbf{,A',}\mathsf{either}\boldsymbol{)})$
  A' has at most as many function symbols as and fewer variables than A.
- if $m > 0$ and $b = \mathsf{term}$ and there is some $(\overrightarrow{s}, \varphi) \in A$ where s_1 is not a variable, then let $f_1, \ldots, f_k$ be all non-value constructors with output sort κ_1 and let $A_1, \ldots, A_n$ be defined as follows: $A_i := \{(\overrightarrow{s}, \varphi) \in A \mid s_1$ is a variable or has the form $f_i(\overrightarrow{t})\}$.
  Now, for all i: if f_i has sort declaration $[\mu_1 \times \ldots \mu_p] \Rightarrow \kappa_1$, then we consider the new sort sequence $\overrightarrow{\kappa'}$ with $\overrightarrow{\kappa'} = (\mu_1, \ldots, \mu_p, \kappa_1, \ldots, \kappa_m)$; for every element

[8] This parameter indicates what constructor instantiations we should consider for s_1.

$(\overrightarrow{s}, \varphi) \in A_i$ we define:

- if $s_1 = f_i(t_1, \ldots, t_p)$, then $\overrightarrow{u} := (t_1, \ldots, t_p, s_2, \ldots, s_m)$ and $\psi := \varphi$
- if s_1 is a variable, then let $y_1, \ldots, y_p$ be fresh variables with sorts $\mu_1, \ldots, \mu_p$ respectively, and let $\overrightarrow{u} := (y_1, \ldots, y_p, s_2, \ldots, s_m)$ and $\psi := \varphi[s_1 := f_i(y_1, \ldots, y_p)]$.

We let B_i be the set of the corresponding $(\overrightarrow{u}, \psi)$ for all $(\overrightarrow{s}, \varphi) \in A_i$. Note that if A_i contains any element where s_i is not a variable, then B_i contains fewer function symbols (not counting constraints), so also fewer than A (as $A_i \subseteq A$). If all s_i are variables, then A_i is a strict subset of A, which misses at least one element where s_i contains a symbol, so also B_i has fewer symbols. Either way, we are safe defining the result as:
$\boldsymbol{OK}(\overrightarrow{x}, B_1, \mathsf{either}) \wedge \ldots \wedge \boldsymbol{OK}(\overrightarrow{x}, B_k, \mathsf{either})$

Lemma 34. *For any suitable $n, m, \overrightarrow{\iota}, \overrightarrow{\kappa}, \overrightarrow{x}, b$ and A such that $OK(\overrightarrow{x}, A, b) =$ `true`, we have, for any sequence $(s_1, \ldots, s_n)$ of values and any sequence $(t_1, \ldots, t_m)$ of constructor-terms: if one of the following conditions holds,*

- $b = \mathsf{either}$ *or* $m = 0$
- $b = \mathsf{value}$ *and* t_1 *is a value*
- $b = \mathsf{term}$ *and* t_1 *is not a value, and for all* $((u_1, \ldots, u_n), \varphi) \in A$*:* $u_i \notin Var(\varphi)$

then there is an element $((u_1, \ldots, u_m), \varphi) \in A$ and a substitution γ with $\gamma(x_i) = s_i$ such that:

- *each* $t_i = u_i\gamma$
- $\varphi\gamma$ *is a valid ground logical constraint*

Proof. We prove this by induction on the derivation of $OK(\overrightarrow{x}, A, b) =$ `true`. Let values $(s_1, \ldots, s_n)$ and constructor-terms $(t_1, \ldots, t_m)$ which satisfy the conditions be given.

If $m = 0$, then let $\psi := \bigvee_{((),\varphi)\in A} \varphi$. By definition of OK, we know that $\exists y_1 \ldots y_k.\psi$ is valid and $Var(\psi) \subseteq \{x_1, \ldots, x_n, y_1, \ldots, y_k\}$. That is, there are values $v_1, \ldots, v_k$ such that for all values $u_1, \ldots, u_n$ the ground constraint $\psi[\overrightarrow{y} := \overrightarrow{v}, \overrightarrow{x} := \overrightarrow{u}]$ holds. In particular, we can take $\overrightarrow{s}$ for $\overrightarrow{u}$. Define $\gamma := [y_1 := v_1, \ldots, y_k := v_k, x_1 := s_1, \ldots, x_n := s_n]$. Then $\psi\gamma$ is valid, and since $\psi\gamma$ is ground we can identify a valid part of the disjunction, so some $((), \varphi) \in A$ where $\varphi\gamma$ is a valid ground logical constraint. This is exactly what the lemma requires!

If $m > 0$ and $b = \mathsf{either}$ and κ_1 does not occur in Σ_{theory}, then $OK(\overrightarrow{x}, A, \mathsf{term})$ holds. Since there are no values of sort κ_1, the term t_1 is not a value; for the same reason, variables of sort κ_1 cannot occur in Σ_{theory}. Thus, the conditions for the induction hypothesis are satisfied; we find a suitable γ and $(\overrightarrow{u}, \varphi)$.

If $m > 0$ and $b = \mathsf{either}$ and κ_1 does occur in Σ_{theory}, and all constructors with output sort κ_1 are values, then $OK(\overrightarrow{x}, A, \mathsf{value})$ holds; moreover, t_1 is necessarily a value, so we can again apply the induction hypothesis.

If $m > 0$ and $b = \mathsf{either}$ and there are both values and other constructors with output sort κ_1, then both $OK(\overrightarrow{x}, T, \mathsf{term})$ and $OK(\overrightarrow{x}, V, \mathsf{value})$ must hold. If t_1 is a value, then the conditions to apply the induction hypothesis with V are satisfied; if not, the conditions to apply it with T are satisfied! Since both T and V are subsets of A, this results in a suitable element and substitution.

If $m > 0$ and $b = \mathsf{value}$, then by the condition we can assume that t_1 is a value. Let x_{n+1} be a fresh variable of sort κ_1, and $A' := \{((u_2, \ldots, u_m), \varphi[u_1 := x_{n+1}]) \mid (\overrightarrow{u}, \varphi) \in A\}$. Thus, we can apply the induction hypothesis with $n + 1, m - 1, (\overrightarrow{\iota}, \kappa_1), (\kappa_2, \ldots, \kappa_m), (x_1, \ldots, x_{n+1}), A', \mathsf{either}$ and $(s_1, \ldots, s_n, t_1)$ and $(t_1, \ldots, t_m)$. We obtain an element $(\overrightarrow{u}, \varphi) \in A'$ and γ such that $\gamma(x_i) = s_i$ for $1 \leq i \leq n$ and $\gamma(x_{n+1}) = t_1$ and each $t_{i+1} = u_i\gamma$, and $\varphi\gamma$ is a valid ground logical constraint. Now, $(\overrightarrow{u}, \varphi)$ can be written as $((w_2, \ldots, w_n), \varphi'[w_1 := x_{n+1}])$ for some $(\overrightarrow{w}, \varphi') \in A$. So let δ be the substitution $\gamma \cup [w_1 := \gamma(x_{n+1})]$. Then, noting that by linearity w_1 may not occur in the other w_i, and that because x_{n+1} does not occur in φ' we have $\varphi'\delta = \varphi\gamma$, indeed each further $\delta(x_i) = \gamma(x_i) = s_i$ and $t_1 = \gamma(x_{n+1}) = \delta(w_1) = w_1\delta$ and $t_i = u_i\gamma = w_i\delta$ for larger i.

If $m > 0$ and $b = \mathsf{term}$ and all $(\overrightarrow{u}, \varphi) \in A$ have a variable for u_1, then we can assume that t_1 is not a value. We use the induction hypothesis and find a suitable element $((u_2, \ldots, u_m), \varphi) \in A'$ and substitution γ with $\gamma(x_i) = s_i$ for all $1 \leq i \leq n$, such that each $t_i = u_i\gamma$ $(i > 1)$ and $\varphi\gamma$ is a valid ground logical constraint. Choose $\delta := \gamma \cup [u_1 := t_1]$ (this is safe by linearity). The same requirements are satisfied, and also $t_1 = u_1\delta$!

Finally, suppose $m > 0$ and $b = \mathsf{term}$ and A has some element $(\overrightarrow{u}, \varphi)$ where u_1 is not a variable; by assumption it is also not a value. By the conditions, we may assume that always $u_1 \notin \mathit{Var}(\varphi)$. Let $f_1, \ldots, f_k$ be all constructors in $\Sigma_{terms} \setminus \mathcal{V}al$ with output sort κ_1. Since also t_1 is not a value, but is a ground constructor term, it can only have the form $f_p(w_1, \ldots, w_k)$ for some $p, \overrightarrow{w}$. Observing that $\mathit{OK}(\overrightarrow{x}, B_p, \mathsf{either})$ must hold, we use the induction hypothesis, for $\overrightarrow{x}$, $\overrightarrow{s}$ and $(w_1, \ldots, w_k, t_2, \ldots, t_m)$, and find both a suitable tuple $((q_1, \ldots, q_k, u_2, \ldots, u_m), \varphi) \in B_p$ and a substitution γ which respects φ, maps $\overrightarrow{x}$ to $\overrightarrow{s}$ and has $q_i\gamma = w_i$ and $u_j\gamma = t_j$ for all i, j. Now, by definition of B_p, we have $((u_1, \ldots, u_m), \varphi) \in A$ for some u_1 which is either a variable (in which case all q_i are fresh variables), or $u_1 = f_p(q_1, \ldots, q_k)$.

In the case of a variable, we note that u_1 cannot occur in any of the other u_i by the linearity requirement, nor in φ by the conditions. Thus, we can safely assume that u_1 does not occur in the domain of γ, and choose $\delta := \gamma \cup [u_1 := f(w_1, \ldots, w_k)]$. Then $\varphi\delta = \varphi\gamma$ is still a valid ground constraint, each $s_i = x_i\gamma = x_i\delta$ and for $i > 0$ also $t_i = u_i\gamma = u_i\delta$. Finally, $t_1 = f_p(w_1, \ldots, w_k) = u_1\delta$ as required. In the alternative case that $u_1 = f_p(q_1, \ldots, q_k)$, we observe that γ already suffices: each $s_i = x_i\gamma$, for $i > 1$ we have $t_i = u_i\gamma$ and $t_1 = f_p(w_1, \ldots, w_k) = f_p(q_1\gamma, \ldots, q_k\gamma) = u_1\gamma$. □

Theorem 35. *A left-linear and left-value-free constructor LCTRS with rules $\mathcal{R}$ is quasi-reductive if for all defined and calculation symbols f: $\mathit{OK}((), A_f, \mathsf{either})$ holds, where $A_f := \{(\overrightarrow{l}, \varphi) \mid \ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{\mathtt{calc}} \wedge \ell = f(\overrightarrow{l})\}$.*

Proof. By Lemma 32 it suffices to prove that all terms of the form $f(s_1, \ldots, s_n)$ can be reduced, where f is a defined or calculation symbol and all s_i are constructor-terms. This holds if there is a rule $f(\ell_1, \ldots, \ell_n) \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{\mathtt{calc}}$ and a substitution γ such that each $s_i = \ell_i\gamma$ and $\varphi\gamma$ is a satisfiable constraint. By Lemma 34 (which we can apply because left-hand sides of rules are linear and value-free), that's exactly the case if $\mathit{OK}((), A_f, \mathsf{either}) = \mathtt{true}$! □

B Other Examples

The following examples are obtained from [17] by an adaptation to LCTRSs. Here, we use only the transformation from Section 3.1, to stay close to [17]. In particular, we do not use the return constructor.

Example 36. Consider the following specification of `double` of non-negative integers and a program implementing it using a loop:

$$\begin{array}{ll} \mathsf{double}(x) = 0 & \text{if } x \leq 0 \\ \mathsf{double}(x) = 2 + \mathsf{double}(x-1) & \text{otherwise} \end{array}$$

```
int double1(int x) {
    int i = 1, z = 0;
    for (i = 1; i <= x; i++) {
        z = sucsuc(z);
    }
    return z;
}

int sucsuc(int x){
    return x + 2;
}
```

The program originates from [17], and is very artificial to show the necessity of an extension of the method in [10]. To show that our method covers the examples for which the (extended) method in [17] is successful, we deal with this artificial program. This program and the specification are transformed into the LCTRS $\mathcal{R}_{double}$:

$$\begin{array}{rll} \mathsf{double}(x) & \to 0 & [x \leq 0] \\ \mathsf{double}(x) & \to 2 + \mathsf{double}(x-1) & [x > 0] \\ \mathsf{double1}(x) & \to \mathsf{u}(x, v_1, v_2) & [v_1 = 1 \wedge v_2 = 0] \\ \mathsf{u}(x, i, z) & \to \mathsf{u}(x, i+1, \mathsf{sucsuc}(z)) & [i \leq x] \\ \mathsf{u}(x, i, z) & \to z & [\neg i \leq x] \\ \mathsf{sucsuc}(x) & \to x + 2 & \end{array}$$

Our method succeeds in proving that $\mathsf{double}(x) \approx \mathsf{double1}(x)$ is an inductive theorem of $\mathcal{R}_{double}$ by generating the following generalization:

$$\mathsf{u}(x, i_1, z_1) \approx 2 + \mathsf{u}(x_0, i_0, z_0) \; [x > 0 \wedge x_0 = x - 1 \wedge i_0 = w_1 + 1 \wedge z_0 = w_2 + 2 \wedge i_0 \leq x \wedge i_1 = i_0 + 1 \wedge z_1 = z_0 + 2]$$

Example 37. Consider the following specification of Fibonacci's numbers and a program implementing it using a loop:

$$\begin{array}{ll} \mathsf{fib}(x) = 0 & \text{if } x \leq 0 \\ \mathsf{fib}(x) = 1 & \text{if } x = 1 \\ \mathsf{fib}(x) = \mathsf{fib}(x-1) + \mathsf{fib}(x-2) & \text{otherwise} \end{array}$$

```
int fib1(int x) {
    int i = 0, y = 0, z = 1;
    for (i = 0; i < x; i++) {
        int tmp = y;
        y = z;
        z += tmp;
    }
    return y;
}
```

This program and the specification are transformed into the LCTRS $\mathcal{R}_{fib}$:

$$\begin{aligned}
\mathsf{fib}(x) &\to \mathsf{0} && [x \leq \mathsf{0}]\\
\mathsf{fib}(x) &\to \mathsf{1} && [x = \mathsf{1}]\\
\mathsf{fib}(x) &\to \mathsf{fib}(x-\mathsf{1}) + \mathsf{fib}(x-\mathsf{2}) && [x > \mathsf{1}]\\
\mathsf{fib1}(x) &\to \mathsf{u}(x, v_1, v_2, v_3) && [v_1 = \mathsf{0} \wedge v_2 = \mathsf{0} \wedge v_3 = \mathsf{1}]\\
\mathsf{u}(x, i, y, z) &\to \mathsf{u}(x, i+\mathsf{1}, z, z+y) && [i < x]\\
\mathsf{u}(x, i, y, z) &\to y && [\neg i < x]
\end{aligned}$$

Our method succeeds in proving that $\mathsf{fib}(x) \approx \mathsf{fib1}(x)$ is *not* an inductive theorem of $\mathcal{R}_{fib}$. If v_1 is set to $\mathsf{1}$ instead of $\mathsf{0}$ we *do* have an inductive theorem, which is proved with the following generalization:

$$\mathsf{u}(x, i_1, z_0, z_1) \approx \mathsf{u}(x_0, i_0, w_3, z_0) + \mathsf{u}(x_1, w_1, w_2, w_3) \quad [x - \mathsf{2} \geq \mathsf{0} \wedge x_0 = x - \mathsf{1} \wedge x_1 = x - \mathsf{2} \wedge z_0 = w_2 + w_3 \wedge i_0 = w_1 + \mathsf{1} \wedge z_1 = w_3 + z_0 \wedge i_1 = 1_0 + \mathsf{1}]$$