Higher-order inductive theorems via recursor templates

K. Hagens[1] and C. Kop[2]

[1] Radboud University, Nijmegen, Netherlands
`kasper.hagens@ru.nl`

[2] Radboud University, Nijmegen, Netherlands
`c.kop@cs.ru.nl`

Abstract

Rewriting Induction (RI) is a formal system in term rewriting for proving inductive theorems. Recently, RI has been extended to higher-order Logically Constrained Term Rewriting Systems (LCSTRSs), which makes it an interesting tool for program verification with inductive theorems as an interpretation for program equivalence. A major challenge when proving inductive theorems with RI is the generation of suitable induction hypothesis, preferably automatically. Two existing heuristics often fail. Here, we consider another approach: rather than inventing new heuristics for proving individual cases, we consider classes of equivalences. This is achieved by introducing templates, describing specific tail and non-tail recursive programs. Whenever each of the two programs fit into such a template we can generate an equation which is guaranteed to be an inductive theorem.

1 Introduction

Rewriting Induction (RI) is a method for inductive theorem proving. Recently, it was extended to higher-order Logically Constrained Term Rewriting Systems (LCSTRSs) [5], making it an interesting tool for program verification with inductive theorems as interpretation for program equivalence. The RI proof system is based on well-founded induction, and proving an equation often requires to introduce another equation, to be used as induction hypothesis. Finding such an induction hypothesis is known to be a non-trivial problem, and the two existing generalization methods for RI do not always succeed.

Inspired by [1], we consider another approach: rather than inventing new heuristics for proving the equivalence of individual program pairs, we consider classes of equivalences. We introduce tail and non-tail recursors, specifically aimed at describing simple bounded loop constructions, governed by some binary integer operator. We then introduce templates for describing specific tail and non-tail recursive programs. Whenever each of the two programs fit into a template we can generate an equation which is guaranteed to be an inductive theorem.

Induction proofs with RI Figure 1 shows four implementations of the factorial function: Tail recursive Upward (TU), Tail recursive Downward (TD), Recursive Upward (RU) and Recursive Downward (RD). Figure 2 shows their LCSTRS representation. Provided $x \geq 1$, they all compute $x \mapsto \prod_{i=1}^{x} i$. We aim to prove all $\binom{4}{2} = 6$ program-pairs being equivalent. In the setting of LCSTRSs the equivalence of, for example, $\mathsf{factTU}\ x$ and $\mathsf{factRU}\ x$ for $x \geq 1$ is expressed by the equation $\mathsf{factTU}\ x \approx \mathsf{factRU}\ x\ [x \geq 1]$.

With RI we then prove that this equation is an *inductive theorem*, meaning that for every ground substitution γ that satisfies $[\![(x \geq 1)\gamma]\!] = \top$ we have $(\mathsf{factTU}\ x)\gamma \leftrightarrow_{\mathcal{R}}^{*} (\mathsf{factRU}\ x)\gamma$. Here, $\leftrightarrow_{\mathcal{R}}^{*}$ is the transitive, reflexive closure of $\rightarrow_{\mathcal{R}} \cup \leftarrow_{\mathcal{R}}$, with $\rightarrow_{\mathcal{R}}$ the rewrite relation generated by $\mathcal{R}$ (and $\mathcal{R}$ the set of all rules involved in the definition of factTU and factRU).

A pleasant property of constrained rewriting is that it incorporates primitive data structures (such as the integers) non-inductively. This in turn is beneficial when it comes to inductive

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

(a) Tail recursive Upward

```
int factTD(int x){
    int a = 1;
    while (x>0){
        a = a*x; x = x-1;}
    return a; }
```

(b) Tail recursive Downward

```
int factRD(int x){
    if (x > 0)
        return(x*factRD(x-1));
    else
        return 1; }
```

(c) Recursive Downward

```
int factRU(int x) = R(1, x);
int R(int i, int x){
    if (i<x)
        return(i*R(i+1, x));
    else
        return x; }
```

(d) Recursive Upward

Figure 1: Four equivalent implementations of $x \mapsto \prod_{i=1}^{x} i$.

$\mathsf{factTU}\ x \to \mathsf{u}\ x\ 1\ 1$

$\mathsf{u}\ x\ i\ a \to a \qquad [i > x]$

$\mathsf{u}\ x\ i\ a \to \mathsf{u}\ x\ (i+1)\ (i * a) \qquad [i \leq x]$

(a) Tail recursive Upward

$\mathsf{factTD}\ x \to \mathsf{d}\ x\ 1$

$\mathsf{d}\ x\ a \to a \qquad [x \leq 0]$

$\mathsf{d}\ x\ a \to \mathsf{d}\ (x-1)\ (a * x) \qquad [x > 0]$

(b) Tail recursive Downward

$\mathsf{factRD}\ x \to 1 \qquad [x \leq 1]$

$\mathsf{factRD}\ x \to x * (\mathsf{factRD}\ (x-1)) \qquad [x > 1]$

(c) Recursive Downward

$\mathsf{factRU}\ x \to \mathsf{R}\ 1\ x$

$\mathsf{R}\ i\ x \to x \qquad [i > x-1]$

$\mathsf{R}\ i\ x \to i * (\mathsf{R}\ (i+1)\ x) \qquad [i \leq x-1]$

(d) Recursive Upward

Figure 2: The LCTRS representations of the programs in Figure 1.

theorem proving, as it allows us to more directly deal with the program definition itself, instead of getting involved in complicated interactions with underlying recursively defined data structures. As we will see below: when working with integer programs we can use polynomials over $\mathbb{Z}$ to express invariants that we need to generate an induction hypothesis (and we can use computer algebra systems to find such polynomials [4]).

We briefly try to give some intuition for the role of generalization during the generation of induction hypotheses in RI proofs, and why this is easier for some equivalences than for others.

Typically, the two existing generalization methods (InGen [2] and matrix invariants [4]) perform well when comparing a tail recursive implementation with a recursive implementation. The corresponding RI proof is usually generated in two stages, which we will illustrate below for the equivalence $\mathsf{factTU}\ x \approx \mathsf{factRD}\ x$.

Stage 1: Eliminating recursive term. The proof of $\mathsf{factTU}\ x \approx \mathsf{factRD}\ x$ generates an induction hypothesis $\mathsf{u}\ x\ 1\ 1 \approx \mathsf{factRD}\ x$, which is applied later in the proof process to transform the equation $\mathsf{u}\ x\ 3\ 2 \approx x * (\mathsf{factRD}\ x_1)\ [x \geq 2 \wedge x_1 = x - 1]$ into $\mathsf{u}\ x\ 3\ 2 \approx x * (\mathsf{u}\ x\ 1\ 1)\ [x \geq 2 \wedge x_1 = x - 1]$. This stage did not require any generalization because $\mathsf{u}\ x\ 1\ 1 \approx \mathsf{factRD}\ x$ was automatically obtained as a proof goal during the RI process, and RI always allows us to save a proof goal as an induction hypothesis.

Stage 2: Divergence solving. After eliminating the recursive term, the proof starts to diverge into an infinite repeating process, each time producing a new proof obligation that we are not able to remove. We only show the first three:

$$\begin{aligned} \mathsf{u}\ x\ 3\ 2 &\approx x * \mathsf{u}\ x_1\ 2\ 1 \quad [x \geq 2 \wedge x_1 = x - 1] \\ \mathsf{u}\ x\ 4\ 6 &\approx x * \mathsf{u}\ x_1\ 3\ 2 \quad [x \geq 3 \wedge x_1 = x - 1] \\ \mathsf{u}\ x\ 5\ 24 &\approx x * \mathsf{u}\ x_1\ 4\ 6 \quad [x \geq 4 \wedge x_1 = x - 1] \end{aligned}$$

None of these equations can be saved as induction hypothesis to remove the succeeding equation. Fortunately, both generalization methods are able to generate an induction hypothesis $\mathsf{u}\ x\ i\ a \approx x * \mathsf{u}\ x_1\ i_1\ a_1\ [i_1 = i - 1 \wedge x \geq i_1 \wedge x_1 = x - 1 \wedge a = a_1 * i_1]$.

There is, however, no guarantee that comparing a tail recursive with a recursive will always lead to such a procedure. For example, when trying to prove $\mathsf{factTU}\ x \approx \mathsf{factRU}\ x\ [x \geq 1]$ we are not able to eliminate the recursive term and immediately run into the divergence shown in Figure 3a. The repeated unfolding of the recursive call $\mathsf{R}\ i\ x \to i * (\mathsf{R}\ (i+1)\ x)$ makes

$$\begin{aligned} \mathsf{u}\ x\ 2\ 1 &\approx \mathsf{R}\ 1\ x & [x \geq 1] \\ \mathsf{u}\ x\ 3\ 2 &\approx 1 * (\mathsf{R}\ 2\ x) & [x \geq 2] \\ \mathsf{u}\ x\ 4\ 6 &\approx 1 * (2 * (\mathsf{R}\ 3\ x)) & [x \geq 3] \\ \mathsf{u}\ x\ 5\ 24 &\approx 1 * (2 * (3 * (\mathsf{R}\ 4\ x))) & [x \geq 4] \end{aligned}$$

(a) Original divergence

$$\begin{aligned} \mathsf{u}\ x\ 2\ 1 &\approx \mathsf{R}\ 1\ x & [x \geq 1] \\ \mathsf{u}\ x\ 3\ 2 &\approx 1 * (\mathsf{R}\ 2\ x) & [x \geq 2] \\ \mathsf{u}\ x\ 4\ 6 &\approx 2 * (\mathsf{R}\ 3\ x) & [x \geq 3] \\ \mathsf{u}\ x\ 5\ 24 &\approx 6 * (\mathsf{R}\ 4\ x) & [x \geq 4] \end{aligned}$$

(b) Processed divergence

Figure 3: Divergence of $\mathsf{factTU}\ x \approx \mathsf{factRU}\ x\ [x \geq 1]$

it difficult to handle by both generalization methods, as it leads to a divergence where each equation has a different term shape. Once we turn the divergence into the shape shown in Figure 3b we can apply the matrix invariants method to generate an induction hypothesis $\mathsf{u}\ x\ i\ z \approx a * (\mathsf{R}\ j\ x)\ [z = a * j \wedge i = j + 1 \wedge j \leq x]$. InGen is not capable of producing this induction hypothesis because it is not able to find the crucial invariant $z = a * j$.

For $\mathsf{factRU}\ x \approx \mathsf{factTD}\ x\ [x \geq 1]$ the situation is worse. We obtain a divergence

$$\begin{aligned} \mathsf{d}\ i_1\ a_1 &\approx \mathsf{R}\ 1\ x & [i_1 = x - 1 \ \wedge\ a_1 = x \ \wedge\ x \geq 1] \\ \mathsf{d}\ i_2\ a_2 &\approx 1 * (\mathsf{R}\ 2\ x) & [i_2 = x - 2 \ \wedge\ a_2 = x * (x-1) \ \wedge\ x \geq 2] \\ \mathsf{d}\ i_3\ a_3 &\approx 1 * (2 * (\mathsf{R}\ 3\ x)) & [i_3 = x - 3 \ \wedge\ a_3 = x * (x-1) * (x-2) \ \wedge\ x \geq 3] \end{aligned}$$

This time, we cannot find an induction hypothesis of the shape $\mathsf{d}\ i\ a \approx a * (\mathsf{R}\ j\ x)\ [\varphi]$, where φ only contains polynomial arithmetical expressions (we can find an invariant $i + j = x$ but this is not sufficient to obtain an induction hypothesis). We are not able to prove this equation.

For $\mathsf{factRU}\ x \approx \mathsf{factRD}\ x\ [x \geq 1]$ and $\mathsf{factTU}\ x \approx \mathsf{factTD}\ x$ we encounter similar problems: the invariants that we can find are not sufficient to generate induction hypothesis.

Recursor templates In section 2 we will introduce recursor templates for LCSTRSs. This allows us to circumvent the need for executing explicit RI proofs, which as we just motivated can be quite cumbersome due to the need of finding induction hypotheses. The dirty work only needs to be done once, when proving the correctness of our templates and the corresponding inductive theorems. After this, we can check whether a specific example can be matched with a template and automatically generate a corresponding inductive theorem.

We will show that we can prove all 6 inductive theorems from Figure 2 with recursor templates. In addition, we will show we can handle higher-order examples as well.

1.1 Prerequisites

LCSTRSs [3] are a higher-order rewriting formalism with built-in support for integers and booleans (or in fact any arbitrary theory such as bitvectors, floating point numbers or integer arrays) as well as logical constraints to model control flow. This considers *applicative* higher-order term rewriting (without λ abstractions) and *first-order* constraints. We will introduce the minimal necessary prerequisites.

We assume a set of sorts (base types) $\mathcal{S}$; the set $\mathcal{T}$ of types is defined by $\mathcal{T} ::= \mathcal{S} \mid \mathcal{T} \to \mathcal{T}$. Here, $\to$ is right-associative. Assume a subset $\mathcal{S}_{theory} \subseteq \mathcal{S}$ of *theory sorts* (e.g., int and bool), and define the *theory types* by $\mathcal{T}_{theory} ::= \mathcal{S}_{theory} \mid \mathcal{S}_{theory} \to \mathcal{T}_{theory}$. Every $\iota \in \mathcal{S}_{theory}$ corresponds to a non-empty interpretation set $\mathcal{I}_\iota$. Here, we will use theory sorts bool and int, with $\mathcal{I}_{\mathsf{bool}} = \{\top, \bot\}$ and $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$ (the set of all integers).

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 $\mathcal{T}$. The set of terms $T(\Sigma, \mathcal{V})$ over Σ and $\mathcal{V}$ are the well-typed expressions in $\mathbb{T}$, defined by $\mathbb{T} ::= \Sigma \mid \mathcal{V} \mid \mathbb{T}\ \mathbb{T}$. For a term t, let $Var(t)$ be the set of variables in t. A term t is *ground* if $Var(t) = \emptyset$. We assume that Σ is the disjoint union $\Sigma_{theory} \uplus \Sigma_{terms}$, where $\mathit{typeof}(\mathsf{f}) \in \mathcal{T}_{theory}$ for all $\mathsf{f} \in \Sigma_{theory}$.

Each $\mathsf{f} \in \Sigma_{theory}$ has an interpretation $[\![\mathsf{f}]\!] \in \mathcal{I}_{\mathit{typeof}(\mathsf{f})}$. Here, we will fix $\Sigma_{theory} = \{+, -, *\} \cup \{<, \leq, >, \geq, =, \wedge, \vee, \neg\} \cup \{\mathtt{true}, \mathtt{false}\} \cup \{\mathsf{n} \mid n \in \mathbb{Z}\}$, where each of these symbols is typed and interpreted as expected (e.g. $* :: \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ is interpreted as multiplication on $\mathbb{Z}$). We use $[\mathsf{f}]$ for prefix or partially applied notation (e.g., $[+]\ x\ y$ and $x + y$ are the same). Symbols in Σ_{terms} (such as $\mathsf{factRD} :: \mathsf{int} \to \mathsf{int}$) do not have an interpretation since their behavior will be defined through the rewriting system.

Values are theory symbols of base type, i.e. $\mathcal{V}al = \{v \in \Sigma_{theory} \mid \mathit{typeof}(v) \in \mathcal{S}_{theory}\}$, which in our setting are $\mathtt{true}, \mathtt{false}$ and all n. Elements of $T(\Sigma_{theory}, \mathcal{V})$ are *theory terms*. A *constraint* is a theory term $\varphi :: \mathsf{bool}$, such that $\mathit{typeof}(x) \in \mathcal{S}_{theory}$ for all $x \in Var(\varphi)$. For example, we have theory terms $x + \mathsf{3}$, $\mathtt{true}$ and $\mathsf{7} * \mathsf{0}$. The latter two are ground. We have $[\![\mathsf{7} * \mathsf{0}]\!] = 0$. An example of a constraint is $x * y > \mathsf{0}$.

A rewrite rule is an expression $\ell \to r\ [\varphi]$ with $\mathit{typeof}(\ell) = \mathit{typeof}(r)$, $\ell = \mathsf{f}\ \ell_1 \cdots \ell_k$ with $\mathsf{f} \in \Sigma$ and $k \geq 0$, φ a constraint and $Var(r) \subseteq Var(\ell) \cup Var(\varphi)$. If $\varphi = \mathtt{true}$, we write $\ell \to r$. We assume familiarity with contexts and substitutions. A substitution γ respects constraint φ if $\gamma(Var(\varphi)) \subseteq \mathcal{V}al$ and $[\![\varphi\gamma]\!] = \top$. We define $\mathcal{R}_{calc} = \{\mathsf{f}\ x_1 \cdots x_m \to y\ [y = \mathsf{f}\ x_1 \cdots x_m] \mid \mathsf{f} \in \Sigma_{theory} \setminus \mathcal{V}al,\ \mathit{typeof}(\mathsf{f}) = \iota_1 \to \ldots \to \iota_m \to \kappa\}$. The reduction relation $\to_{\mathcal{R}}$ is defined by:

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

For example, we have a reduction $\mathsf{factRD}\ \mathsf{2} \to_{\mathcal{R}} \mathsf{2} * (\mathsf{factRD}\ (\mathsf{2} - \mathsf{1})) \xrightarrow{\mathcal{R}_{calc}} \mathsf{2} * (\mathsf{factRD}\ \mathsf{1}) \to_{\mathcal{R}} \mathsf{2} * \mathsf{1} \xrightarrow{\mathcal{R}_{calc}} \mathsf{2}$. An *equation* is a triple $s \approx t\ [\varphi]$ with $\mathit{typeof}(s) = \mathit{typeof}(t)$ and φ a constraint. A substitution γ respects $s \approx t\ [\varphi]$ if γ respects φ and $Var(s) \cup Var(t) \subseteq \mathit{dom}(\gamma)$. An equation $s \approx t\ [\varphi]$ is an *inductive theorem* if $s\gamma \leftrightarrow^*_{\mathcal{R}} t\gamma$ for every ground substitution γ that respects it. Here $\leftrightarrow_{\mathcal{R}} = \to_{\mathcal{R}} \cup \leftarrow_{\mathcal{R}}$, and $\leftrightarrow^*_{\mathcal{R}}$ is its transitive, reflexive closure.

RI is a deduction system on proof states, which are pairs of the shape $(\mathcal{E}, \mathcal{H})$. Intuitively, $\mathcal{E}$ 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 $\vdash$. If a RI deduction removes every proof goal in $\mathcal{E}$ then $\mathcal{E}$ only contains inductive theorems. This is expressed the following soudness 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".

Template		**Inductive theorem**
$C_{x,y}[i,a] \to a$	$[i > y]$	$C_{x,y}[i,a] \approx \mathsf{tailup}\ \mathsf{f}\ i\ x\ y\ a$
$C_{x,y}[i,a] \to C_{x,y}[i+1, \mathsf{f}\ i\ a]$	$[i \leq y]$	$[x \leq i \leq y]$
$C_{x,y}[i,a] \to a$	$[i < x]$	$C_{x,y}[i,a] \approx \mathsf{taildown}\ \mathsf{f}\ i\ x\ y\ a$
$C_{x,y}[i,a] \to C_{x,y}[i-1, \mathsf{f}\ a\ i]$	$[i \geq x]$	$[x \leq i \leq y]$
$C_{x,y}[i] \to D[i]$	$[i > y]$	$C_{x,y}[i] \approx \mathsf{recup}\ \mathsf{f}\ i\ x\ y\ D[z]$
$C_{x,y}[i] \to \mathsf{f}\ i\ C_{x,y}[i+1]$	$[i \leq y]$	$[x \leq i \leq y \wedge z = y+1]$
$C_{x,y}[i] \to D[i]$	$[i < x]$	$C_{x,y}[i] \approx \mathsf{recdown}\ \mathsf{f}\ i\ x\ y\ D[z]$
$C_{x,y}[i] \to \mathsf{f}\ i\ C_{x,y}[i-1]$	$[i \geq x]$	$[x \leq i \leq y \wedge z = x-1]$

Table 1: Recursor templates and corresponding inductive theorems

2 Recursor templates

We define recursors of type $(\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ as follows

$$\begin{aligned}
\mathsf{tailup}\ f\ i\ x\ y\ a &\to \mathsf{tailup}\ f\ (i+1)\ x\ y\ (f\ i\ a) && [x \leq i \leq y]\\
\mathsf{taildown}\ f\ i\ x\ y\ a &\to \mathsf{taildown}\ f\ (i-1)\ x\ y\ (f\ a\ i) && [x \leq i \leq y]\\
\mathsf{recup}\ f\ i\ x\ y\ a &\to f\ i\ (\mathsf{recup}\ f\ (i+1)\ x\ y\ a) && [x \leq i \leq y]\\
\mathsf{recdown}\ f\ i\ x\ y\ a &\to f\ i\ (\mathsf{recdown}\ f\ (i-1)\ x\ y\ a) && [x \leq i \leq y]
\end{aligned}$$

We furthermore define $\mathsf{F}\ f\ i\ x\ y\ a \to a\ [i < x \vee i > y]$ for all $\mathsf{F} \in \{\mathsf{tailup}, \mathsf{taildown}, \mathsf{recup}, \mathsf{recdown}\}$.

What these recursors have in common is that in each call the iterator i is increased/decreased by 1 (executing its recursive call) until it surpasses the lower bound x or upperbound y (returning accumulator a). The same can be said about the examples in Figure 2. For example, Figure 2c is equivalent to $\mathcal{R} = \{\mathsf{factRD}\ i \to 1\ [i < 2],\ \mathsf{factRD}\ i \to i * (\mathsf{factRD}\ (i-1))\ [i \geq 2]\}$, satisfying this behavior (except that here, we have lower bound $x = 2$ but no upper bound y).

Let us consider downward recursion more abstractly, using the following template consisting of two rewrite rules (Figure 2c fits in by taking $C_{x,y}[i] = \mathsf{recdown}\ i$, $\mathsf{f} = *$, $D[i] = 1$, $x = 2$).

$$\begin{cases} C_{x,y}[i] \to D[i] & [i < x]\\ C_{x,y}[i] \to \mathsf{f}\ i\ C_{x,y}[i-1] & [i \geq x] \end{cases}$$

With RI we can prove that this template corresponds to the inductive theorem $C_{x,y}[i] \approx \mathsf{recdown}\ \mathsf{f}\ i\ x\ y\ D[z]\ [x \leq i \leq y \wedge z = x-1]$. For Figure 2c this yields $\mathsf{factRD}\ i \approx \mathsf{recdown}\ [*]\ i\ 2\ y\ 1\ [2 \leq i \leq y]$. The absence of an upper bound in the definition of factRD is reflected by the corresponding inductive theorem: variable y occurs only on the right-hand side and in the constraint. We can freely choose any y which satisfies $i \leq y$.

Table 1 summarizes the templates and corresponding inductive theorems for all the 4 types of recursion that we will consider.

Example 2.1. By renaming $x := y$ in Figure 2a we obtain the equivalent LCSTRS with rules $\mathsf{factTU}\ y \to \mathsf{u}\ y\ 1\ 1$, $\mathsf{u}\ y\ i\ a \to a\ [i > y]$, $\mathsf{u}\ y\ i\ a \to \mathsf{u}\ y\ (i+1)\ (i*a)\ [i \leq y]$. The u-rules fit into the tailup template of Table 1 (take $\mathsf{f} = *$ and $C_{x,y}[i,a] = \mathsf{u}\ y\ i\ a$). This yields an inductive theorem $\mathsf{u}\ y\ i\ a \approx \mathsf{tailup}\ [*]\ i\ x\ y\ a\ [x \leq i \leq y]$. Since $\mathsf{factTU}\ y \to_{\mathcal{R}} \mathsf{u}\ y\ 1\ 1$, we obtain the inductive theorem $\mathsf{factTU}\ y \approx \mathsf{tailup}\ [*]\ 1\ x\ y\ 1\ [x \leq 1 \leq y]$.

3 Proving inductive theorems with recursor templates

With Table 1 we can automatically generate inductive theorems, relating program definitions to one of the pre-defined recursors tailup, $\mathsf{taildown}$, recup and $\mathsf{recdown}$. To conclude program equivalence we in addition need to derive inductive theorems between the recursors themselves.

Lemma 3.1. $\mathsf{recdown}\ f\ y\ x\ n\ a \approx \mathsf{tailup}\ f\ x\ m\ y\ a\ [m \leq x \leq y \leq n]$ *is an inductive theorem.*

This lemma is easily proven with RI.

Example 3.1. Variable-renaming the inductive theorem from Example 2.1 yields $\mathsf{factTU}\ i \approx \mathsf{tailup}\ [*]\ 1\ x\ i\ 1\ [x \leq 1 \leq i]$. We also deduced $\mathsf{factRD}\ i \approx \mathsf{recdown}\ [*]\ i\ 2\ y\ 1\ [2 \leq i \leq y]$. Moreover, we easily prove the $*$-specific equation $\mathsf{recdown}\ [*]\ i\ 2\ y\ a \approx \mathsf{recdown}\ [*]\ i\ 1\ y\ a$, which gives us

$$\begin{aligned} \mathsf{factTU}\ i &\approx \mathsf{tailup}\ [*]\ 1\ x\ i\ 1 && [x \leq 1 \leq i] \\ \mathsf{factRD}\ i &\approx \mathsf{recdown}\ [*]\ i\ 1\ y\ 1 && [2 \leq i \leq y] \end{aligned}$$

fitting into Lemma 3.1 by substituting $[x := 1,\ m := x,\ y := i,\ a := 1,\ n := y]$. We obtain $\mathsf{factTU}\ i \approx \mathsf{factRD}\ i\ [x \leq 1 \leq 2 \leq i \leq y]$, or equivalently $\mathsf{factTU}\ i \approx \mathsf{factRD}\ i\ [i \geq 2]$.

The remaining recursor equivalences we can only prove conditionally: under assumption $f = \mathsf{f} \in \Sigma$ satisfies extra properties (here, we need commutativity/associativity). We collect our assumptions in a set $\mathcal{A}$ of axioms, required to be proven by a RI deduction $(\mathcal{A}, \emptyset) \vdash^* (\emptyset, \mathcal{H})$.

Definition 3.1 (Conditional inductive theorems)**.** Let $\mathcal{A}$ be a set of equations (axioms) and $\mathcal{E}$ be a set of equations. We define the *conditional inductive theorem* $\mathcal{A} \vDash^c \mathcal{E}$ as follows: "If there is a set $\mathcal{H}$ and a RI-deduction $(\mathcal{A}, \emptyset) \vdash^* (\emptyset, \mathcal{H})$ then every equation in $\mathcal{E}$ is an inductive theorem."

For $\mathsf{f} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \in \Sigma$, we define axiom sets $\mathsf{C(f)} = \{\mathsf{f}\ x\ y \approx \mathsf{f}\ y\ x\}$ and $\mathsf{AC(f)} = \{\mathsf{f}\ x\ (\mathsf{f}\ y\ z) \approx \mathsf{f}\ (\mathsf{f}\ x\ y)\ z,\ \mathsf{f}\ x\ y \approx \mathsf{f}\ y\ x\}$.

Lemma 3.2. *Let* $\mathsf{f} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \in \Sigma$*. The following are conditional inductive theorems*

$$\begin{aligned} \mathsf{AC(f)} &\vDash^c \mathsf{recdown}\ \mathsf{f}\ y\ x\ n\ a \approx \mathsf{recup}\ \mathsf{f}\ x\ m\ y\ a && [m \leq x \wedge x \leq y \wedge y \leq n] \\ \mathsf{AC(f)} &\vDash^c \mathsf{taildown}\ \mathsf{f}\ y\ x\ n\ a \approx \mathsf{tailup}\ \mathsf{f}\ x\ m\ y\ a && [m \leq x \wedge x \leq y \wedge y \leq n] \\ \mathsf{C(f)} &\vDash^c \mathsf{taildown}\ \mathsf{f}\ y\ x\ n\ a \approx \mathsf{recup}\ \mathsf{f}\ x\ m\ y\ a && [m \leq x \wedge x \leq y \wedge y \leq n] \\ \mathsf{AC(f)} &\vDash^c \mathsf{taildown}\ \mathsf{f}\ i\ x\ y\ a \approx \mathsf{recdown}\ \mathsf{f}\ i\ x\ y\ a \\ \mathsf{AC(f)} &\vDash^c \mathsf{tailup}\ \mathsf{f}\ i\ x\ y\ a \approx \mathsf{recup}\ \mathsf{f}\ i\ x\ y\ a \end{aligned}$$

With RI we easily show $(\mathsf{AC}(*), \emptyset) \vdash^* (\emptyset, \mathcal{H})$ for $\mathcal{H} = \emptyset$. Using Lemma 3.2 we derive the remaining inductive theorems in Figure 2, such as $\mathsf{factTD}\ x \approx \mathsf{factTU}\ x\ [x \geq 1]$.

Higher-order equivalences With Lemma 3.1, 3.2 we prove all equivalences from Figure 1. For higher-order equivalences, however, they no longer suffice. Consider the following higher-order variants of the LCSTRSs in Figure 2a and Figure 2b, both computing $(f, x) \mapsto \prod_{i=1}^{x} f(i)$

$$\begin{aligned} &\mathsf{funfactTU}\ f\ y \to \mathsf{u}\ f\ y\ 1\ 1 && && \mathsf{funfactTD}\ f\ i \to \mathsf{d}\ f\ i\ 1 \\ &\mathsf{u}\ f\ y\ i\ a \to a && [i > y] && \mathsf{d}\ f\ i\ a \to a && [i < 1] \\ &\mathsf{u}\ f\ y\ i\ a \to \mathsf{u}\ f\ y\ (i+1)\ ((f\ i) * a) && [i \leq y] && \mathsf{d}\ f\ i\ a \to \mathsf{d}\ f\ (i-1)\ (a * (f\ i)) && [i \geq 1] \end{aligned}$$

By Table 1 we obtain $\mathsf{funfactTD}\ f\ i \approx \mathsf{taildown}\ (\lambda a, i.(f\ i) * a)\ i\ 1\ y\ 1\ [1 \leq i \leq y]$ and $\mathsf{funfactTU}\ f\ y \approx \mathsf{tailup}\ (\lambda i, a.(f\ i) * a)\ 1\ x\ y\ 1\ [x \leq 1 \leq y]$. Here, λ is used as meta-language notation. For example, $\lambda a, i.(f\ i) * a$ denotes a function symbol $\mathsf{G}_f \in \Sigma$ defined by $\mathsf{G}_f\ a\ i \to (f\ i)*a$. Note that Lemma 3.2 does not apply: we do not even have $\lambda a, i.(f\ i)*a = \lambda i, a.(f\ i)*a$.

However, we can prove the equivalence once we have the following result

Lemma 3.3. *Let* $\mathsf{F} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \in \Sigma$. *The following are conditional inductive theorems*

$$\mathsf{AC}(\mathsf{F}) \vDash^c \mathsf{recdown}\ (\lambda i, a.\ (\mathsf{F}\ (f\ i)\ a))\ y\ x\ n\ a \approx \mathsf{recup}\ (\lambda i, a.\ (\mathsf{F}\ (f\ i)\ a))\ x\ m\ y\ a\ [m \le x \le y \le n]$$

$$\mathsf{AC}(\mathsf{F}) \vDash^c \mathsf{taildown}\ (\lambda a, i.\ (\mathsf{F}\ (f\ i)\ a))\ y\ x\ n\ a \approx \mathsf{tailup}\ (\lambda i, a.\ (\mathsf{F}\ (f\ i)\ a))\ x\ m\ y\ a\ [m \le x \le y \le n]$$

$$\mathsf{AC}(\mathsf{F}) \vDash^c \mathsf{taildown}\ (\lambda a, i.\ (\mathsf{F}\ (f\ i)\ a))\ i\ x\ y\ a \approx \mathsf{recdown}\ (\lambda i, a.\ (\mathsf{F}\ (f\ i)\ a))\ i\ x\ y\ a$$

$$\mathsf{AC}(\mathsf{F}) \vDash^c \mathsf{tailup}\ (\lambda i, a.\ (\mathsf{F}\ (f\ i)\ a))\ i\ x\ y\ a \approx \mathsf{recup}\ (\lambda i, a.\ (\mathsf{F}\ (f\ i)\ a))\ i\ x\ y\ a$$

$$\emptyset \vDash^c \mathsf{taildown}\ (\lambda a, i.\ (F\ (f\ i)\ a))\ y\ x\ n\ a \approx \mathsf{recup}\ (\lambda i, a.\ (F\ (f\ i)\ a))\ x\ m\ y\ a\ [m \le x \le y \le n]$$

4 Closing remarks

Constrained rewriting The facility for non-inductively defined primitive data structures is very specific to constrained rewriting. This made it possible to define recursors and templates being able to describe integer loops in a manner that is intuitively very close to real-life programming (where we can also treat the integers as being given for free). The templates defined here, we cannot define in ordinary higher-order rewriting. We specifically aimed at loop constructions having an integer counter i which is increased/decreased by 1 in each loop iteration. In future work we can further extend our existing templates or add new ones, e.g. generalizing our templates to increases/decreases by some arbitrary number k. We could also introduce recursors that iterate over lists, obtaining foldl and foldr. However, such recursors we can already define in ordinary higher-order rewriting.

Related & future work The idea to use templates for inductive theorem proving is not new. A comparable work for unconstrained first-order rewriting is [1], where the authors define templates to verify program transformations. In contrast to our approach, equivalence between templates is proven directly (i.e. no intermediate recursors like tailup are used) using the notion *equivalent term rewriting systems* (instead of using RI), which they can prove with specifically designed transformation rules. It seems that the underlying mechanism is fundamentally different, because their method assumes confluence (whereas RI relies on termination).

In future work we could investigate if we could benefit from this and other existing work on program transformations based on term rewriting, such as context moving transformations [6].

Implementation We recently implemented RI for LCSTRSs [5] in Cora (see `https://github.com/hezzel/cora`), and we are currently working on implementing the template method as well.

References

[1] Y. Chiba, T. Aoto, and Y. Toyama. Program transformation templates for tupling based on term rewriting. *IEICE TRANSACTIONS on Information and Systems*, E93-D(5):963–973, 2010.

[2] C. Fuhs, C. Kop, and N. Nishida. Verifying procedural programs via constrained rewriting induction. *ACM Transactions On Computational Logic (TOCL)*, 18(2):14:1–14:50, 2017.

[3] L. Guo and C. Kop. Higher-order LCTRSs and their termination. In *Proc. ESOP 24*, volume 14577 of *LNCS*, pages 331–357, 2024.

[4] K. Hagens and C Kop. Matrix invariants for program equivalence in lctrss. In *Proc. WPTE 23*, 2023.

[5] K. Hagens and C. Kop. Rewriting induction for higher-order constrained term rewriting systems. In *Proc. LOPSTR 24*, volume 14919, pages 202–219, 2024.

[6] K. Sato, K. Kikuchi, T. Aoto, and Y. Toyama. Correctness of context-moving transformations for term rewriting systems. In *Proc. LOPSTR 15*, volume 9527 of *LNCS*, pages 331–345, 2015.