Bounded Rewriting Induction for LCSTRSs

Kasper Hagens
Radboud University Nijmegen, Netherlands
kasper.hagens@ru.nl

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

Rewriting Induction (RI) is a method for inductive theorem proving in equational reasoning, introduced in 1990 by Reddy. Using Logically Constrained Simply-typed Term Rewriting Systems (LCSTRSs) makes it into an interesting tool for program verification (in particular program equivalence), as LCSTRSs closely describe real-life programming. Correctness of RI depends on well-founded induction, and one of the core obstacles for obtaining a practically useful proof system is to find suitable well-founded orderings automatically. Using naive approaches, induction hypotheses must be oriented within the well-founded ordering, leading to very strong ordering requirements, which in turn, severely limits the proof capacity of RI. Here, we introduce bounded RI: an adaption of RI for LCSTRSs where such requirements are being minimized.

1 Introduction

Rewriting Induction (RI) is a proof system for showing equations $s \approx t$ to be inductive theorems, meaning that every variable-free instance of $s \approx t$ is related by $\leftrightarrow_{\mathcal{R}}^*$: the reflexive, transitive closure of $\leftrightarrow_{\mathcal{R}} = \rightarrow_{\mathcal{R}} \cup \leftarrow_{\mathcal{R}}$ (and with $\mathcal{R}$ the set of rewrite rules that completely describe the reduction behavior of s and t). RI was introduced by Reddy [10], as a method to validate inductive proof procedures based on Knuth-Bendix completion. Classically, it is used in equational reasoning to prove properties of inductively defined mathematical structures such as natural numbers or lists. For example, one could use RI to prove an equation $\mathsf{add}(x,y) \approx \mathsf{add}(y,x)$, expressing commutativity of addition on the natural numbers. It was adapted to constrained rewriting [6], and recently to higher-order constrained rewriting [9]. These formalisms closely relate to real-life programming and therefore have a natural place in the larger toolbox for program verification. Programs are represented by term rewriting systems, and inductive theorems provide an interpretation of program equivalence.

Why constrained rewriting? Using RI for program equivalence somewhat differs from the standard setting in equational reasoning where, for example, the Peano axioms are used to prove statements about the natural numbers. In our case, we are not so much interested in proving properties about the natural numbers themselves, but about programs that operate on them. Of course, we can express the Peano axioms as rewrite rules and use this to define programs on natural numbers. However, the disadvantage of this approach is that we also have to define add and mul to represent $+$ and $*$. In this way, studying program equivalence becomes a cumbersome experience, getting involved in complicated interactions between add, mul and the program definition itself. Like in real-life programming, we want to treat the natural numbers as being given for free. With standard term rewriting this is not possible.

Constrained rewriting provides a solution here, as they natively support primitive data structures, such as natural numbers and integers. This makes it possible to distinguish between the actual program definition (represented by rewrite rules), and underlying data structures with their operators (represented by distinguished terms with pre-determined semantical interpretations). This allows us to shift some of

Submitted to:
WPTE 2025

© K. Hagens & C. Kop
This work is licensed under the
Creative Commons Attribution License.

the proof-burden from the rewriting side to the semantical side (which e.g. could be handled by an SMT solver).

In constrained rewriting, rewrite rules are of the shape $s \to t\ [\varphi]$ where the boolean constraint φ acts as a guard, managing control flow over primitive data structures (such as the natural numbers). Here, we will consider Logically Constrained Simply-typed Term Rewriting Systems (LCSTRSs), which concerns applicative higher-order rewriting (without λ abstractions) and first-order constraints [8]. In particular, we will build on our earlier work [9] where we defined RI for LCSTRSs.

Rewriting Induction and Termination The name Rewriting Induction refers to the principle that for a terminating rewrite system $\mathcal{R}$, the rewrite relation $\to^+_{\mathcal{R}}$ defines a well-founded order on the set of all terms, and therefore can be used for proofs by well-founded induction. In many cases, however, we will need a well-founded order $\succ$ which is strictly larger than $\to^+_{\mathcal{R}}$. This is because the role of induction hypothesis in RI is also taken by equations, which must be applied like a rewrite rule, in a decreasing direction w.r.t. $\succ$. That is, we are only allowed to use an induction hypothesis $s \approx t$ if $s \succ t$ or $t \succ s$ holds. Consequently, termination of $\mathcal{R}$ itself is not enough, since equations are not usually orientable by $\to^+_{\mathcal{R}}$.

One solution to this problem is to for instance let $\succ = \to^+_{\mathcal{R}\cup\{s\to t\}}$; or, in the case of multiple induction hypothesis, to collect all the corresponding rewrite rules in a set $\mathcal{H}$ and use $\succ = \to^+_{\mathcal{R}\cup\mathcal{H}}$. However, doing this leaves us with a proof obligation to show termination of $\mathcal{R}\cup\mathcal{H}$. Even if we already know that $\mathcal{R}$ is terminating, it may not be easy or even possible to prove that the same holds for $\mathcal{R}\cup\mathcal{H}$ (think for instance of an induction hypothesis $\mathsf{add}(x,y) \approx \mathsf{add}(y,x)$, which is not orientable in either direction). In such a situation a RI proof might get stuck.

Our goal is to redefine RI for LCSTRSs in such a way that we minimize the termination requirements. As already observed by Reddy [10], we do not necessarily need every induction hypothesis being oriented, as long if we can guarantee that an induction rule $s \to t$ is only applied to terms $\succ$-smaller than s. For this, it is not required to choose the well-founded ordering $\succ = \to^+_{\mathcal{R}\cup\mathcal{H}}$. Reddy proposed to use modulo rewriting to build a well-founded $\succ$ which may not need to contain all induction rules. This approach was investigated by Aoto, who introduced several extensions of RI for first-order unconstrained rewriting [1, 2, 3]. Here we will follow a strategy along the same idea: by redefining RI we can construct a well-founded relation $\succ$ during the RI process, aiming to keep it as small as possible.

2 Preliminaries

2.1 Logically Constrained Simply Typed Rewriting Systems

Types and Terms Assume a set of sorts (base types) $\mathcal{S}$; the set $\mathcal{T}$ of types is defined by the grammar $\mathcal{T} ::= \mathcal{S} \mid \mathcal{T} \to \mathcal{T}$. Here, $\to$ is right-associative, so all types may be written as $type_1 \to \ldots \to type_m \to sort$ with $m \geq 0$. We also assume a subset $\mathcal{S}_{theory} \subseteq \mathcal{S}$ of *theory sorts* (e.g., int and bool), and define the *theory types* by $\mathcal{T}_{theory} ::= \mathcal{S}_{theory} \mid \mathcal{S}_{theory} \to \mathcal{T}_{theory}$. Each theory sort $\iota \in \mathcal{S}_{theory}$ is associated with a non-empty set $\mathcal{I}_\iota$ (e.g., $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$, the set of all integers), and we let $\mathcal{I}_{\iota\to\sigma}$ be the set of functions from $\mathcal{I}_\iota$ to $\mathcal{I}_\sigma$.

We assume a signature Σ of *function symbols* and a disjoint set $\mathcal{V}$ of variables, and a function *typeof* from $\Sigma\cup\mathcal{V}$ to $\mathcal{T}$; we require that there are infinitely many variables of all types. The set of terms $T(\Sigma,\mathcal{V})$ over Σ and $\mathcal{V}$ are the expressions in $\mathbb{T}$ – defined by the grammar $\mathbb{T} ::= \Sigma \mid \mathcal{V} \mid \mathbb{T}\ \mathbb{T}$ – that are *well-typed*: if $s :: \sigma \to \tau$ and $t :: \sigma$ then $s\ t :: \tau$, and $a :: typeof(a)$ for $a \in \Sigma\cup\mathcal{V}$.

Application is left-associative, which allows all terms to be written in a form $a\ t_1 \cdots t_n$ with $a \in \Sigma\cup\mathcal{V}$ and $n \geq 0$. Writing $t = a\ t_1 \cdots t_n$, we define $head(t) = a$. For a term t, let $Var(t)$ be the set of variables in

t. A term t is *ground* if $Var(t) = \emptyset$. We say that a type is *inhabited* if there are ground terms of that type. We assume that Σ is the disjoint union $\Sigma_{theory} \uplus \Sigma_{terms}$, where $typeof(\mathsf{f}) \in \mathcal{T}_{theory}$ for all $\mathsf{f} \in \Sigma_{theory}$. Each $\mathsf{f} \in \Sigma_{theory}$ has an interpretation $[\![\mathsf{f}]\!] \in \mathcal{I}_{typeof(\mathsf{f})}$. For example, a theory symbol $* :: \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ may be interpreted as multiplication on $\mathbb{Z}$. We use infix notation for the binary symbols, or use $[\mathsf{f}]$ for prefix or partially applied notation (e.g., $[+]\ x\ y$ and $x + y$ are the same).

Symbols in Σ_{terms} do not have an interpretation since their behavior will be defined through the rewriting system. Values are theory symbols of base type, i.e. $\mathcal{V}al = \{v \in \Sigma_{theory} \mid typeof(v) \in \mathcal{S}_{theory}\}$. We assume there is exactly one value for each element of $\mathcal{I}_\iota$ ($\iota \in \mathcal{S}_{theory}$). Elements of $T(\Sigma_{theory}, \mathcal{V})$ are called *theory terms*. For *ground* theory terms, we define $[\![s\ t]\!] = [\![s]\!]([\![t]\!])$. We fix a theory sort bool with $\mathcal{I}_{\mathsf{bool}} = \{\top, \bot\}$. A *constraint* is a theory term $s :: \mathsf{bool}$, such that $typeof(x) \in \mathcal{S}_{theory}$ for all $x \in Var(s)$.

Example 1 In this text we always use $\mathcal{S}_{theory} = \{\mathsf{int}, \mathsf{bool}\}$ and $\Sigma_{theory} = \{+, -, *, <, \leq, >, \geq, =, \wedge, \vee, \neg, \mathtt{true}, \mathtt{false}\} \cup \{\mathsf{n} \mid n \in \mathbb{Z}\}$, with $+, -, * :: \mathsf{int} \to \mathsf{int} \to \mathsf{int}$, $<, \leq, >, \geq, = :: \mathsf{int} \to \mathsf{int} \to \mathsf{bool}$, $\wedge, \vee :: \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$, $\neg :: \mathsf{bool} \to \mathsf{bool}$, $\mathtt{true}, \mathtt{false} :: \mathsf{bool}$ and $\mathsf{n} :: \mathsf{int}$. We let $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$, $\mathcal{I}_{\mathsf{bool}} = \{\top, \bot\}$ and interpret all symbols as expected. The values are $\mathtt{true}, \mathtt{false}$ and all n. Theory terms are for instance $x + 3$, $\mathtt{true}$ and $7 * 0$. The latter two are ground. We have $[\![7 * 0]\!] = 0$. Let $x \in \mathcal{V}$ of type int. Then theory term $x > 0$ is a constraint, but the theory term $(f\ x) > 0$ with $f \in \mathcal{V}$ of type $\mathsf{int} \to \mathsf{int}$ is not (since $typeof(f) \notin \mathcal{S}_{theory}$), nor is $[>]\ 0 :: \mathsf{int} \to \mathsf{bool}$ (since constraints must have type bool).

Substitutions, contexts and subterms A substitution is a type-preserving mapping $\gamma : \mathcal{V} \to T(\Sigma, \mathcal{V})$. The domain of a substitution is defined as $dom(\gamma) = \{x \in \mathcal{V} \mid \gamma(x) \neq x\}$, and the image of a substitution as $im(\gamma) = \{\gamma(x) \mid x \in dom(\gamma)\}$. A substitution on finite domain $\{x_1, \ldots, x_n\}$ is often denoted $[x_1 := s_1, \ldots, x_n := s_n]$. A substitution γ is extended to a function $s \mapsto s\gamma$ on terms by placewise substituting variables in the term by their image: **(i)** $t\gamma = t$ if $t \in \Sigma$, **(ii)** $t\gamma = \gamma(t)$ if $t \in \mathcal{V}$, and **(iii)** $(t_0\ t_1)\gamma = (t_0\gamma)\ (t_1\gamma)$. If $M \subseteq T(\Sigma, \mathcal{V})$ then $\gamma(M)$ denotes the set $\{t\gamma \mid t \in M\}$. A *ground substitution* is a substitution γ such that for all $x \in dom(\gamma)$ of an inhabited type, $\gamma(x)$ is a ground term. A substitution γ *respects* a constraint φ if $\gamma(Var(\varphi)) \subseteq \mathcal{V}al$ and $[\![\varphi\gamma]\!] = \top$. We say that a constraint φ is *satisfiable* if there exists a substitution γ that respects φ, and is *valid* if $[\![\varphi\gamma]\!] = \top$ for all ground substitutions γ that map each $x \in Var(\varphi)$ to values. Let $\Box_1, \ldots, \Box_n$ be fresh, typed constants ($n \geq 1$). A context $C[\Box_1, \ldots, \Box_n]$ (or just: C) is a term in $T(\Sigma \cup \{\Box_1, \ldots, \Box_n\}, \mathcal{V})$ in which each $\Box_i$ occurs exactly once. The term obtained from C by replacing each $\Box_i$ by a term t_i of the same type is denoted by $C[t_1, \ldots, t_n]$. We say that t is a *subterm* of s, notation $s \trianglerighteq t$, if either $s = t$ or $s = a\ s_1 \cdots s_n$ and $s_i \trianglerighteq t$ for some i. We say that t is a *strict* subterm of s, notation $s \triangleright t$, if $s \trianglerighteq t$ and $s \neq t$. (Here we deviate from the typical norm in higher-order rewriting, since we do *not* for instance include s as a subterm of a term $s\ t$. This is deliberate, because we are only interested in "maximally applied" subterms.)

Rewrite rules and reduction relation A rewrite rule is an expression $\ell \to r\ [\varphi]$. Here ℓ and r are terms of the same type, ℓ has a form $\mathsf{f}\ \ell_1 \cdots \ell_k$ with $\mathsf{f} \in \Sigma$ and $k \geq 0$, φ is a constraint and $Var(r) \subseteq Var(\ell) \cup Var(\varphi)$. If $\varphi = \mathtt{true}$, we just write $\ell \to r$. In what follows we fix a signature Σ. We define the set of *calculation rules* as: $\mathcal{R}_{calc} = \{\mathsf{f}\ x_1 \cdots x_m \to y\ [y = \mathsf{f}\ x_1 \cdots x_m] \mid \mathsf{f} \in \Sigma_{theory} \setminus \mathcal{V}al$ with $typeof(\mathsf{f}) = \iota_1 \to \ldots \to \iota_m \to \kappa\}$. We furthermore assume a set of rewrite rules $\mathcal{R}$ satisfying the following properties

- for all $\ell \to r\ [\varphi] \in \mathcal{R}$: ℓ is not a theory term (such rules are contained in $\mathcal{R}_{calc}$)
- for all $\mathsf{f}\ \ell_1 \cdots \ell_k \to r\ [\varphi]$, $\mathsf{g}\ \ell'_1 \cdots \ell'_n \to r'\ [\psi] \in \mathcal{R} \cup \mathcal{R}_{calc}$: if $\mathsf{f} = \mathsf{g}$ then $k = n$

The latter restriction blocks us for instance from having both a rule $\mathsf{append}\ \mathsf{nil} \to \mathsf{id}$ and a rule $\mathsf{append}\ (\mathsf{cons}\ x\ y)\ z \to \mathsf{cons}\ x\ (\mathsf{append}\ y\ z)$. While such rules would normally be allowed in higher-order

rewriting, we need to impose this limitation for the notion of *quasi-reductivity* to make sense, as discussed in [9]. This does not really limit expressivity, since we can use a strategy similar to η-expansion, padding both sides of a rule with variables, e.g., replacing the first rule above by $\mathsf{append}\ \mathsf{nil}\ x \to \mathsf{id}\ x$.

Elements of $\mathcal{D} = \{\mathsf{f} \in \Sigma \mid \exists \mathsf{f}\ \ell_1 \cdots \ell_k \to r\ [\varphi] \in \mathcal{R}\}$ are called *defined symbols*. Elements of $\mathcal{C} = \mathcal{V}al \cup (\Sigma_{terms} \setminus \mathcal{D})$ are called *constructors*. Elements of $\Sigma_{calc} = \Sigma_{theory} \setminus \mathcal{V}al$ are called *calculation symbols*.

For every defined or calculation symbol $\mathsf{f} :: \sigma_1 \to \ldots \to \sigma_m \to \iota$ with $\iota \in \mathcal{S}$, we let $ar(\mathsf{f}) \leq m$ be the number such that for every rule of the form $\mathsf{f}\ \ell_1 \cdots \ell_k \to r\ [\varphi]$ in $\mathcal{R} \cup \mathcal{R}_{calc}$ we have $ar(\mathsf{f}) = k$. (By the restrictions above, this number exists.) For all constructors $\mathsf{f} \in \mathcal{C}$, we define $ar(\mathsf{f}) = \infty$.

The reduction relation $\to_{\mathcal{R}}$ is defined by:

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

Note that by definition of context, reductions may occur at the head of an application. For example, if $\mathsf{append}\ \mathsf{nil} \to \mathsf{id} \in \mathcal{R}$, then we could reduce $\mathsf{append}\ \mathsf{nil}\ s\ \to_{\mathcal{R}} \mathsf{id}\ s$.

LCSTRS An LCSTRS is a pair $(T(\Sigma,\mathcal{V}), \to_{\mathcal{R}})$ generated by $(\mathcal{S}, \mathcal{S}_{theory}, \Sigma_{terms}, \Sigma_{theory}, \mathcal{V}, typeof, \mathcal{I}, [\![\cdot]\!], \mathcal{R})$. We often refer to an LCSTRS by $\mathcal{L} = (\Sigma, \mathcal{R})$, or just $\mathcal{R}$, leaving the rest implicit.

We say $\mathcal{L} = (\Sigma, \mathcal{R})$ is *terminating* if there is no infinite reduction sequence $s_0 \to_{\mathcal{R}} s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} \ldots$ for any $s_0 \in T(\Sigma,\mathcal{V})$. A term s *has normal form* t if $s \to_{\mathcal{R}}^* t$ and t cannot be reduced. We say $\mathcal{L}$ is *weakly normalising* if every term has at least one normal form. Note that termination implies weak normalisation, but not the other way around.

Example 2 Let $\mathcal{R}$ consist of the following rules

$$\textbf{(R1)}\ \mathsf{recdown}\ f\ n\ i\ a \to a \quad [i < n] \qquad \textbf{(R2)}\ \mathsf{recdown}\ f\ n\ i\ a \to f\ i\ (\mathsf{recdown}\ f\ n\ (i-1)\ a) \quad [i \geq n]$$
$$\textbf{(R3)}\ \mathsf{tailup}\ f\ i\ m\ a \to a \quad [i > m] \qquad \textbf{(R4)}\ \mathsf{tailup}\ f\ i\ m\ a \to \mathsf{tailup}\ f\ (i+1)\ m\ (f\ i\ a) \quad [i \leq m]$$

The intuition is that recdown and tailup define recursors that can be used to describe a class of simple real-life programs which compute a return-value using a recursive or tail-recursive procedure. More specifically, we consider programs that use a loop index i, being decreased/increased by 1 during each recursive call, until i reaches a value below lower bound n or above upper bound m. For example, we can represent the following two programs (both computing the factorial function $x \mapsto \prod_{i=1}^{x}$ when restricting to non-negative integers)

```
int factRec(int x){
    if (x >= 1)
        return(x*factRec(x-1));
    else
        return 1; }
```

```
int factTail(int x){
    int a = 1; int i = 1;
    while (i<=x){
        a = i*a; i = i+1;}
    return a; }
```

with recdown and tailup by introducing rewrite rules $\mathsf{factRec}\ x \to \mathsf{recdown}\ [*]\ 1\ x\ 1$ (loop index x and lower bound 1) and $\mathsf{factTail}\ x \to \mathsf{tailup}\ [*]\ 1\ x\ 1$ (loop index 1 and upper bound x).

In general, we can think about $\mathsf{recdown}\ [*]\ n\ i\ a$ to compute $(\prod_{k=n}^{i} k) \cdot a$ and we can think about $\mathsf{tailup}\ [*]\ j\ m\ b$ to compute $(\prod_{k=j}^{m} k) \cdot b$. Hence, all ground instances of $\mathsf{recdown}\ [*]\ n\ i\ a$ and $\mathsf{tailup}\ [*]\ n\ i\ a$ produce the same result. We will prove this with bounded rewriting induction in subsection 3.2; not just for $f = *$, but for arbitrary function $f :: \mathsf{int} \to \mathsf{int} \to \mathsf{int}$.

Considering $\mathcal{R} = \{\textbf{(R1)}, \textbf{(R2)}, \textbf{(R3)}, \textbf{(R4)}\}$ we have $\mathcal{S} = \mathcal{S}_{theory} = \{\mathsf{int}, \mathsf{bool}\}$, $\Sigma_{terms} = \{\mathsf{recdown}, \mathsf{tailup} :: (\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}\}$ and Σ_{theory} as in Example 1. Furthermore, $\Sigma_{calc} = \{+, -, *, <, \leq, >, \geq, =, \wedge, \vee\}$, $\mathcal{D} = \Sigma_{terms}$ and $\mathcal{C} = \mathcal{V}al = \{\mathtt{true}, \mathtt{false}\} \cup \{\mathsf{n} \mid n \in \mathbb{Z}\}$. Substitution $\gamma = [i := 1,\ n := 0]$ induces a reduction $\mathsf{recdown}\ f\ 0\ 1\ a \to_{\mathcal{R}} f\ 1\ (\mathsf{recdown}\ f\ 0\ (1-1)\ a) \to_{\mathcal{R}_{calc}}$

$f\ 1\ (\mathsf{recdown}\ f\ 0\ 0\ a) \to_{\mathcal{R}} f\ 1\ (f\ 0\ (\mathsf{recdown}\ f\ 0\ (0{-}1)\ a)) \to_{\mathcal{R}_{calc}} f\ 1\ (f\ 0\ (\mathsf{recdown}\ f\ 0\ (-1)\ a))$ $\to_{\mathcal{R}} f\ 1\ (f\ 0\ a)$. It is easy to check that $(\mathsf{tailup}\ f\ n\ i\ a)\gamma = \mathsf{tailup}\ f\ 0\ 1\ a$ also reduces to $f\ 1\ (f\ 0\ a)$.

We will limit our interest to *quasi-reductive* LCSTRSs (defined below), which is needed to guarantee correctness of RI. Intuitively, this property expresses that pattern matching on ground terms is exhaustive (i.e. there are no missing reduction cases). For example, the rewrite system $\mathcal{R} = \{\textbf{(R1)}, \textbf{(R2)}\}$ is quasi-reductive because $i < n$ and $i \geq n$ together cover all ground instances of $\mathsf{recdown}\ f\ n\ i\ a$. But if we, for example, replace **(R2)** by $\mathsf{recdown}\ f\ n\ i\ a \to f\ i\ (\mathsf{recdown}\ f\ n\ (i-1)\ a)\ [i > n]$ then it is not, as we are missing all ground reduction cases for $i = n$ (for example $\mathsf{recdown}\ [*]\ 0\ 0\ 0$ does not reduce anymore).

For first-order the LCTRSs, quasi-reductivity is achieved by demanding that there are no other ground normal forms than the ground constructor terms $T(\mathcal{C}, \emptyset)$. For higher-order LCSTRSs, however, this approach does not work as we can have ground normal forms with partially applied defined symbols (for example, $\mathsf{recdown}\ [+]$). Hence, the notion of constructor terms is generalized to the higher-order setting.

Quasi-reductivity Let $\mathcal{L} = (\Sigma, \mathcal{R})$ be some LCSTRS. The *semi-constructor terms* over $\mathcal{L}$, notation $\mathcal{SCT}_{\mathcal{L}}$, are defined by **(i)**. $\mathcal{V} \subseteq \mathcal{SCT}_{\mathcal{L}}$, **(ii)**. if $\mathsf{f} \in \Sigma$ with $\mathsf{f} :: \sigma_1 \to \ldots \to \sigma_m \to \iota$, $\iota \in \mathcal{S}$ and $s_1 :: \sigma_1, \ldots, s_n :: \sigma_n \in \mathcal{SCT}_{\mathcal{L}}$ with $n \leq m$, then $\mathsf{f}\ s_1 \cdots s_n \in \mathcal{SCT}_{\mathcal{L}}$ if $n < ar(\mathsf{f})$.

Semi-constructor terms are always normal forms. Furthermore, as $ar(\mathsf{f}) = \infty$ for $\mathsf{f} \in \mathcal{C}$, the constructor terms $T(\mathcal{C}, \mathcal{V})$ are contained in $\mathcal{SCT}_{\mathcal{L}}$. *Ground* semi-constructor terms $\mathcal{SCT}_{\mathcal{L}}^{\emptyset}$ are the terms built without (i). A *ground semi-constructor substitution* (*gsc substitution*) is a substitution such that $im(\gamma) \subseteq \mathcal{SCT}_{\mathcal{L}}^{\emptyset}$.

$\mathcal{L}$ is quasi-reductive if for every $t \in T(\Sigma, \emptyset)$ we have $t \in \mathcal{SCT}_{\mathcal{L}}^{\emptyset}$ or t reduces with $\to_{\mathcal{R}}$. Put differently, the only ground normal forms are semi-constructor terms. Weak normalization and quasi-reductivity together ensure that every ground term reduces to a semi-constructor term. Note that, if $s_1, \ldots, s_n$ are ground normal forms and $\mathsf{f} \in \Sigma$, then $\mathsf{f}\ s_1 \cdots s_n$ is a ground normal form if and only if $n < ar(\mathsf{f})$.

Equations and inductive theorems An *equation* is a triple $s \approx t\ [\varphi]$ with $typeof(s) = typeof(t)$ and φ a constraint, such that all variables in $Var(s) \cup Var(t) \cup Var(\varphi)$ have an inhabited type. If φ equals `true`, we will simply write the equation as $s \approx t$. A substitution γ respects $s \approx t\ [\varphi]$ if γ respects φ. An equation $s \approx t\ [\varphi]$ is an *inductive theorem* (aka *ground convertible*) if $s\gamma \leftrightarrow_{\mathcal{R}}^* t\gamma$ for every ground substitution γ that respects φ. Here $\leftrightarrow_{\mathcal{R}} = \to_{\mathcal{R}} \cup \leftarrow_{\mathcal{R}}$, and $\leftrightarrow_{\mathcal{R}}^*$ is its transitive, reflexive closure.

Example 3 The LCSTRS from Example 2 admits an equation $\mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a$. Since it has constraint `true`, any substitution respects it. In subsection 3.2 we will prove that this equation is an inductive theorem, meaning that $(\mathsf{recdown}\ f\ n\ i\ a)\gamma \leftrightarrow_{\mathcal{R}}^* (\mathsf{tailup}\ f\ n\ i\ a)\gamma$ for any ground substitution γ.

3 Rewriting Induction

RI was introduced [10] as a deduction system for proving inductive theorems, using unconstrained first-order term rewriting systems. Since then, many variations on the system have appeared (e.g., [1, 2, 3, 5, 6, 9]), including a version for LCSTRSs. All are based on well-founded induction, using some well-founded relation $\succ$. Some [10, 5, 6, 9] use a fixed strategy to construct a terminating rewrite system $A \supseteq \mathcal{R}$ and then choose $\succ = \to_A^+$. However, as explained in the introduction, this approach leads to heavy termination requirements, because these strategies include all induction hypothesis into A.

To improve on this, some work has been done [1, 2, 3] employing a well-founded relation $\succ$ that satisfies certain requirements (like monotonicity and stability, but also ground totality). This relation may either be fixed beforehand (e.g., the lexicographic path ordering), or constructed during or after the

proof, as the proof process essentially accumulates termination requirements. The RI system is designed to keep termination requirements as mild as possible, for example by allowing reduction steps with an induction hypothesis to be oriented using a second relation $\succeq$ rather than the default $\succ$. However, this approach also imposes more bureaucracy, since derivation rules rely on several steps being done at once – for example, by reasoning *modulo* the set of induction hypotheses. This makes it quite hard to use especially when the relation $\succ$ is not fixed beforehand but rather constructed on the fly.

Here, we aim to combine the best of both worlds. We try to reduce termination requirements using a pair $(\succ,\succeq)$, which may either be fixed in advance, or constructed as part of the proof process. Importantly, we do not impose the ground totality requirement (which would be extremely restrictive in higher-order rewriting!), and thus allow for $\succ$ to for instance be a relation $(\to_A \cup \rhd)^+$, or a construction based on dependency pairs. We avoid the bureaucracy of combining steps by introducing the notion of an equation *context*, which keeps track of an extra pair of terms to be used for ordering requirements.

3.1 Equation contexts and proof states

RI is a deduction system on proof states, which are pairs of the shape $(\mathcal{E},\mathcal{H})$. Intuitively (and following the existing literature), $\mathcal{E}$ is a set of equations, describing all proof goals, and $\mathcal{H}$ is the set of induction hypotheses that have been assumed. At the start $\mathcal{E}$ consists of all equations that we want to prove to be inductive theorems, and $\mathcal{H}=\emptyset$. With a deduction rule we may transform a proof state $(\mathcal{E},\mathcal{H})$ into another proof state $(\mathcal{E}',\mathcal{H}')$. This is denoted as $(\mathcal{E},\mathcal{H}) \vdash (\mathcal{E}',\mathcal{H}')$. We write $\vdash^*$ for the reflexive, transitive closure of $\vdash$. Correctness of RI is guaranteed by the following principle: "If $(\mathcal{E},\mathcal{H}) \vdash^* (\emptyset,\mathcal{H})$ for some set $\mathcal{H}$, then every equation in $\mathcal{E}$ is an inductive theorem" [6, 9]. Intuitively, this reads as: if we can remove every proof obligation (making $\mathcal{E}$ empty) then every equation in $\mathcal{E}$ is an inductive theorem.

In Bounded RI, we will deviate from this setting in one respect: instead of letting $\mathcal{E}$ be a set of equations, we will use a set of *equation contexts*.

3.2 Bounded Rewriting Induction

We will now introduce *bounded rewriting induction*, considering proof states containing only *bounded equation contexts*. For this, we assume a bounding pair:

Definition 1 (Bounding Pair) A *bounding pair* for an LCSTRS $\mathcal{L}=(\Sigma,\mathcal{R})$ is a pair $(\succ,\succeq)$ with $\succ$ a well-founded partial ordering on $T(\Sigma,\emptyset)$ (that is, $\succ$ is a transitive, anti-symmetric, irreflexive and well-founded relation) and $\succeq$ a quasi-order on $T(\Sigma,\emptyset)$ (that is, $\succeq$ is a transitive and reflexive relation) such that $\succ \subseteq \succeq$, $\succ \cdot \succeq \subseteq \succ$, $\succeq \cdot \succ \subseteq \succ$ and such that $s \succeq t$ whenever $s \to_{\mathcal{R}} t$ or $s \rhd t$.

A bounding pair is extended to non-ground terms with constraint: we define $s \succ t \ [\psi]$ iff $s\gamma \succ t\gamma$ for all ground substitutions γ that respect ψ. ($s \succeq t \ [\psi]$ is defined similarly)

If $\mathcal{L}$ is terminating we can choose $\succ = (\to_{\mathcal{R}} \cup \rhd)^+$ and $\succeq$ the reflexive closure of $\succ$. But there are other ways to choose a bounding pair, for example monotonic algebras or recursive path orderings.

Definition 2 (Equation context) Let $\bullet$ be a fresh symbol. We define $\bullet \succ s$ and $\bullet \succeq s$ for all $s \in T(\Sigma,\mathcal{V})$, and also $\bullet \succeq \bullet$. An *equation context* $(\varsigma \ ; \ s \approx t \ ; \ \tau) \ [\psi]$ is a tuple of two elements $\varsigma,\tau \in T(\Sigma,\mathcal{V}) \cup \{\bullet\}$, two terms s,t and a constraint ψ. We write $(\varsigma \ ; \ s \simeq t \ ; \ \tau) \ [\psi]$ (so with $\simeq$ instead of $\approx$) to denote either an equation context $(\varsigma \ ; \ s \approx t \ ; \ \tau) \ [\psi]$ or an equation context $(\tau \ ; \ t \approx s \ ; \ \varsigma) \ [\psi]$.

A *bounded equation context* is an equation context such that both $\varsigma \succeq s \ [\psi]$ and $\tau \succeq t \ [\psi]$. A substitution γ *respects* an equation context $(\varsigma \ ; \ s \approx t \ ; \ \tau) \ [\psi]$ if γ respects ψ.

An equation context couples an equation with a bound on the induction: we implicitly work with the induction hypothesis "all ground instances of an equation in $\mathcal{H}$ that are strictly smaller than the current instance of $\varsigma \approx \tau\ [\psi]$ are convertible". For example, in an equivalence proof of two implementations of the factorial function, we may encounter induction hypothesis $\mathsf{fact}_1\ n \approx \mathsf{fact}_2\ n\ [n \geq 0]$, and equation context $(\mathsf{fact}_1\ n\ ;\ \mathsf{fact}_1\ k \approx \mathsf{fact}_2\ k\ ;\ \mathsf{fact}_2\ n)\ [n > 0 \wedge n = k+1]$. We can apply the instance $\mathsf{fact}_1\ k \approx \mathsf{fact}_2\ k\ [k \geq 0]$ of the induction hypothesis to $\mathsf{fact}_1\ k \approx \mathsf{fact}_2\ k\ [n > 0 \wedge n = k+1]$ because

(**1**). $n > 0 \wedge n = k+1$ implies $k \geq 0$, and (**2**). both $\mathsf{fact}_1\ n \succ \mathsf{fact}_1\ k\ [n > 0 \wedge n = k+1]$ and $\mathsf{fact}_2\ n \succ \mathsf{fact}_2\ k\ [n > 0 \wedge n = k+1]$ hold for an appropriately chosen $\succ$.

Definition 3 (Proof state) $(\mathcal{E}, \mathcal{H})$ is a proof state if $\mathcal{E}$ a set of equation contexts and $\mathcal{H}$ a set of equations.

From Definition 2 we can see that $\bullet$ behaves as an infinity term with respect to $\succ$ and $\succeq$. As expressed by Theorem 1: when using bounded RI to prove a set of equations, we pour them into a set $\mathcal{E}$ of equation contexts using infinite bounds $\varsigma = \tau = \bullet$. This is not a problem, because we always start with the proof state $(\mathcal{E}, \emptyset)$, so there are no induction hypothesis available yet. As soon we add an induction hypothesis to the proof state, the bounds are correctly getting lowered, as dictated by Figure 1.(Induct).

Theorem 1 (Correctness of Bounded RI) *Let $\mathcal{L}$ be a weakly normalizing, quasi-reductive LCSTRS; let $\mathcal{A}$ be a set of equations; and let $\mathcal{E}$ be the set of equation contexts $\{(\bullet\ ;\ s \approx t\ ;\ \bullet)\ [\psi] \mid s \approx t\ [\psi] \in \mathcal{A}\}$. Let $(\succ, \succeq)$ be some bounding pair, such that $(\mathcal{E}, \emptyset) \vdash^* (\emptyset, \mathcal{H})$, for some $\mathcal{H}$ using the derivation rules in Figure 1. Then every equation in $\mathcal{A}$ is an inductive theorem.*

The deduction rules for bounded rewriting induction are provided in Figure 1, and explained in detail below via a running example. This figure uses one particular new notation $\psi \models^{\delta} \varphi$, defined as follows:

Definition 4 ($\models^{\delta}$) Let δ be a substitution and φ, ψ be constraints. We write $\psi \models^{\delta} \varphi$ if $\delta(Var(\varphi)) \subseteq \mathcal{V}al \cup Var(\psi)$, and $\psi \Longrightarrow \varphi\delta$ is a valid constraint.

We will now elaborate on the rules of Figure 1, and illustrate their use through examples. To start, we will use the LCSTRS from Example 2 applied on the equation $\mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a$. Following Theorem 1, we will show that there is a set $\mathcal{H}$ such that

$$(\mathcal{E}_1, \emptyset) \vdash^* (\emptyset, \mathcal{H}) \text{ with } \mathcal{E}_1 := \{(\bullet\ ;\ \mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a\ ;\ \bullet)\ [\mathtt{true}]\}$$

We use the proof process to accumulate requirements on $\succ$ to be used, but precommit to a bounding pair such that $\succeq$ is the reflexive closure of $\succ$. We also assume that $s \succ t$ whenever $s \to_{\mathcal{R}} t$ or $s \rhd t$. To guarantee a well-defined proof system on bounded equation contexts we should demonstrate (which we will not do here) that all deduction rules preserve the following property

$$(\star\star): \quad \begin{array}{l} \text{For every equation context } (\varsigma\ ;\ s \approx t\ ;\ \tau)\ [\psi] \\ \text{either } \varsigma = s \text{ or } \varsigma \succ s\ [\psi], \text{ and also either } \tau = t \text{ or } \tau \succ t\ [\psi]. \end{array}$$

(Induct) This deduction rule starts an induction proof. From a proof-technical point of view two things happen. First, and most importantly for the proof progress, the current equation is added to $\mathcal{H}$, making it available for later application of (Hypothesis) or ($\mathcal{H}$-Delete). Second, the bounding terms ς, τ are replaced by s, t. This ensures that, when an induction hypothesis $s \approx t\ [\psi]$ is applied, it is only on equations that are strictly smaller than $s \approx t\ [\psi]$.

In our running example, we use (Induct) to obtain $(\mathcal{E}_1, \emptyset) \vdash (\mathcal{E}_2, \mathcal{H}_2)$ where

$$\mathcal{E}_2 = \{(\varsigma_2\ ;\ \mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a\ ;\ \tau_2)\ [\mathtt{true}]\} \quad \mathcal{H}_2 = \{\mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a\}$$

We recall $\varsigma_2 = \mathsf{recdown}\ f\ n\ i\ a$ and $\tau_2 = \mathsf{tailup}\ f\ n\ i\ a$ for later usage in the RI process.

Figure 1: Derivation rules for bounded rewriting induction, given a bounding pair $(\succ, \succeq)$.

(Simplify)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \,;\, C[\ell\delta] \simeq t \,;\, \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E} \cup \{(\varsigma \,;\, C[r\delta] \approx t \,;\, \tau)\ [\psi]\}, \mathcal{H})} \quad \ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{calc} \text{ and } \psi \models^{\delta} \varphi$$

(Case)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \,;\, s \approx t \,;\, \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E} \cup \{(\varsigma\delta \,;\, s\delta \approx t\delta \,;\, \tau\delta)\ [\psi\delta \wedge \varphi] \mid (\delta, \varphi) \in \mathcal{C}\}, \mathcal{H})} \quad \begin{array}{l}\mathcal{C} \text{ a cover set of } s \approx t\ [\psi] \\ \text{(see Definition 5)}\end{array}$$

(Delete)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \,;\, s \approx t \,;\, \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E}, \mathcal{H})} \quad \psi \text{ unsatisfiable, or } s = t$$

(Semi-constructor)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \,;\, f\ s_1 \cdots s_n \approx f\ t_1 \cdots t_n \,;\, \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E} \cup \{(\varsigma \,;\, s_i \approx t_i \,;\, \tau)\ [\psi] \mid 1 \leq i \leq n\}, \mathcal{H})} \quad n > 0 \text{ and } (f \in \mathcal{V} \text{ or } n < ar(f))$$

(Induct)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \,;\, s \approx t \,;\, \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E} \cup \{(s \,;\, s \approx t \,;\, t)\ [\psi]\}, \mathcal{H} \cup \{s \approx t\ [\psi]\})}$$

(Hypothesis)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \,;\, C[\ell\delta] \simeq t \,;\, \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E} \cup \{(\varsigma \,;\, C[r\delta] \approx t \,;\, \tau)\ [\psi]\}, \mathcal{H})} \quad \begin{array}{l}\ell \simeq r\ [\varphi] \in \mathcal{H} \text{ and } \psi \models^{\delta} \varphi \text{ and} \\ \varsigma \succ \ell\delta\ [\psi] \text{ and } \varsigma \succ r\delta\ [\psi] \text{ and } \varsigma \succeq C[r\delta]\ [\psi]\end{array}$$

($\mathcal{H}$-Delete)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \,;\, C[\ell\delta] \simeq C[r\delta] \,;\, \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E}, \mathcal{H})} \quad \begin{array}{l}\ell \simeq r\ [\varphi] \in \mathcal{H} \text{ and } \psi \models^{\delta} \varphi \text{ and} \\ \varsigma \succ \ell\delta\ [\psi] \text{ or } \tau \succ r\delta\ [\psi]\end{array}$$

(Generalize)/(Alter)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \,;\, s \approx t \,;\, \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E} \cup \{(\varsigma' \,;\, s' \approx t' \,;\, \tau')\ [\varphi]\}, \mathcal{H})} \quad \begin{array}{l}(\varsigma' \,;\, s' \approx t' \,;\, \tau')\ [\varphi] \text{ generalizes/alters } (\varsigma \,;\, s \approx t \,;\, \tau)\ [\psi] \\ \text{(see Definition 6), and } \varsigma' \succeq s'\ [\varphi] \text{ and } \tau' \succeq t'\ [\varphi]\end{array}$$

(Postulate)

$$\frac{(\mathcal{E}, \mathcal{H})}{(\mathcal{E} \cup \{(\bullet \,;\, s \approx t \,;\, \bullet)\ [\psi]\}, \mathcal{H})}$$

(Case) Comparing $\mathcal{E}_2$ to $\mathcal{R}$, which of the rules should we apply? As we will see in (Simplify), this requires information about how the variables i,n in the equation are instantiated, since we have to distinguish between the cases $i < n$ and $i \geq n$. This is where (Case) can help us, splitting an equation into multiple cases. Of course, we have to make sure that the cases together cover the original equation.

Definition 5 (Cover set) A cover set of $s \approx t\ [\psi]$ is a set $\mathcal{C}$ of pairs (δ, φ), with δ a substitution and φ a constraint, such that for every gsc substitution γ respecting $s \approx t\ [\psi]$, there exists $(\delta, \varphi) \in \mathcal{C}$ such that $s\gamma \approx t\gamma\ [\psi\gamma]$ is an instance of $s\delta \approx t\delta\ [\psi\delta \wedge \varphi]$. (That is, there is a substitution σ that respects $\psi\delta \wedge \varphi$ such that $s\delta\sigma = s\gamma$ and $t\delta\sigma = t\gamma$.)

Continuing our example: the only gsc terms of type int are values. Hence, $\mathcal{C} = \{([], i < n),\ ([], i \geq n)\}$ is a cover set of $\mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a$. Using (Case), we obtain $(\mathcal{E}_2, \mathcal{H}_2) \vdash (\mathcal{E}_3, \mathcal{H}_2)$ with $\mathcal{E}_3 =$

$$\big\{(\varsigma_2\ ;\ \mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a\ ;\ \tau_2)\ [i < n],\ (\varsigma_2\ ;\ \mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a\ ;\ \tau_2)\ [i \geq n]\big\}$$

The bounding terms ς_2, τ_2 are unchanged because the substitutions in the cover set are both empty.

(Simplify) With (Simplify) we use a rule $\ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{calc}$ to rewrite an equation $C[\ell\delta] \simeq t\ [\psi]$. The requirement $\psi \models^{\delta} \varphi$ makes sure that the δ-instance of $\ell \to r\ [\varphi]$ is actually applicable. The bounding terms are not affected by the reduction.

Continuing our example, the first equation in $\mathcal{E}_3$ has constraint $i < n$, so we apply (Simplify) on both sides of this equation, using **(R1)** and **(R3)**. For the second equation, we also apply (Simplify) to both sides, using **(R2)** and **(R4)**. We obtain $(\mathcal{E}_3, \mathcal{H}_2) \vdash^* (\mathcal{E}_4, \mathcal{H}_2)$ with

$$\mathcal{E}_4 = \left\{ \begin{array}{cc} (\varsigma_2\ ;\ a \approx a\ ;\ \tau_2) & [i < n] \\ (\varsigma_2\ ;\ f\ i\ (\mathsf{recdown}\ f\ n\ (i-1)\ a) \approx \mathsf{tailup}\ f\ (n+1)\ i\ (f\ n\ a)\ ;\ \tau_2) & [i \geq n] \end{array} \right\}$$

(Delete) This deduction rule allows us to remove an equation that has an unsatisfiable constraint, or whose two sides are syntactically equal. In our example, we use (Delete) and obtain $(\mathcal{E}_4, \mathcal{H}_2) \vdash (\mathcal{E}_5, \mathcal{H}_2)$

$$\mathcal{E}_5 = \{(\varsigma_2\ ;\ f\ i\ (\mathsf{recdown}\ f\ n\ (i-1)\ a) \approx \mathsf{tailup}\ f\ (n+1)\ i\ (f\ n\ a)\ ;\ \tau_2)\ [i \geq n]\}$$

(Alter) If is often useful to rewrite an equation (context) to another that might be syntactically different, but has the same ground instances (or at least: the same ground semi-constructor instances). Indeed, this may even be necessary, for instance to support the application of a rewrite rule through (Simplify).

Definition 6 We say that an equation context $(\varsigma'\ ;\ s' \approx t'\ ;\ \tau')\ [\varphi]$ *generalizes* $(\varsigma\ ;\ s \approx t\ ;\ \tau)\ [\psi]$ if for every gsc substitution γ that respects $(\varsigma\ ;\ s \approx t\ ;\ \tau)\ [\psi]$ there is a substitution δ that respects $(\varsigma'\ ;\ s' \approx t'\ ;\ \tau')\ [\varphi]$ such that $s\gamma = s'\delta$ and $t\gamma = t'\delta$, and $\varsigma\gamma \succeq \varsigma'\delta$ and $\tau\gamma \succeq \tau'\delta$. It *alters* $(\varsigma\ ;\ s \approx t\ ;\ \tau)\ [\psi]$ if both $(\varsigma'\ ;\ s' \approx t'\ ;\ \tau')\ [\varphi]$ generalizes $(\varsigma\ ;\ s \approx t\ ;\ \tau)\ [\psi]$, and $(\varsigma\ ;\ s \approx t\ ;\ \tau)\ [\psi]$ generalizes $(\varsigma'\ ;\ s' \approx t'\ ;\ \tau')\ [\varphi]$.

There are many ways to use the (Alter) rule, but following the discussion in [9], we will particularly apply it in two ways: **(i)**. Replacing a constraint by an equi-satisfiable one **(ii)**. Replacing variables by equivalent variables or values.

Continuing our example, we apply (Alter) with case (i) to obtain $(\mathcal{E}_5, \mathcal{H}_2) \vdash (\mathcal{E}_6, \mathcal{H}_2)$, with

$$\mathcal{E}_6 = \big\{(\varsigma_2\ ;\ f\ i\ (\mathsf{recdown}\ f\ n\ (i-1)\ a) \approx \mathsf{tailup}\ f\ (n+1)\ i\ (f\ n\ a)\ ;\ \tau_2)\ [i' = i-1 \wedge n' = n+1 \wedge i \geq n]\big\}$$

To allow this rule to be applied, we must have $\varsigma_2 \succeq f\ i\ (\mathsf{recdown}\ f\ n\ (i-1)\ a)\ [\varphi]$ and $\tau_2 \succeq \mathsf{tailup}\ f\ (n+1)\ i\ (f\ n\ a)\ [\varphi]$ where φ is the constraint $i' = i-1 \wedge n' = n+1 \wedge i \geq n$. But this follows immediately from $(\star\star)$: if $\varsigma \succ s\ [i \geq n]$ then also $\varsigma \succ s\ [i' = i-1 \wedge n' = n+1 \wedge i \geq n]$, and similar for $\tau \succ t\ [i \geq n]$.

Our previous (Alter) step allows us to continue on the example by two successive (Simplify) steps, using calculation rules $i-1 \to i'\ [i' = i-1]$ and $n+1 \to n'\ [n' = n+1]$, to obtain $(\mathcal{E}_7, \mathcal{H}_2)$, with

$$\mathcal{E}_7 = \left\{(\varsigma_2\ ;\ f\ i\ (\mathsf{recdown}\ f\ n\ i'\ a) \approx \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a)\ ;\ \tau_2)\ [i' = i-1 \wedge n' = n+1 \wedge i \geq n]\right\}$$

(Hypothesis) Similar to (Simplify), we can use an induction hypothesis to reduce either side of an equation. Here, finally, the bounding terms ς, τ come into play, as we need to make sure that we have a decrease of some kind, to apply induction.

We apply (Hypothesis) on the lhs of $\mathcal{E}_7$ with the induction hypothesis from $\mathcal{H}_2$ in the direction $\mathsf{recdown}\ f\ n\ i\ a \ \to\ \mathsf{tailup}\ f\ n\ i\ a$, with substitution $[i := i']$. We obtain $(\mathcal{E}_7, \mathcal{H}_2) \vdash (\mathcal{E}_8, \mathcal{H}_2)$ with

$$\mathcal{E}_8 = \left\{(\varsigma_2\ ;\ f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) \approx \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a)\ ;\ \tau_2) \quad [i' = i-1 \wedge n' = n+1 \wedge i \geq n]\right\}$$

To be allowed to apply this deduction rule, we must show that the following $\succ$ requirements are satisfied:

$$\begin{array}{rlll} & \mathsf{recdown}\ f\ n\ i\ a & \succ & \mathsf{recdown}\ f\ n\ i'\ a & [i' = i-1 \wedge n' = n+1 \wedge i \geq n] \\ & \mathsf{recdown}\ f\ n\ i\ a & \succ & \mathsf{tailup}\ f\ n\ i'\ a & [i' = i-1 \wedge n' = n+1 \wedge i \geq n] \\ \textbf{(REQ1)} & \mathsf{recdown}\ f\ n\ i\ a & \succeq & f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) & [i' = i-1 \wedge n' = n+1 \wedge i \geq n] \end{array}$$

The first of these is satisfied by property $(\star\star)$. The second is an immediate consequence of the third, since $\mathsf{f}\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) \rhd \mathsf{tailup}\ f\ n\ i'\ a$ and we have committed to let $\rhd$ be included in $\succ$. For the third, we remember that **(REQ1)** still needs to be satisfied. Since we have set $\succeq$ as the reflexive closure of $\succ$, this property is only satisfied if $\mathsf{recdown}\ f\ n\ i\ a \succ f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a)\ [i' = i-1 \wedge n' = n+1 \wedge i \geq n]$.

Let $\varsigma_9 = f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a)$ and $\tau_9 = \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a)$. We apply (Induct) to $(\mathcal{E}_8, \mathcal{H}_2)$ and obtain

$$\begin{aligned} \mathcal{E}_9 &= \left\{(\varsigma_9\ ;\ f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) \approx \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a)\ ;\ \tau_9)\ [i' = i-1 \wedge n' = n+1 \wedge i \geq n]\right\} \\ \mathcal{H}_9 &= \mathcal{H}_2 \cup \{f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) \approx \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a)\ [i' = i-1 \wedge n' = n+1 \wedge i \geq n]\} \end{aligned}$$

Next, we use (Case) once more, splitting up the constraint $\mathcal{E}_9$ into $i = n$ and $i > n$, giving $(\mathcal{E}_{10}, \mathcal{H}_9)$:

$$\mathcal{E}_{10} = \left\{\begin{array}{l} (\varsigma_9\ ;\ f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) \approx \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a)\ ;\ \tau_9)\ [i' = i-1 \wedge n' = n+1 \wedge i = n] \\ (\varsigma_9\ ;\ f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) \approx \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a)\ ;\ \tau_9)\ [i' = i-1 \wedge n' = n+1 \wedge i > n] \end{array}\right\}$$

Observing that $i' = i-1 \wedge n' = n+1 \wedge i = n$ implies both $n > i'$ and $n' > i$, and that $i' = i-1 \wedge n' = n+1 \wedge i > n$ implies both $n \leq i'$ and $n' \leq i$, we use (Simplify) on both sides of the first equation with **(R3)** and on both sides of the second equation with **(R4)** respectively, to deduce $(\mathcal{E}_{10}, \mathcal{H}_9) \vdash^* (\mathcal{E}_{11}, \mathcal{H}_9)$:

$$\mathcal{E}_{11} = \left\{\begin{array}{l} (\varsigma_9\ ;\ f\ i\ a \approx f\ n\ a\ ;\ \tau_9) \qquad\qquad [i' = i-1 \wedge n' = n+1 \wedge i = n] \\ (\varsigma_9\ ;\ f\ i\ (\mathsf{tailup}\ f\ (n+1)\ i'\ (f\ n\ a)) \approx \mathsf{tailup}\ f\ (n'+1)\ i\ (f\ n'\ (f\ n\ a))\ ;\ \tau_9) \\ \qquad\qquad [i' = i-1 \wedge n' = n+1 \wedge i > n] \end{array}\right\}$$

The first equation above does not satisfy the requirements for (Delete), even though the $i = n$ part of the constraint makes it look very delete-worthy. With (Alter) (case (ii)), we replace the first equation context by $(\varsigma_9\ ;\ f\ n\ a \approx f\ n\ a\ ;\ \tau_9)\ [i' = i-1 \wedge n' = n+1 \wedge i = n]$, which may immediately be deleted. We also use (Alter) (now case (i)) on the second equation, succeeded by (Simplify). This yields $(\mathcal{E}_{12}, \mathcal{H}_9)$ with

$$\mathcal{E}_{12} = \left\{\begin{array}{l} (\varsigma_9\ ;\ f\ i\ (\mathsf{tailup}\ f\ n'\ i'\ (f\ n\ a)) \approx \mathsf{tailup}\ f\ n''\ i\ (f\ n'\ (f\ n\ a))\ ;\ \tau_9) \\ \qquad\qquad [i' = i-1 \wedge n' = n+1 \wedge n'' = n'+1 \wedge i > n] \end{array}\right\}$$

($\mathcal{H}$-Delete). With this deduction rule we may rewrite an equation with an instance of equation in $\mathcal{H}$.

In our example, consider the second equation in $\mathcal{H}_9$. Let $\delta = [n := n', n' := n'', a := f\ n\ a]$. Using ($\mathcal{H}$-Delete), we can deduce $(\mathcal{E}_{12}, \mathcal{H}_9) \vdash (\emptyset, \mathcal{H}_9)$ if one of the following ordering requirements are satisfied:

$$\begin{array}{rll} \varsigma_9 = & f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) \succ f\ i\ (\mathsf{tailup}\ f\ n'\ i'\ (f\ n\ a)) & [i' = i-1 \wedge n' = n+1 \wedge i \geq n] \\ \tau_9 = & \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a) \succ \mathsf{tailup}\ f\ n''\ i\ (f\ n'\ (f\ n\ a)) & [i' = i-1 \wedge n' = n+1 \wedge i \geq n] \end{array}$$

We obtained $(\mathcal{E}_1, \emptyset) \vdash^* (\emptyset, \mathcal{H}_9)$. By Theorem 1 $\mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a$ is an inductive theorem – provided we have a suitable bounding pair that satisfies **(REQ1)**. But this is easily achieved: let $\succ$ equal $(\to_{\mathcal{R}\cup\mathcal{Q}} \cup \rhd)^+$ where $\mathcal{Q} = \{\mathsf{recdown}\ f\ n\ i\ a \to f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a)\ [i' = i-1 \wedge n' = n+1 \wedge i \geq n]\}$.

It is easy to see that this is indeed a bounding pair if $\to_{\mathcal{R}\cup\mathcal{Q}}$ is terminating. Termination can for instance be proved using static dependency pairs [7].

Remark 1 The choice to let $\succ$ be a relation $(\to_{\mathcal{R}\cup\mathcal{Q}} \cup \rhd)^+$ is quite natural: in traditional definitions of rewriting induction [10, 6, 9] this is the only choice for $(\succ, \succeq)$, with $\mathcal{Q}$ always being a directed version of the last $\mathcal{H}$ (so in the case of this example, $\mathcal{H}_9$). However, while such a choice is natural in strategies for rewriting induction, we leave it open in the definition to allow for alternative orderings.

4 Closing remarks

Two deduction rules we did not demonstrate are (Generalize) and (Postulate). Although (Generalize) appears to be very similar to (Alter) (in fact, every step that can be done by (Alter) can also be done by (Generalize)), they are used quite differently: (Alter) is designed to set up an equation for the use of simplification or deletion, while (Generalize) and (Postulate) are a way to perform of lemma generation.

Lemma generation is often needed in practice to obtain a successful RI proof. This was not visible in the running example in subsection 3.2, where we could for example continue on the equation in $\mathcal{E}_7$ by applying the hypothesis in $\mathcal{H}_2$, which was automatically generated by (Induct), in an (Hypothesis)-step. However, in many practical situations the hypothesis generated by (Induct) is not applicable, and we first have to use (Generalize) to introduce a more general equation, suitable to save as hypothesis for later usage in (Hypothesis) or ($\mathcal{H}$-Delete). How to find such generalizations automatically is a separate topic, and beyond the scope of this paper. The idea of (Postulate) is similar to (Generalize), but rather than replacing the original equation by a generalized equation, we add the generalized equation as a new equation and apply (Induct) on this new equation to obtain the required induction hypothesis.

We have not demonstrated (Semi-constructor) either. With this deduction rule we can split up an equation that contains a constructor or partially applied function symbols. For example, we can split up $\mathsf{foldl}\ \mathsf{g}\ (\mathsf{h}\ 0\ x) \approx \mathsf{foldl}\ \mathsf{h}\ (\mathsf{g}\ 0\ x)$ into two equations: $\mathsf{g} \approx \mathsf{h}$ and $\mathsf{h}\ 0\ x \approx \mathsf{g}\ 0\ x$.

Implementation and work in progress A basic version of Bounded RI for LCSTRSs has been implemented in Cora (available on `https://github.com/hezzel/cora`).

Considering work in progress, we are currently working on RI as a method for proving ground confluence in LCSTRSs. This builds on the work of [4], where the authors showed that this is possible for first-order unconstrained rewriting. Ground confluence is of relevance for completeness because for ground confluent LCSTRSs we can extend RI with a new deduction rule, in order to disprove equations to be inductive theorems. For first-order constrained rewriting this has been shown in [6], and we want to generalize this result to RI for LCSTRSs.

References

[1] T. Aoto (2006): *Dealing with Non-orientable Equations in Rewriting Induction*. In: *Proc. RTA 06*, *Lecture Notes in Computer Science* 4098, pp. 242–256, doi:10.1007/11805618_18. Available at `https://doi.org/10.1007/11805618_18`.

[2] T. Aoto (2008): *Designing a rewriting induction prover with an increased capability of non-orientable theorems*. In: *Proc. SCSS 08*.

[3] T. Aoto (2008): *Soundness of Rewriting Induction Based on an Abstract Principle*. *Inf. Media Technol.* 3(2), pp. 225–235, doi:10.11185/IMT.3.225. Available at `https://doi.org/10.11185/imt.3.225`.

[4] T. Aoto & Y. Toyama (2016): *Ground Confluence Prover based on Rewriting Induction*. In: *Proc. FSCD 16*, *LIPIcs* 52, pp. 33:1–33:12, doi:10.4230/LIPICS.FSCD.2016.33. Available at `https://doi.org/10.4230/LIPIcs.FSCD.2016.33`.

[5] S. Falke & D. Kapur (2012): *Rewriting Induction + Linear Arithmetic = Decision Procedure*. In: *Proc. IJCAR 12*, *LNAI* 7364, pp. 241–255, doi:10.1007/978-3-642-31365-3_20.

[6] C. Fuhs, C. Kop & N. Nishida (2017): *Verifying Procedural Programs via Constrained Rewriting Induction*. *ACM Transactions On Computational Logic (TOCL)* 18(2), pp. 14:1–14:50, doi:10.1145/3060143.

[7] L. Guo, K. Hagens, C Kop & D. Vale (2024): *Higher-Order Constrained Dependency Pairs for (Universal) Computability*. In: *Proc. MFCS 24*, doi:10.48550/arXiv.2406.19379.

[8] L. Guo & C. Kop (2024): *Higher-Order LCTRSs and Their Termination*. In: *Proc. ESOP 24*, *LNCS* 14577, pp. 331–357, doi:10.1007/978-3-031-57267-8_13.

[9] K. Hagens & C. Kop (2024): *Rewriting Induction for Higher-Order Constrained Term Rewriting Systems*. In: *Proc. LOPSTR 24*, 14919, pp. 202–219, doi:10.1007/978-3-031-71294-4_12. Available at `https://doi.org/10.1007/978-3-031-71294-4_12`.

[10] U.S. Reddy (1990): *Term Rewriting Induction*. In: *Proc. CADE '90*, *LNCS* 449, pp. 162–177, doi:10.1007/3-540-52885-7_86.